Entwurf und Realisierung von Softwarebibliotheken: Massendaten im Tycoon-System

Diplomarbeit von
Björn Lotter
Stockflethweg 78
22417 Hamburg
und
Thorsten Römer
Harzburger Weg 12
22459 Hamburg

29. März 1996

Betreuer:
Prof. Dr. Joachim W. Schmidt
Prof. Dr. Heinz Züllighoven

Universität Hamburg
Fachbereich Informatik
Datenbanken und Informationssysteme

Erklärung

Hiermit erklären wir, daß wir die vorliegende Diplomarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.
Die Kapitel 1 bis 4 wurden von Thorsten Römer, die Kapitel 5 bis 8 von Björn Lotter geschrieben.

Hamburg, den 29. März 1996

(Björn Lotter) (Thorsten Römer)

Inhaltsverzeichnis

1 Einleitung **1**

2 Softwarebibliotheken **4**
2.1 Bedeutung von Softwarebibliotheken 5
2.2 Qualitätsmerkmale von Softwarebibliotheken 6
2.2.1 Wiederverwendbarkeit 9
2.2.2 Verständlichkeit 9
2.2.3 Übersichtlichkeit 10
2.2.4 Erweiterbarkeit 10

3 Grundstrukturen für Softwarebibliotheken **12**
3.1 Modulorientierte Softwarebibliotheken 12
3.1.1 Eigenschaften und Ziele 12
3.1.2 Schwächen modulorientierter Bibliotheken 15
3.1.2.1 Einschränkung der Wiederverwendung und Spezialisierung . 15
3.1.2.2 Mangelnde Unterstützung inkrementeller Erweiterung 17
3.1.2.3 Einschränkungen der Kombinierbarkeit 18
3.1.2.4 Fehlende Unterstützung der Standardisierung 18
3.1.2.5 Fehlende einfache Sicht auf komplexe Schnittstellen 19
3.1.3 Die Massendatentypen-Bibliothek von Tycoon 19
3.1.3.1 Struktur der Bibliothek 20
3.1.3.2 Vorgehensweise bei der Implementierung von Bibliotheken in Tycoon 20
3.1.4 Bewertung modulorientierter Bibliotheken 22
3.2 Objektorientierte Softwarebibliotheken 23
3.2.1 Terminologie 23
3.2.1.1 Objekte, Eigenschaften und Methoden 23
3.2.1.2 Klassen 24
3.2.1.3 Vererbung und Klassenhierarchie 25
3.2.2 Typisierung und Polymorphismus 26
3.2.2.1 Typisierung 27
3.2.2.2 Polymorphismus 28
3.2.3 Vorteile des objektorientierten Ansatzes für Bibliotheken 29
3.2.3.1 Orientierung an den Daten statt an Funktionen 29
3.2.3.2 Vorteile durch Vererbung 30
3.2.3.3 Kapselung 32

3.2.4 Formen der Vererbung 33
3.2.4.1 Implementierungsvererbung 33
3.2.4.2 Konzeptvererbung 34
3.2.4.3 Typvererbung 35
3.2.5 Ergebnis des Vergleichs der Grundstrukturen 35

4 Beispiele objektorientierter Softwarebibliotheken 36
4.1 C++ 37
4.1.1 Sprache 38
4.1.1.1 Mehrfachvererbung 38
4.1.1.2 Kapselung 38
4.1.1.3 Typsystem 39
4.1.1.4 Überschreiben von Methoden 42
4.1.2 Die C++-Bibliothek NIHCL 43
4.1.3 Bewertung der NIHCL 50
4.2 Vererbung: Ein „Produzentenmechanismus“? 53
4.2.1 Vererbungshierarchie 55
4.2.2 Schnittstellenhierarchie 55
4.2.3 Ergebnis 56
4.3 Smalltalk 57
4.3.1 Sprache 57
4.3.2 Bibliothek 58
4.3.3 Bewertung der Smalltalk-Bibliothek 58
4.4 Eiffel 59
4.4.1 Sprache 59
4.4.2 Die Eiffel-Bibliothek 61
4.5 Fazit 64

5 Bibliotheksentwurf in TL 67
5.1 Die Programmiersprache TL 67
5.2 Beispiele 68
5.2.1 Subtypbeziehung zwischen abstrakten Datentypen 68
5.2.2 Späte Bindung 70
5.2.2.1 Ausnutzung der dynamischen Parameterbindung 71
5.2.2.2 Ausnutzung rekursiver Bindung 73
5.3 Folgerungen 74

6 TooL: Eine objektorientierte Variante von TL 75
6.1 Kapselung 76
6.2 Mehrfachvererbung 76
6.3 Metaklassen 78
6.4 Abstrakte Klassen 79
6.5 Typisierung 80
6.6 Polymorphismus 83
6.7 Semantikspezifikation 83

7 Die TooL Behälterklassenbibliothek **85**
7.1 Klassifizierungsschema 86
7.2 Allgemeine Behältereigenschaften 87
7.2.1 Eigenschaftsunabhängiges Protokoll 89
7.2.2 Eigenschaften der Aufnahmefähigkeit 90
7.2.3 Zugriffsformen 92
7.2.4 Traversierungsformen 94
7.2.5 'Sortiertheit' 95
7.2.6 Sortierbarkeit 97
7.3 Behälterkategorien 98
7.3.1 Listen 98
7.3.2 Bäume 102
7.3.3 Weitere Behälterklassen 104
7.4 Implementierung 106
7.4.1 Implementierungsvarianten 106
7.4.2 Vermeidung von Redefinitionen 108
7.4.3 Erweiterbare Methodenketten 109
7.5 Generische Behälterklassen 111
7.6 Iterationsabstraktion 114
7.6.1 'Stream'-Varianten 115
7.6.2 Abgrenzung 116
7.7 Zwischenergebnis 118
7.8 Verwendung von Entwurfsmustern 121
7.8.1 Brückenklassen 121
7.8.2 Fabrikklassen 122
7.8.3 Anwendung 123
7.9 Entwurfshilfsmittel 128
7.9.1 Methodenakkumulation 128
7.9.2 Visualisierung der Vererbungsbeziehungen 130

8 Zusammenfassung und Ausblick **133**

Literaturverzeichnis **135**

Abbildungsverzeichnis

2.1 Hierarchie der die Qualität bestimmenden Eigenschaften (nach [Boehm et al. 78]) 8

3.1 Entstehung des Gesamtsystems aus aufeinander aufbauenden Komponenten (nach [Cohen 72]) 14
3.2 Erster Zerlegungsschritt (nach [Budgen 89]) 16
3.3 Zweiter Zerlegungsschritt (nach [Budgen 89]) 16
3.4 Ineffizienz der Kombination von Funktionalität 19
3.5 Bibliothek für Massendaten des Tycoon-Systems 21
3.6 Begriffsklärung: Objekt (nach [Otto 91]) 24
3.7 Objekte derselben Klasse (nach [Otto 91]) 25
3.8 Beispiel einer Klassenhierarchie (nach [Otto 91]) 26
3.9 Vererbung (nach [Otto 91]) 27
3.10 Formen von Polymorphismus (nach [Cardelli, Wegner 85]) 28
3.11 Konventionelle und objektorientierte Zerlegung eines Problems (nach [Otto 91]) 30
3.12 Drei Formen der Vererbung 35

4.1 Ausschnitt der NIHCL-Klassenbibliothek 43
4.2 Verteilung der Methoden in NIHCL 44
4.3 Kategorisierung der Klassen in NIHCL 51
4.4 Verstecken geerbter Methoden in NIHCL 53
4.5 Entwicklung der Protokollhierarchie (nach [Cook 92]) 56
4.6 Ausschnitt der Smalltalk-Klassenbibliothek 59
4.7 Schnittstellenhierarchie der Smalltalk-Bibliothek (nach [Cook 92]) 60
4.8 Ausschnitt der Eiffel-Klassenbibliothek 62
4.9 Verteilung der Methoden der Eiffel-Klassenbibliothek 63
4.10 Beispiel für das Entfernen von Methoden in der Eiffel-Bibliothek 64

5.1 Subtypisierung von abstrakten Datentypen 69
5.2 Schichtung von Spezifikation und Repräsentation 69
5.3 Virtuelle Methoden I. 70
5.4 Fortbestand der alten Bindung 71
5.5 Methoden mit **Self**-Parameter 72
5.6 Methoden mit Self-Parameter und Typanpassung 73
5.7 Rekursive Bindung statt **Self**-Parameter 74

6.1 Kontrollstrukturen als Botschaften 75
6.2 Kapselung 76

6.3 Mehrfachvererbung 77
6.4 Klassenpräzedenzliste (cpl) 77
6.5 Eine Metaklasse 78
6.6 Abstrakte Klassen 79
6.7 Verlust der Subtypbeziehung 81
6.8 Verlust der Ähnlichkeitsbeziehung 81
6.9 Rückgewinnung der Subtypbeziehung 82
6.10 Parametrischer Polymorphismus 83
6.11 Semantikspezifikation 84

7.1 Klassifizierungsschema 86
7.2 Allgemeine Eigenschaftsgruppen 88
7.3 Eigenschaftsunabhängiges Protokoll 89
7.4 Eigenschaften der Aufnahmefähigkeit 90
7.5 Klassifizierung der Zugriffsformen 92
7.6 Eigenschaftsgruppe Traversierungsformen 94
7.7 Ordnungsrelation für Objekte 96
7.8 Sortierte Behälter 96
7.9 Sortierbarkeit und sortierungserhaltendes Mischen 97
7.10 Ausnutzung des Ähnlichkeitspolymorphismus 99
7.11 Die Behälterkategorie Liste 100
7.12 Kombination spezieller Listeneigenschaften 101
7.13 Nutzung der geerbten Implementierung 102
7.14 Bäume für geordnete Elemente 103
7.15 Arrays 104
7.16 Kettungsabstraktion 107
7.17 Funktionalitätsverteilung für die Methode '=' 111
7.18 In Subtypbeziehung stehende Elementtypen 112
7.19 Stream-Varianten 115
7.20 Entflechtung von Implementierung und Abstraktion: Brückenbildung (aus [Gamma et al.,S.153]) 122
7.21 Kontextbezogene Objekterzeugung: Abstrakte Fabrik (nach [Gamma et al. 95, S.88]) 123
7.22 Listen mit separater Repräsentationsimplementation 124
7.23 Erzeugung von Listenvarianten: Listenfabriken 126
7.24 Fabrikation 126
7.25 Anzeige der Klassendefinition : Kurzform 128
7.26 Anzeige der Klassendefinition: akkumulierende Form 129
7.27 Eine Klasse als WWW-Seite 130
7.28 Die Behälterklassenbibliothek im WWW 131

Kapitel 1

Einleitung

Bei der Programmierung großer Systeme müssen immer wieder Design-Entscheidungen getroffen werden, Funktionalität in ein System einzubauen oder sie durch einen auf Wiederverwendung und Systemschichtung beruhenden Prozeß zur Verfügung zu stellen. Auch bei der Entwicklung von Programmiersprachen steht der Sprachdesigner vor einer ähnlichen Frage. Datentypen können in die Sprache eingebaut (*Built-in*-Ansatz) oder wegen der größeren Flexibilität in Form von Softwarebibliotheken angeboten werden (*Add-on*-Ansatz) [Schmidt, Matthes 91].

Im Rahmen einer Dissertation [Matthes 93] entstand am Arbeitsbereich DBIS das Tycoon-System[1] [Mathiske et al. 93] als persistente polymorphe Programmierumgebung zur Realisierung datenintensiver Anwendungen. Das Tycoon-System verfügt über eine Softwarebibliothek, die unter anderem zur Verwaltung von Massendatentypen [Atkinson et al. 90] geeignet ist und in TL[2] [Matthes, Schmidt 92], der polymorphen Programmiersprache des Systems, implementiert wurde.

Da die Bibliothek von einer größeren Gruppe über einen längeren Zeitraum ohne eine zentrale, das Projekt steuernde Instanz entwickelt wurde, mangelt es der Bibliothek historisch gewachsen an der notwendigen Einheitlichkeit. Mit der Zahl der an der Bibliothek arbeitenden Personen stieg auch die Zahl der unterschiedlichen Konzepte. Vor diesem Hintergrund wurde 1995 ein Projekt zur Neugestaltung der Bibliothek des Tycoon-Systems ins Leben gerufen. [Jung 95] untersuchte dabei die Bibliothek für Standarddatentypen, während die vorliegende Arbeit die Weiterentwicklung der in der Bibliothek des Tycoon-Systems zahlreich vorkommenden Module zur Verwaltung von Massendatentypen darstellt.

Ein wesentliches Ziel der Entwicklung von Softwarebibliotheken ist es, qualitativ hochwertige Software anzubieten. Fragen der Wiederverwendbarkeit, Übersichtlichkeit und Kombinierbarkeit von Software spielen dabei eine besondere Rolle. Durch Wiederverwendbarkeit von Software, die im Falle von Bibliotheken wesentlich auf Übersichtlichkeit und Kombinierbarkeit basiert, können nicht nur die Kosten von Softwareprojekten gesenkt, sondern auch die Softwarequalität entscheidend erhöht werden. Ein Grund hierfür ist die steigende Lebensdauer der Softwarekomponenten, die das Auffinden und Beheben von Mängeln begünstigt.

Ein modulorientierter Entwurfsansatz ist eine Möglichkeit, Wiederverwendbarkeit in Softwarebibliotheken zu realisieren. Module bieten dem Benutzer über eine Schnittstelle Funktionalität an. Die interne Realisierung dieser Funktionalität inklusive der internen Repräsenta-

[1] **T**yped **c**ommunicating **o**bjects in **o**pen e**n**vironments

[2] **T**ycoon **L**anguage

tion der Daten bleibt dem Benutzer verborgen.

Die in modulorientierten Sprachen häufig anzufindende Trennung von Definition und Implementierung, verbunden mit dem Konzept der Kapselung, führt zu einem typischen Vorgehen bei der Entwicklung von Bibliotheken für Massendatentypen. Durch das Konzept des abstrakten Datentyps wird die Kombination von Funktionalität in modulorientierten Sprachen erheblich erschwert. Ein typisches Modul einer Massendatentypen-Bibliothek realisiert seine Funktionalität unter Verwendung eines anderen abstrakten Datentyps zur Speicherung der Daten. Die Schichtenbildung aus abstrakten Datentypen beschränkt sich in der Regel auf zwei Ebenen. Somit führt eine derartige Organisation von Bibliotheken zu einer Vielzahl unverknüpft nebeneinander stehender Modulpaare. Die Wiederverwendung innerhalb einer modulorientierten Bibliothek ist damit relativ gering. Eine Kombination der von den Modulen angebotenen Funktionalität ist nicht effizient möglich.

Objektorientierte Sprachen erweitern die Möglichkeiten der Wiederverwendung gegenüber modulorientierten Sprachen um das Konzept der Vererbung. Neben der aus modulorientierten Sprachen bereits bekannten Benutztbeziehung besitzen die Klassen einer objektorientierten Sprache auch die Möglichkeit der Wiederverwendung von Konzepten und Implementierungen innerhalb einer Bibliothek durch die Vererbungsbeziehung. Vererbung gibt dem Bibliotheksentwickler somit die Möglichkeit, die zu implementierenden Klassen zunächst nur auf einer konzeptuellen Ebene zu modellieren und diese Konzepte schrittweise zu spezialisieren. Durch die von manchen objektorientierten Sprachen angebotene Mehrfachvererbung wird zusätzlich die Kombination von Funktionalität innerhalb der Bibliothek ermöglicht.

Durch diese Erweiterung der Sprachkonzepte wird die Wiederverwendung auch innerhalb der Bibliothek unterstützt, statt sie nur dem Benutzer der Bibliothek zukommen zu lassen. Als wesentlicher Vorteil für den Benutzer ergibt sich eine einheitliche und einfache Sicht auf komplizierte Objekte sowie eine Hilfe bei der Auswahl der zu verwendenden Klassen, da Klassen die Eigenschaften der von ihnen erbenden Klassen abstrakt zusammenfassen und auf diese Weise die Eigenschaften der Klassen der Bibliothek klassifizieren.

Angesichts der erheblichen Vorteile objektorientierter gegenüber modulorientierten Bibliotheken wird von einer bloßen Überarbeitung der bestehenden modulorientierten Bibliothek zur Verwaltung von Massendaten im Tycoon-System abgesehen. Ziel der Arbeit ist statt dessen die Entwicklung einer neuen, diese Vorteile nutzende Bibliothek. Um Vorgehensweisen, Stärken aber auch Schwächen bestehender Bibliotheken erkennen zu können, erfolgt im Rahmen der Arbeit eine Untersuchung mehrerer objektorientierter Sprachen und ihrer Bibliotheken zur Verwaltung von Massendatentypen. Verglichen werden daher die Sprache C++ [Stroustrup 92] und die in C++ geschriebene Bibliothek NIHCL[3] sowie die Sprachen Smalltalk [Goldberg, Robson 83] und Eiffel [Meyer 92] mit ihren Bibliotheken.

Die Analyse dieser objektorientierten Bibliotheken zeigt den erheblichen Einfluß von Spracheigenschaften auf die Gestaltung von Bibliotheken und deckt einige Schwächen auf, die aus einer unvorsichtigen Verwendung von Vererbung resultieren. Der mit Hilfe von Objektorientierung erreichbare Vorteil einer einheitlichen Sicht auf komplizierte Objekte wird von den Bibliotheken verfehlt, was als wesentliche Schwäche der untersuchten objektorientierten Bibliotheken zu vermerken ist.

Ausgehend von gezeigten Vorteilen objektorientierter Modellierung wird versucht, die objektorientierte Programmierung auch in der modulorientierten Sprache des Tycoon-Systems zu realisieren. Wegen der geringen Unterstützung objektorientierter Programmierung in der

[3] **N**ational **I**nstitute of **H**ealth **C**lass **L**ibrary

Sprache TL entstand als Forschungsprojekt die objektorientierte TL-Variante TooL[4] [Gawecki, Matthes 95]. Neben Klassen und Vererbung bietet das Typsystem von TooL die Möglichkeit zur kontravarianten Spezialisierung von Methodenparametern. Dies wird durch die Integration einer weiteren Typrelation, der Ähnlichkeitsrelation (*type matching*) erreicht.

Die im Rahmen dieser Arbeit geleistete vollständige Neuentwicklung, Erweiterung und Implementierung der Tycoon-Bibliothek zur Verwaltung von Massendatentypen erfolgt in der neuen, prototypischen Sprache TooL. Gesichtspunkte der Wiederverwendung innerhalb der Bibliothek sowie das Angebot einer einfachen Schnittstelle auch für komplexe Objekte stehen hierbei im Vordergrund. Durch die Entwicklung eines neuartigen Klassifizierungsschemas zur Spezialisierung und Kombination der in der Bibliothek angebotenen Funktionalität kann unter Ausnutzung der Möglichkeiten des Typsystems der Sprache TooL eine Bibliothek realisiert werden, welche die Schwächen der untersuchten Bibliotheken vermeidet.

Kapitel 2 greift zunächst die Abwägung zwischen fest in das System eingebundener Funktionalität und der Verwendung einer Softwarebibliothek auf. Nach der Erläuterung der Bedeutung von Bibliotheken beschreibt Kapitel 2 die Qualitätsanforderungen an Software und zeigt, daß an die Qualität von Softwarebibliotheken besondere Anforderungen zu stellen sind. Ausgehend von diesen Erkenntnissen vergleicht Kapitel 3, ob und inwieweit Modul- bzw. Objektorientierung die Entwicklung qualitativ hochwertiger Bibliotheken unterstützt. Nach der Beschreibung der Vorteile objektorientierter Bibliotheken erfolgt in Kapitel 4 eine Analyse bestehender objektorientierter Bibliotheken. Kapitel 5 untersucht die Eignung von TL, der ursprünglichen Programmiersprache des Tycoon-Systems, für die Entwicklung einer die Vorteile der Objektorientierung realisierenden Software-Bibliothek. Hierbei zeigt sich, welche Möglichkeiten, aber auch welche Grenzen die Sprache TL für die objektorientierte Programmierung aufweist. Eine Einführung in die für die vollständige Neuentwicklung verwendete Programmiersprache TooL findet sich in Kapitel 6. Kapitel 7 beschreibt die im Rahmen der Arbeit gestaltete und implementierte Behälterklassen-Bibliothek der Sprache TooL, ihre Konzepte und das dabei verwendete Klassifzierungsschema.

[4] **T**ycoon **o**bject-**o**riented **L**anguage

Kapitel 2

Softwarebibliotheken

Mit wachsendem Umfang von Softwareprodukten wird es zunehmend schwieriger, Software monolithisch in einer geschlossenen Struktur zu entwickeln. Bei einer monolithischen Architektur eines Softwaresystems ist für jedwede Änderung ein Überblick über das gesamte System notwendig. Hinzu kommt, daß bei der anschließenden Übersetzung des Systems dieses in vollem Umfang neu übersetzt werden muß, was zu langen Laufzeiten des Übersetzers führt.

Weiterhin entfällt ein großer Teil der durch ein Programm verursachten Kosten nicht auf die reinen Entwicklungskosten, sondern auf Weiterentwicklung und Wartung [Gladney 82]. Mit der Größe des Programms und der Wartung steigen aber nicht nur die Kosten, sondern auch die Gefahr von Softwarefehlern.

Die Notwendigkeit von Softwareveränderungen nach dem Entdecken eines Fehlers liegt auf der Hand. Parallel mit dem Vordringen in ständig neue Anwendungsbereiche werden Softwaresysteme jedoch auch mit neuen Anforderungen in Form von zusätzlich geforderter Funktionalität oder mächtigeren Datenstrukturen konfrontiert. Diese Änderungen der Aufgabenstellung führen regelmäßig zu Änderungen von Software.

Der Grund für die Anpassung alter Software an neue Anforderungen sind die Kosten der Neuentwicklung größerer Anwendungen. Die immensen Kosten einer Neuentwicklung erzwingen in der Regel eine Anpassung von bestehender Software an neue Bedingungen. [Lientz, Swanson 79] geben an, daß 41,8% der Wartungskosten durch Änderungen der Benutzeranforderungen entstehen.

Bei der Untersuchung von Systemen zeigt sich, daß zusätzliche Anforderungen zu zwei grundsätzlich verschiedenen Reaktionen des Anbieters des Softwareprodukts führen können. Eine Möglichkeit, zusätzliche Funktionalität anzubieten, besteht darin, diese in das System zu integrieren (*Built-in*-Ansatz). Ein typischer Vertreter für diese Vorgehensweise ist z.B. die an diesem Arbeitsbereich entwickelte relationale Datanbankprogrammiersprache DBPL[1] [Schmidt et al. 88].

In der Sprache DBPL sind das Datenmodell, die Strukturen zur Verwaltung der Massendaten und die Funktionalität zur Unterstützung datenintensiver Anwendungen fest in das Datenbanksystem eingebaut (*built-in*) [Niederée 92].

[1] **D**ata**b**ase **P**rogramming **L**anguage

2.1 Bedeutung von Softwarebibliotheken

Bereits Ende der 60er Jahre [Schoett 81] richtete sich das Interesse auf die Entwicklung von Verfahren, mit denen sich der Prozeß der Softwareentwicklung verbessern läßt, so daß der Arbeitsaufwand bei der Erstellung sinkt und zugleich die Entwicklung qualitativ hochwertiger Software unterstützt wird.

Dieses Kapitel erläutert zunächst die wachsende Bedeutung von Softwarebibliotheken. Als Alternative wird der Verwendung von Bibliotheken ein monolithischer Ansatz gegenüber gestellt, bei dem die gesamte Funktionalität in das System direkt eingebaut ist (*Built-in*-Ansatz). Die Vorteile, Funktionalität durch die Verwendung einer Bibliothek bei Bedarf einem System hinzufügen zu können (*Add-on*-Ansatz), spielen dabei eine besondere Rolle.

Die gesamte Funktionalität in das System fest einzubauen führt zu Problemen. Eine dynamische Anpassung und systematische Erweiterung des Systems als Reaktion auf neue Anforderungen ist in der Regel nicht mehr oder nur unter großen Schwierigkeiten möglich.

Einen höheren Grad an Flexibilität in Bezug auf die Erweiterbarkeit von Systemen bieten Softwarebibliotheken, deren Dienste bei Bedarf in das System eingefügt werden können (*Add-On*-Ansatz), anstatt sie fest einzubauen [Matthes, Schmidt 91]. Softwarebibliotheken sind Sammlungen von Softwarekomponenten, die ihre Funktionalität dem Benutzer zur Wiederverwendung in seinen Projekten anbieten. Die vorliegende Arbeit beschränkt sich dabei auf die Untersuchung von Bibliotheken, die Strukturen zur Verwaltung von Massendatentypen (*bulk data types*) anbieten.

Die Verwendung von Bibliotheken stellt damit eine Möglichkeit zur Senkung der durch Software verursachten Kosten dar, da durch die erneute und mehrfache Verwendung bereits erstellter Softwarekomponenten aus der Bibliothek in Projekten der Anwender Entwicklungsarbeit eingespart werden kann.

Softwarebibliotheken bieten eine Verbesserung der Erweiterbarkeit, Flexibilität und Skalierbarkeit von Systemen [Niederée 92]. Die Verwendung von Bibliotheken erlaubt eine flexible Reaktion im Falle neuer Anforderungen. Die Funktionalität des Systems kann durch die zusätzliche Verwendung von Funktionalität aus der Bibliothek erhöht werden, wobei immer nur der gerade benötigte Teil der Bibliothek in das System geladen werden muß. Ein hohes Maß an Kombinierbarkeit der Dienste der Bibliothek ist hierfür Voraussetzung.

Die Möglichkeit, Dienste einer Softwarebibliothek in ein Programm übernehmen zu können, bedeutet auch für die Phase des Entwurfs und Designs eines Systems eine erhebliche Vereinfachung. Bei dem Teil der Funktionalität, der aus der Bibliothek in das System übernommen wird, kann der Entwerfende von den Einzelheiten abstrahieren. Somit erleichtert die Bibliothek die Konzentration auf die besonderen Aufgaben des zu entwerfenden Systems, da von der konkreten Realisierung der Standardfunktionalität aus der Bibliothek abstrahiert werden kann.

Bei der Verwaltung von Massendaten ist in der Regel eine Optimierung nötig, um die Effizienz des Systems zu gewährleisten. Die Konzentration der Funktionalität zur Verwaltung der Massendaten in einer Bibliothek bedeutet, daß bei der Entwicklung von Systemen auf bereits optimierte Funktionalität zurückgegriffen werden kann. Der Anwender spart somit nicht nur den Aufwand der Erstellung von Funktionalität, sondern auch die Optimierung derselben. Dies ist dadurch besonders kosteneffizient, daß die Bibliothek nur einmal erstellt, aber mehrfach verwendet wird. Gegenüber dem *Built-in*-Ansatz werden also vielfache Entwicklungs- und Optimierungskosten eingespart.

Ein weiterer Vorteil der Verwendung von Bibliotheken liegt in der Erhöhung der Sicherheit.

Da Bibliotheken in zahlreichen Projekten zum Einsatz kommen, steigt mit der Wiederverwendung auch die Wahrscheinlichkeit der Entdeckung von Fehlern. Von der in einer Bibliothek angebotenen Software ist daher ein höheres Maß an Korrektheit zu erwarten. Tabelle 2.1 faßt die Gegenüberstellung beider Ansätze zusammen.

	Built-In	**Add-On**
Funktionalität	fest eingebaut	erweiterbar
Erweiterbarkeit	schwierig	durch Verwendung zusätzlicher Dienste der Bibliothek
Wiederverwendbarkeit von Komponenten	sehr gering	hoch
Optimierung	beim Systementwurf	bei der Implementierung der Bibliothek
Sicherheit	gering	hoch durch Verwendung erprobter Komponenten

Tabelle 2.1: *Add-On* versus *Built-In*

2.2 Qualitätsmerkmale von Softwarebibliotheken

Der Begriff der Qualität wird in der Literatur folgendermaßen definiert: *Qualität ist Übereinstimmung mit Anforderungen* [Crosby 72]. Dieser Begriff ist damit subjektiv bzw. kontextabhängig [Cavano, McCall 78]. Was für eine Person eine Steigerung der Qualität bedeuten mag, kann für die nächste eine Verschlechterung bedeuten [Weinberg 92], abhängig davon, welche Anforderungen Personen an Software stellen.

Um Qualität trotz Personenabhängigkeit meßbar zu machen, wurden zahlreiche Verfahren entwickelt. Eine Zusammenfassung findet sich in [Höcker et al. 84, S. 3]. Darüber hinausgehend schlägt [McCall et al. 81] vor, die Messung der Softwarequalität zu automatisieren.

Durch den Einsatz in zahlreichen Projekten ist die Qualität von Softwarebibliotheken von besonders großer Bedeutung. Gleichzeitig fordert die geplante Verwendung der Bibliothek durch fremde, nicht an der Entwicklung der Bibliothek beteiligte Personen, die Erfüllung besonderer Qualitätsmerkmale durch die Bibliothek. Bibliotheken und die Art ihrer Gestaltung sind daher von großer Bedeutung für die Unterstützung der Entwicklung qualitativ hochwertiger Software.

Durch eine Verbesserung der Softwarequalität können die Gesamtkosten von Softwareprojekten gesenkt werden. Laut [Cho 80] machen bei herkömmlichen Verfahren der Softwareerstellung die Aufwendungen für das Testen und die Integration der Software in das Gesamtsystem etwa die Hälfte der Gesamtkosten eines Projekts aus. Eine Studie [Cho 80] zeigt, daß durch höhere Aufwendungen zu Gunsten der Qualität in der Phase der Analyse und des Designs eines Systems erhebliche Kosten in den Bereichen Test und Anpassung eingespart werden können.

In der Literatur finden sich zahlreiche verschiedene Einteilungen der Qualitätsfaktoren. In

[Cho 80] werden die Faktoren beispielsweise danach unterschieden, ob sie für einzelne Softwareprodukte oder für ganze Softwaresysteme wichtig sind. Erst für ganze Systeme verlangt [Cho 80] Qualitätsmerkmale wie Strukturiertheit, Robustheit und Verständlichkeit, obwohl sich diese Begriffe auch auf einzelne Softwareprodukte anwenden ließen.

[Meyer 88] hingegen teilt die Qualitätsmerkmale in innere und äußere Faktoren ein. Als äußere Qualitätsfaktoren werden diejenigen Faktoren zusammengefaßt, die für einen Betrachter von außen, also ohne Betrachtung des Programmtextes, erkennbar sind. Zu diesen Qualitätsmerkmalen gehören sowohl der Grad der Korrektheit der Software als auch die Möglichkeiten der Anpassung an neue und unvorhergesehene Umweltbedingungen. Bei der Anpassung an neue Umweltbedingungen wird der Fall einer unvorhergesehen Situation im laufenden Betrieb von einer neuen Benutzeranforderung unterschieden. Innere Qualitätsfaktoren sind hingegen Faktoren, die nur unter Betrachtung des Quelltextes der Software festzustellen sind. [Meyer 88] nennt Modularität, Lesbarkeit und Verständlichkeit als innere Qualitätsfaktoren.

Diese Einteilung scheint jedoch angreifbar zu sein, da [Meyer 88] die Qualitätsfaktoren Wiederverwendbarkeit und Erweiterbarkeit den äußeren Faktoren zuordnet. Ein Blick in den Quelltext eines Programms wird sich aber kaum vermeiden lassen, wenn festgestellt werden soll, ob das Produkt wiederverwendbar ist. Weiterhin paßt ein Teil der von [Meyer 88] genannten Qualitätsfaktoren nicht zur Einteilung in innere und äußere Faktoren. Diese Faktoren werden deshalb weitere Faktoren genannt.

Die Qualitätsfaktoren für Software sind nicht unabhängig voneinander. Sie bauen vielmehr aufeinander auf und lassen sich in einer Hierarchie darstellen. Abbildung 2.1 zeigt die von [Boehm et al. 78] vorgeschlagene Hierarchie. Die in der Hierarchie weiter oben stehenden Qualitätsfaktoren implizieren die darunter eingezeichneten. So bedeutet beispielsweise eine hohe Wartbarkeit notwendigerweise auch eine hohe Testbarkeit und Verständlichkeit. Testbarkeit und Verständlichkeit sind somit die Grundlage für Wartbarkeit. [Boehm et al. 78] wählt für die Darstellung der Beziehungen der einzelnen Qualitätsfaktoren einen zyklischen Graphen. Dadurch wird betont, daß viele der Qualitätsfaktoren mehrere Ziele gleichzeitig unterstützen. So erhöht beispielsweise die Strukturiertheit von Software gleichzeitig deren Testbarkeit, Verständlichkeit und Anpassbarkeit. Diese drei durch die Strukturiertheit unterstützten Eigenschaften werden anschließend unter dem Begriff der Wartbarkeit zusammengefaßt.

Einen ähnlichen Ansatz, allerdings unter Verwendung eines azyklischen Graphen, verfolgt [Dunn 93]. Er nennt Zuverlässigkeit, Benutzbarkeit, Wartbarkeit und Anpassbarkeit als grundsätzliche Qualitätsmerkmale. Diese Merkmale werden dann in Untermerkmale zerlegt. Den Begriff der Anpassbarkeit zerlegt er beispielsweise in die Qualitätsaspekte Veränderbarkeit, Erweiterbarkeit und Portabilität.

Das zweifellos wichtigste Qualitätsmerkmal für Software ist die Korrektheit. Korrektheit ist die Fähigkeit von Softwareprodukten, ihre Aufgaben exakt zu erfüllen, wie sie durch Anforderungen und Spezifikationen definiert sind [Meyer 88]. Doch obwohl diese Qualität alle anderen Anforderungen an Software dominiert, ist die Korrektheit nicht ohne Schwierigkeiten zu bestimmen. Die Feststellung der Korrektheit setzt den Vergleich des tatsächlichen Verhaltens mit den aufgestellten Anforderungen voraus. Das Verhalten von Software mit formalen Mitteln eindeutig zu spezifizieren ist jedoch nur mit großem Aufwand möglich.

Natürlich stellt ein Benutzer auch an eine Bibliothek die Anforderungen, die an einzelne Softwarekomponenten gestellt werden. Hierzu gehören neben der Korrektheit insbesondere die auf der folgenden Seite genannten Merkmale [Meyer 88].

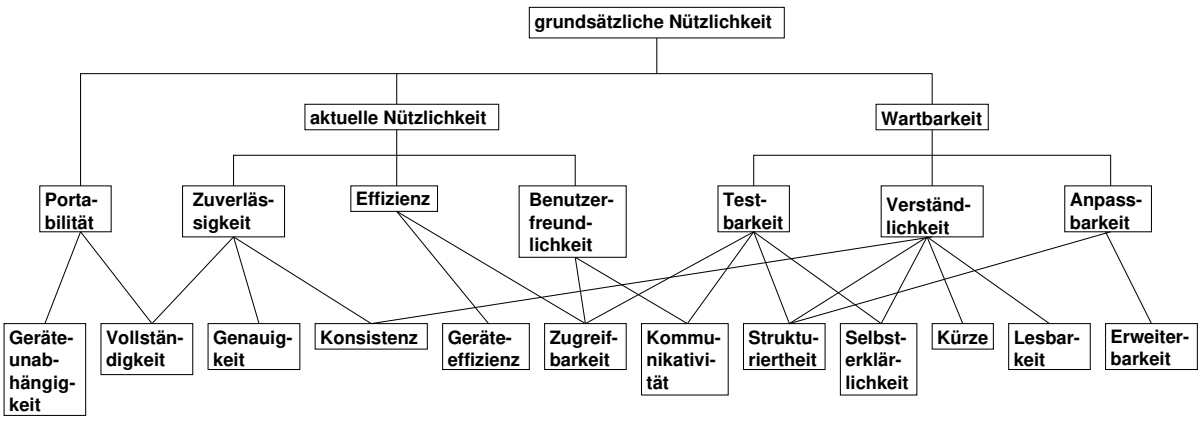

Abbildung 2.1: Hierarchie der die Qualität bestimmenden Eigenschaften (nach [Boehm et al. 78])

Robustheit ist die Fähigkeit von Softwaresystemen, auch unter außergewöhnlichen Bedingungen zu funktionieren. Robustheit bezeichnet damit den Grad der Angemessenheit, in der ein Softwaresystem in einer außerhalb der Spezifikation liegenden Situation im laufenden Betrieb reagiert. Obwohl die Spezifikation für das Verhalten in der maßgebenden Situation keinerlei Forderungen aufstellt, ist dieses nicht völlig beliebig. Qualitativ hochwertige Software zeichnet sich dadurch aus, daß sie auch in nicht von der Spezifikation erfaßten besonderen Situationen ein angemessenes und konstruktives Verhalten zeigt. Das Auftreten einer nicht vorhergesehenen Situation darf also bei einem Softwaresystem nicht zu unangemessenen, destruktiven Aktionen führen.

Effizienz ist ein Maß für den Aufwand zum Erreichen des Ziels. Bezogen auf Bibliotheken versteht man unter Effizienz den Speicher- und Platzbedarf der Algorithmen.

Portabilität bezeichnet die Leichtigkeit, mit der sich Software in eine andere Rechnerumgebung übertragen läßt.

Wartbarkeit meint das Maß der Leichtigkeit, mit der sich Software korrigieren oder anpassen läßt.

Verifizierbarkeit bezeichnet das Maß der Leichtigkeit, mit dem sich die Übereinstimmung von Spezifikation und tatsächlichem Verhalten von Software testen läßt.

Integrität bezeichnet den Schutz der Software vor unberechtigten Zugriffen oder Veränderungen.

Kompatibilität (Verträglichkeit) ist das Maß der Leichtigkeit, mit der Softwareprodukte mit anderen verbunden werden können.

Benutzerfreundlichkeit ist, obwohl hier (wieder mal) an letzter Stelle aufgezählt, von großer Wichtigkeit für die bei der Schulung der Endanwender anfallenden Kosten.

Da eine Bibliothek jedoch eine Sammlung von Funktionalität ist, müssen für die Qualität einer Bibliothek weitere Anforderungen der Benutzer beachtet werden, die über jene an die einzelnen Bestandteile der Bibliothek hinausgehen.

Die Anforderungen an Bibliotheken steigen auch dadurch, daß bei ihnen in der Regel die Entwickler der Bibliothek nicht mit der Personengruppe übereinstimmt, die das Projekt bearbeitet, bei dem die Bibliothek eingesetzt wird. Anforderungen wie Übersichtlichkeit und Verständlichkeit rücken damit gegenüber den Anforderungen anderer Software besonders in den Vordergrund, um die Einsetzbarkeit der Bibliothek in Projekten dritter Personen zu gewährleisten.

2.2.1 Wiederverwendbarkeit

Wiederverwendbarkeit ist die Eigenschaft von Software, ganz oder teilweise in neuen Softwareprodukten erneut verwendet werden zu können [Meyer 88]. Softwarebibliotheken stellen somit Investitionen in zukünftige Projekte dar. Wiederverwendung von Software ist unter zwei Gesichtspunkten interessant:

- Wiederverwendung kann die Kosten von Softwareprodukten senken, wenn für neue Anwendungen Teile einer alten Anwendung erneut verwendet werden.
- Wiederverwendung kann die Korrektheit von Softwareprodukten erhöhen, da bei einer Wiederverwendung die Lebensdauer der erneut verwendeten Teile der Software steigt und somit die Wahrscheinlichkeit wächst, darin eventuell enthaltene Mängel zu entdecken [Card et al. 86].

Wiederverwendbarkeit von Software ist damit für die Qualität von sehr hoher Bedeutung. Hilfsmittel können die Auswahl und das Auffinden von Softwarebausteinen erleichtern. Im Rahmen dieser Arbeit entstand deshalb der in Abschnitt 7.9 beschriebene TooL-Browser, der es dem Benutzer erlaubt, mit Hilfe von HTML-Seiten[2] [Koch 96] durch die Bibliothek zu navigieren.

2.2.2 Verständlichkeit

Der Wert einer Softwarebibliothek hängt entscheidend davon ab, daß ihre Verwendung ohne längere Einarbeitungszeit möglich ist. Die Bibliothek muß ihre Funktionalität daher verständlich anbieten. [Jung 95] beschreibt, wie eine Dokumentation zweckmäßig durchzuführen ist. Zur Dokumentation eines Softwarebausteins können beispielsweise Zustandsgraphen verwendet werden. Auch Vor– und Nachbedingungen der Dienste der Bibliothek sind von Bedeutung.

Die Dokumentation einer Softwarekomponente richtet sich sowohl an den Benutzer wie auch an den Softwareentwickler, der Änderungen an der Komponente vornimmt. Insbesondere Informationen über die hinter der Funktionalität stehenden Konzepte und Abhängigkeiten von anderen Komponenten erleichtern das für Änderungen notwendige Verständnis. Auch Konventionen bei Benennungen erleichtern das für Änderungen nötige Verständnis. Die Verständlichkeit von Bibliotheken steigt auch durch möglichst geringe Abhängigkeiten zwischen ihren einzelnen Komponenten.

Weiterhin ist darauf zu achten, daß ein Verständnis der Bibliothek ohne die Kenntnis aller Details und Verfahren möglich bleibt. Abstraktionsmechanismen sind bei der Verwendung

[2] **H**yper **T**ext **M**arkup **L**anguage

von Bibliotheken nützlich. Sie können helfen, die große Menge angebotener Funktionalität zu organisieren und auf diese Weise verständlich anzubieten. Abstraktionsmechanismen ermöglichen es, zunächst von (unwichtigen) Details abzusehen und sich auf die Grobstruktur der Information zu konzentrieren. Ein üblicher Abstraktionsmechanismus ist beispielsweise Klassifizierung.

Auch die Anwendung des Geheimnisprinzips bei der Gestaltung der Schnittstellen der Bibliothek ist eine Möglichkeit zur Steigerung der Verständlichkeit. Das Geheimnisprinzip besagt, daß Benutzer die Art und Weise, in der eine Funktionalität erbracht wird, nicht zu kennen brauchen, sondern nur deren Schnittstelle betrachten. Auch eine Trennung von Definition und Implementierung kann helfen, die Verständlichkeit zu erhöhen.

2.2.3 Übersichtlichkeit

Eine wesentliche Voraussetzung für eine wirtschaftliche Wiederverwendbarkeit ist die Übersichtlichkeit von Bibliotheken. Nur wenn der Benutzer der Bibliothek schnell feststellen kann, wo sich welche Art von Funktionalität in der Bibliothek befindet, kann er die Bibliothek für seine Arbeit sinnvoll einsetzen. Zur Steigerung der Übersichtlichkeit einer Bibliothek gibt es mehrere Möglichkeiten:

- Dokumentation: Übersichtliche Dokumentation der Bibliothek, ihrer Struktur und der hinter der Bibliothek stehenden Konzepte.

- Konventionen: Einheitliche Benennungen von Funktionen, Prozeduren oder Methoden.

- Vererbung: Objektorientierte Sprachen verfügen mit dem Mechanismus der Vererbung auch gleichzeitig über eine Möglichkeit, eine Bibliothek zu strukturieren und für den Benutzer übersichtlich zu gestalten.

Vererbung unterstützt Wiederverwendung auch innerhalb der Bibliothek. Speziell durch diese Form der Wiederverwendung kann die Übersichtlichkeit der Bibliothek für den Benutzer erheblich gesteigert werden. Voraussetzung ist jedoch, daß die Wiederverwendung sich auf die in der Bibliothek enthaltenen Konzepte bezieht. Bei geschickter Anwendung kann Wiederverwendung durch ihre klassifizierende Wirkung innerhalb der Bibliothek auf diese Weise das Auffinden von in der Bibliothek enthaltenen Konzepten erleichtern. Abschnitt 3.2.4 untersucht die unterschiedlichen Formen der Vererbung näher auf ihre Tauglichkeit zur Erhöhung der Übersichtlichkeit.

2.2.4 Erweiterbarkeit

Erweiterbarkeit bezeichnet die Leichtigkeit, mit der Softwareprodukte an Spezifikationsänderungen angepaßt werden können [Meyer 88]. Software kann nur dann dauerhaft eingesetzt werden, wenn sie sich den verändernden Bedürfnissen der Benutzer anpassen läßt. Bei kleineren Programmen bereitet eine Änderung der Spezifikation in der Regel kaum Schwierigkeiten. Doch bei der „Programmierung im Großen“ [DeRemer, Kron 76] führen Veränderungen des Systems leicht zu erheblichen Problemen. Große Softwaresysteme neigen dazu, eine derart große Zahl von wechselseitigen Abhängigkeiten zwischen ihren Komponenten zu enthalten, daß eine Änderung in ihnen kaum mehr möglich ist.

Um die Erweiterbarkeit von Software zu erhöhen sollte daher der Entwurf des Systems möglichst einfach und dezentral gehalten und Abhängigkeiten zwischen Komponenten minimiert werden.

Erweiterbarkeit verlangt von der Software also eine Öffnung gegenüber Änderungen. Die Forderung nach Offenheit steht in einem scheinbaren Widerspruch zur Forderung nach Wiederverwendbarkeit. [Meyer 88] beschreibt dies unter dem Begriff des „Offen-geschlossen-Prinzips“ von Modulen. Module müssen nach [Meyer 88] sowohl offen als auch geschlossen sein können. Ein Modul wird als offen bezeichnet, wenn Erweiterungen an ihm vorgenommen werden können. Die Geschlossenheit von Modulen kennzeichnet den Zustand, in dem sich ein Benutzer des Moduls auf die Konstanz der Eigenschaften des Moduls verlassen kann.

Um Software erfolgreich wiederzuverwenden, ist die Anpassbarkeit an neue Aufgaben unbedingte Voraussetzung. Wiederverwendung und Erweiterbarkeit stehen also nur in einem scheinbaren Widerspruch zu einander. Tatsächlich ist die Erweiterbarkeit ein Unterstützung für die Wiederverwendbarkeit.

In späteren Abschnitten der Arbeit wird Vererbung als Möglichkeit der Kombination von Wiederverwendung und Erweiterbarkeit eine zentrale Rolle spielen.

Zusammenfassung

Die vorherigen Abschnitte zeigen, daß an die Qualität einer Softwarebibliothek für Massendaten besondere Anforderungen gestellt werden, die über jene an „normale“ Software hinausgeht. Da das Ziel einer Bibliothek die Wiederverwendung ihrer Dienste ist, besitzt die Wiederverwendbarkeit einer Bibliothek einen hohen Stellenwert. Der Anbieter einer Bibliothek sollte versuchen, Benutzer bei der Verwendung der Bibliothek zu unterstützen. Hierbei zeigt sich, daß das Ziel der Wiederverwendbarkeit mit den Zielen Verständlichkeit und Übersichtlichkeit eng zusammenhängt.

Zur Steigerung der Qualität von Bibliotheken bieten sich zunächst die von der jeweils verwendeten Programmiersprache angebotenen Hilfsmittel an. Insbesondere das Geheimnisprinzip und die Trennung von Definition und Implementierung sind hilfreich. Auch Konventionen über Benennung im Programmtext erleichtern das Verständnis der Bibliothek. Ein großer Teil der Dokumentation einer Bibliothek sollte auf diese Weise und in Form von Kommentaren bereits im Quelltext enthalten sein, da separate Dokumentation oft nicht aktuell ist.

In der Regel ist es erforderlich, zusätzlich Mechanismen außerhalb des Quelltextes der Bibliothek anzuwenden. Dokumentation ist eine Möglichkeit, dem Benutzer Bibliotheken zugänglich zu machen. Wegen der Fülle der in Bibliotheken angebotenen Diensten kommen oft auch Browser zum Einsatz.

Kapitel 3

Grundstrukturen für Softwarebibliotheken

Abhängig davon, ob die Softwarekomponenten einer Bibliothek Module oder Klassen sind, lassen sich zwei verschiedene Formen von Bibliotheken unterscheiden, die modul- bzw. objektorientierten Bibliotheken. Dieses Kapitel beschreibt sowohl die Eigenschaften als auch die sich daraus ergebenden Vor- und Nachteile modul- und objektorientierter Bibliotheken. Ziel des Vergleichs ist es, festzustellen, welche Form von Bibliotheken die in Kapitel 2 genannten Qualitätsmerkmale von Bibliotheken bei der Anwendung auf Bibliotheken für Massendatentypen besser erfüllt.

3.1 Modulorientierte Softwarebibliotheken

Eine mögliche Grundstruktur für Softwarebibliotheken ist die Aufteilung des Systems in funktionale Untereinheiten, auch „Module“ genannt [Schoett 81]. Den Prozeß der Gliederung und seine Ergebnisse bezeichnet man dabei als „Modularisierung“ [Maynard 72; Denert 79; Schoett 81]. Die nachfolgenden Abschnitte beschreiben zunächst die Eigenschaften und Ziele der Zerlegung von Software in Module und die sich aus dieser Vorgehensweise ergebenden Nachteile in Bezug auf Softwarebibliotheken. Am Beispiel der Bibliothek für Massendatentypen des Tycoon-Systems werden die auftretenden Probleme exemplarisch verdeutlicht.

3.1.1 Eigenschaften und Ziele

Modularisierung unterstützt die Verständlichkeit und Testbarkeit von Systemen, die Wiederverwendung einzelner Module und die arbeitsteilige Erstellung von Modulen durch verschiedene Programmierer [Hennicker 92]. Als Richtlinien der modulorientierten Zerlegung von Software faßt [Cohen 72] folgende Punkte zusammen:

- Jedes Modul bearbeitet eine logische Untermenge des gesamten Problems.
- Module sind in sich abgeschlossen. Ihre Größe soll angemessen sein[1].
- Module sind Unterprogramme.

[1] [Cohen 72] gibt als Richtwert an, daß ein Modul in der Funktionalität 200 Cobol–Programmzeilen entsprechen soll.

- ▷ Module besitzen Ein– und Ausgang.
- ▷ Das Verhalten von Modulen entspricht dem Gedanken der „Black–Box“, d. h. ihr Verhalten hängt nur von den Eingangswerten ab, nicht von einem zu erinnernden internen Zustand der Module.
- ▷ Alle Ein– und Ausgaben der Module sollten in Unterprogrammen gebündelt werden.

Wesentliche Vorteile der Modularisierung sind die Möglichkeit der getrennten Übersetzung der Module und die Erstellung der Module mit weniger Wissen über den Code in anderen Teilen des Gesamtsystems [Parnas 72a]. Die Aufteilung des Programms in kleine Module vereinfacht nicht nur die Implementierung, sondern erhöht auch die Testbarkeit des entstehenden Systems. Module sind im Sinne der Testbarkeit abgeschlossene Funktionseinheiten, die sämtliche Außenbeziehungen über ihre wohldefinierte Schnittstelle abwickeln. Statt ein monolithisches System mit unzähligen verschiedenen Zuständen auf Korrektheit prüfen zu müssen, läßt sich der Test nun in zwei Phase aufteilen [Stolze 73; Cohen 72]:

- ▷ Modultest: Im Rahmen des Modultests werden alle Module einzeln einem Test unterzogen. Alle in ihrer Schnittstelle angebotenen Methoden sind auf Korrektheit zu überprüfen.
- ▷ Gesamttest[2]: Nach erfolgreicher Durchführung des Modultests kann das aus den einzelnen Modulen zusammengesetzte Gesamtsystem auf Korrektheit getestet werden. Da die einzelnen Komponente des Systems bereits getestet sind, sollten in dieser Phase nur noch Fehler auftreten, die ihren Ursprung im Zusammenspiel der Module haben [Stolze 73].

Die Änderungsfreundlichkeit der Software steigt, wenn die Module nicht nur Verarbeitungsschritte, sondern wesentliche Entwurfsentscheidungen zusammenfassen[3] [Schoett 81; Parnas 72a]. Abhängigkeiten zwischen den Modulen sind zu minimieren, um Module unabhängig herstellen und verstehen zu können. Ein derartiges Design steigert die Modifizierbarkeit des gesamten Systems, da veränderte Anforderungen an das System nun in der Regel nicht mehr zu globalen Änderungen in zahlreichen Modulen führen.

Als Systemarchitektur schlägt [Cohen 72] vor (Abbildung 3.1), das Gesamtsystem durch eine Schichtenbildung aus aufeinander aufbauenden Modulen (im Bild A bis E) zu realisieren, so daß jede Schicht die Funktionalität der unter ihr liegenden bei der Realisierung eigener Funktionalität nutzt

Um die Unabhängigkeit zwischen den Modulen zu gewährleisten, schlägt [Parnas 71] das sog. Geheimnisprinzip (*information hiding*) vor. Der Entwickler eines Moduls soll danach die Weitergabe von Informationen über das Modul als bewußte Designentscheidung verstehen [Schoett 81]. Die Benutzer von Modulen neigen nach Ansicht von Parnas dazu, sich bei der Konstruktion eigener Programme mit Hilfe der angebotenen Module auf alle ihnen über die

[2] [Liggesmeyer 90] bezeichnet diesen Test als „Integrationstest“.

[3] [Cohen 72] kritisiert, daß in der Praxis trotz der Möglichkeiten der modularen Programmierung lediglich die ursprünglichen monolithischen Systeme in einzelne Module aufgeteilt würden, ohne sich über die zu dieser Aufteilung führenden Entscheidungen klar zu sein. Die Vorteile der modularen Programmierung gehen bei einer solchen Vorgehensweise verloren, da sich ändernde Benutzeranforderungen bei zu enger Koppelung der Module Änderungen in allen oder zumindest zahlreichen Modulen des Systems nach sich ziehen könnten.

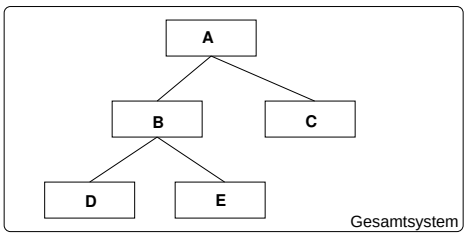

Abbildung 3.1: Entstehung des Gesamtsystems aus aufeinander aufbauenden Komponenten (nach [Cohen 72])

Funktionsweise der Module bekannten Informationen zu verlassen. Daher schlägt Parnas vor, nur so viele Informationen über die Module weiterzugeben, wie ein Benutzer wirklich benötigt (auch in: [Parnas 72b]). Diese veröffentlichten Informationen stellen damit als dauerhafte Eigenschaften der Module Schnittstelleninformationen dar. Veröffentlichte Eigenschaften dürfen vom Anbieter anschließend nicht mehr verändert werden, da durch diese Änderungen die Programme derjenigen Benutzer ungültig werden könnten, die sich auf die Dauerhaftigkeit verlassen haben. Die übrigen Eigenschaften können jedoch auch zu einem späteren Zeitpunkt noch verändert werden.

Bei einer wörtlichen Interpretation des vorgeschlagenen Geheimnisprinzips entstehen Probleme bei der Softwareentwicklung. Ein Mitglied des Teams dürfte sich beispielsweise nicht den von einem anderen Mitglied geschriebenen Quelltext eines Moduls ansehen, da ihm sonst auch alle Informationen über die interne Realisierung des Moduls bekannt würden, die der Anbieter des Moduls nicht zur Veröffentlichung freigegeben hat. Dies bedeutet, daß ein Programmierer auch nicht mehr beurteilen kann, welche von einem Modul angebotenen Prozeduren besonders effizient implementiert sind. Auch eine sinnvolle Auswahl zwischen verschiedenen Modulen mit gleichem Verwendungszweck ist bei dieser Interpretation des Geheimnisprinzips nicht mehr möglich [Schoett 81]. Eine explizite Spezifikation dauerhafter Eigenschaften, von denen die Korrektheit anderer Komponenten abhängen darf, vermeidet die bei einer wörtlichen Interpretation des Geheimnisprinzips auftretenden Nachteile [Schoett 81].

Als typische Entwurfsentscheidung, die in einem Modul zusammengefaßt werden sollte, nennt [Parnas 72a] eine Datenstruktur, ihre internen Verbindungen sowie die lesenden und schreibenden Zugriffsfunktionen. [Schoett 81] bezeichnet dies als ein „Datentyp-Modul“. Durch Anwendung des Spezifikationsprinzips auf Datentyp-Module gelangt man zur Entwurfsmethode der „Datenabstraktion“ oder der „Abstrakten Datentypen“ [Schoett 81]. Zugriffsfunktionen und ihr Verhalten in Form der Modulschnittstellen werden spezifiziert, nicht jedoch die interne Repräsentation der Daten. Benutzerprogramme, die nur auf Basis der Spezifikation der Module verifiziert wurden, können damit durch Änderungen der internen Repräsentation der Daten nicht ungültig werden.

Mit Hilfe der modularen Programmiersprachen kann man also abstrakte Datentypen realisieren. Eine Instanz eines solchen abstrakten Datentyps kann Daten speichern und bietet Funktionalität auf den Daten an. Ein Zugriff auf die gespeicherten Daten ist nur unter Verwendung der Schnittstelle des abstrakten Datentyps möglich. Aus diesem Grund kann man auch von einer horizontalen Trennung sprechen. Ein abstrakter Datentyp entspricht damit in der praktischen Programmkonstruktion einem Modularisierungskonzept [Schoett 81]. Die

Eigenschaften von Modulen fassen [Balzert 82; Antoniou, Sperschneider 88] zusammen:

- ▷ Bereitstellung einer funktionalen Abstraktion oder eines abstrakten Datentyps.
- ▷ Kontextunabhängigkeit bis auf eine definierte Schnittstelle, d. h. ein Modul ist unabhängig von seiner Umgebung entwickelbar, übersetzbar, prüfbar, wartbar und verständlich. [Goos 74] merkt hierzu an, daß die Schnittstelle übersichtlich und möglichst einfach sein soll. Das Auftreten komplexer Datenstrukturen in einer Schnittstelle deute auf eine mangelhafte modulare Zerlegung hin.
- ▷ Spezifizierung allein durch die Schnittstellenbeschreibung.
- ▷ Realisierung des Geheimnisprinzips
- ▷ Realisierung des Lokalitätsprinzips. Programmteile, die eine Funktion betreffen, sind an einer Stelle konzentriert.

Aus der Diskussion um die Zerlegung großer Programme in kleinere Module sind auch eigene Programmiersprachen hervorgegangen, die das Konzept der Modularisierung und der abstrakten Datentypen direkt unterstützen. Ihr wohl prominentester Vertreter ist Modula-2 [Wirth 88; Cin et al. 89].

Korrektheit, Robustheit, Wartbarkeit und damit Veränderbarkeit sind also die wesentlichen Ziele bei der Entwicklung des modularen Programmierstils. Kapselung von Daten und Aufteilung in einzelne Module helfen, diese Ziele zu erreichen und bilden die Grundlage der von modulorientierten Bibliotheken angestrebten Wiederverwendbarkeit.

3.1.2 Schwächen modulorientierter Bibliotheken

Überlegungen der Trennung, Aufteilung und Kapselung stehen beim modularen Stil im Vordergrund. Dieser Abschnitt greift diesen Gedanken auf und beschreibt, welche Formen der Trennung und Kapselung in modularer Programmierung vorkommen und welche Wirkung diese Spracheigenschaften und Mechanismen auf die Entwicklung und Gestaltung qualitativ hochwertiger modulorientierter Softwarebibliotheken haben. Die Qualität von Bibliotheken wird dabei insbesondere an den folgenden Qualitätsmerkmalen gemessen:

- ▷ Wiederverwendbarkeit und Spezialisierung
- ▷ Kombinierbarkeit
- ▷ Übersichtlichkeit durch Vereinheitlichung von Protokollen
- ▷ einfache Schnittstellen für komplizierte Funktionalität sowie Standardisierung

3.1.2.1 Einschränkung der Wiederverwendung und Spezialisierung

Verglichen mit der objektorientierten Programmierung treten Wiederverwendung und Spezialisierung bei modulorientierter Programmierung in anderer Form und mit anderer Bedeutung auf. Modulorientierte Programmierung wird häufig eingesetzt, um große Softwaresysteme zu realisieren. Der Systementwurf erfolgt dabei oft durch eine Zerlegung der Gesamtfunktionalität in einzelne Schritte (*top-down decomposition*) [Budgen 89]. Dieses Vorgehen ist auch als schrittweise Verfeinerung (*stepwise refinement*) bekannt [Wirth 71].

Bei der Anwendung dieser Entwurfstechnik stellt das zu entwerfende Gesamtsystem für den Entwerfer zunächst eine Funktion dar, die im Rahmen des Entwurfsprozesses spezifiziert werden muß. Im Anschluß an die Spezifizierung des Gesamtsystems teilt der Entwerfer die Funktionalität, die das Gesamtsystem erbringen soll, in Hauptfunktionsgruppen auf. Diese Dekomposition wiederholt sich mit den Hauptfunktionsgruppen, die zunächst in Funktionsgruppen und schließlich in Funktionen zerfallen, bis der Umfang der Funktionalität so überschaubar ist, daß die einzelne Funktion im Rahmen eines Moduls implementiert werden kann. Abbildung 3.2 zeigt einen solchen Entwurfsprozeß.

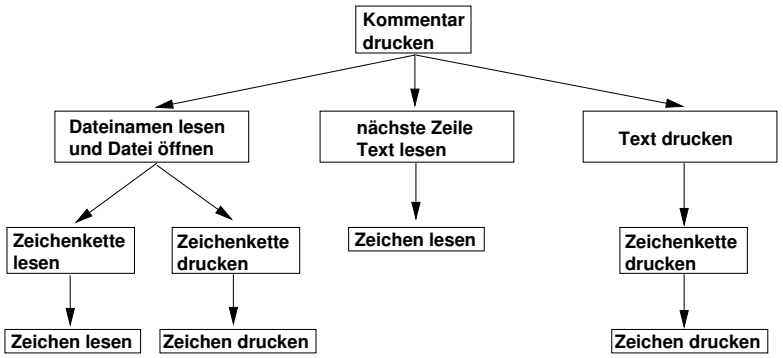

Abbildung 3.2: Erster Zerlegungsschritt (nach [Budgen 89])

In diesem ersten Schritt der schrittweisen Zerlegung kommt es noch zu doppelter Eintragung von gleicher Funktionalität. Die Abbildung zeigt, daß beispielsweise die Funktionalität, ein Zeichen oder eine Zeichenkette zu lesen bzw. zu schreiben, mehrfach enthalten ist. Dies zeigt, daß diese Funktionalität an mehreren Stellen innerhalb des Systems von Nutzen ist. Ein zweiter Schritt des Entwurfs reduziert doppelt auftretende Funktionalität auf eine wiederverwendbare Funktionalität. Abbildung 3.3 zeigt das Ergebnis des zweiten Entwurfsschrittes.

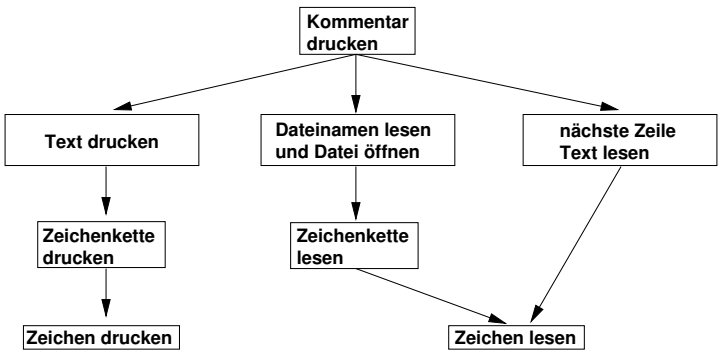

Abbildung 3.3: Zweiter Zerlegungsschritt (nach [Budgen 89])

In großen Systemen setzt sich der beschriebene Algorithmus zur Aufteilung der Funktionalität in handhabbare Einheiten fort, indem aus den einzelnen Komponenten Schichten

gebildet werden. Jede Schicht nutzt in einem solchen Modell die Funktionalität der unter ihr liegenden Schicht und bietet zusätzliche Funktionalität an.

Die bisher beschriebenen Aufteilungen betreffen nur die Funktionalität des Systems, nicht die Daten, auf denen die Funktionalität arbeitet. Die gesamte Funktionalität, deren Aufteilung in den beiden Abbildungen dargestellt wird, arbeitet auf derselben Repräsentation der Daten. Das Format dieser Daten wird nicht verändert. In sehr großen Systemen ist dies nicht immer der Fall. In vielen Fällen ist die Veränderung der Repräsentation von Daten sogar Teil der gewünschten Funktionalität. Ein Datenbanksystem ist ein Beispiel für eine solche Veränderung der Repräsentation. Bei Übergängen von einer Systemschicht des Datenbanksystems in die nächste werden die Daten in eine andere Repräsentation konvertiert. Auf diese Weise erfolgt eine Abbildung von Relationen auf einfache Daten, die auf der Magnetplatte gespeichert sind. Der Wechsel der Repräsentation ist in diesen Fällen keine Ineffizienz, sondern Teil der gewünschten Funktionalität des Systems. Wiederverwendung zeigt sich also in modulorientierten Systemen in der Regel dadurch, daß Funktionalität in kleinere Einheiten zerlegt wird und die Gesamtfunktionalität unter Verwendung der einzelnen Komponenten realisiert ist.

Diese Form der Wiederverwendung korrespondiert mit Spezialisierung, da auf einer funktionalen Ebene des Entwurfsvorganges die Hauptfunktionalität in eine kleinere Granularität aufgespalten wird. Ziel dieses Vorgangs ist jedoch eine Menge von Funktionseinheiten, deren Funktion jeweils disjunkt ist. Eine Übereinstimmung der Funktionalität, wie sie zwischen mehreren Klassen in einem objektorientierten System durch die Ableitung von derselben Oberklasse zu finden ist, gibt es in einem modulorientierten System in der Regel nicht.

Spezialisierung findet also bei modulorientierter Programmierung auf der Ebene der Funktionalität im Rahmen einer disjunkten Zerlegung während des Entwurfsvorgangs statt, während sie sich in objektorientierten Bibliotheken auf die Eigenschaften von Objekten bezieht. Modulorientierte Zerlegung führt damit zu disjunkten Komponenten, objektorientierte Zerlegung zu ähnlichen Komponenten.

3.1.2.2 Mangelnde Unterstützung inkrementeller Erweiterung

Es gibt zwei Alternativen, ein Modul inkrementell zu erweitern:

- Änderungen direkt im ursprünglichen Quelltext durchzuführen:
  Nach der Änderung existiert die alte Version des Quelltextes nicht mehr. Alle Programme, welche die alte Version des Moduls benutzen, müssen angepaßt und neu übersetzt werden. Diese Form der Erweiterung bietet sich damit an, wenn etwa eine neue Prozedur in die Schnittstelle einzufügen ist. Die Veränderung bestehender Prozeduren gefährdet jedoch die Gültigkeit von Klienten.

- Änderungen auf der Kopie des Quelltextes:
  Anschließend existieren zwei Versionen des Moduls, die zum größten Teil identisch sind. Wenn sich zu einem späteren Zeitpunkt Teile des identischen Codes als fehlerhaft oder nicht optimal erweisen, ist die Fehlerkorrektur oder Optimierung der Algorithmen in beiden Modulen parallel durchzuführen. Eine derartige Überarbeitung von gleichartigem Code ist teurer als die Änderung eines einzelnen Moduls und sorgt für eine neue Fehlerquelle.

Beide Methoden sind somit mit nicht unerheblichen Nachteilen behaftet.

3.1.2.3 Einschränkungen der Kombinierbarkeit

Abschnitt 3.1.2.1 beschrieb die Aufteilung der Funktionalität des Gesamtsystems als typisches Vorgehen modulorientierter Programmierung. Das Gesamtsystem ist damit eine Kombination der Funktionalität der einzelnen Komponenten. Diese Beobachtung legt die Vermutung nahe, modulorientierte Sprachen würden die Kombination von Modulen unterstützen. Diese Vermutung ist jedoch unzutreffend. Der Versuch, in einer modularen Sprache Funktionalität aus verschiedenen Modulen zu kombinieren, stößt schnell auf Widerstände. Bei der Entwicklung einer Bibliothek zur Verwaltung von Massendatentypen sind diese Widerstände besonders ausgeprägt.

Funktionalität ist in der Informatik in der Regel die Bereitstellung von Aktionsmöglichkeiten auf Daten. Die von einem Modul angebotene Funktionalität muß bereits vollständig realisiert sein. Eine vollständige Realisierung setzt aber voraus, daß einem Modul bekannt ist, in welcher Repräsentation die Daten vorliegen, auf denen die Funktionalität des Moduls arbeiten soll[4]. Die Funktionalität verschiedener Module arbeitet also auf einem jeweiligen Datentyp. Ausgehend vom Gedanken der Kapselung kann diese Funktionalität nur über die Schnittstelle des jeweils zugehörigen abstrakten Datentyps verwendet werden.

Abbildung 3.4 verdeutlicht dieses Problem. Die Abbildung zeigt zwei Hierarchien von abstrakten Datentypen, die aus jeweils 3 Schichten bestehen. Die Repräsentation der Daten erfolgt in unterschiedlicher Weise in den jeweils niedrigsten Schichten der beiden Hierarchien. Auf dieser Schicht baut jeweils ein abstrakter Datentyp (A und B) auf. Die Funktionalität von A und B wird von jeweils einem weiteren abstrakten Datentyp (C und D) erweitert, ohne die Repräsentation der Daten zu verändern. Eine Kombination der Funktionalität der beiden abstrakten Datentypen C und D scheitert nun jedoch an deren Prinzip der Kapselung. Es ist also unmöglich, die Funktionalität von C und D effizient auf dieselben Daten anzuwenden. Eine solche Kombination wäre nur möglich, indem die Daten des einen abstrakten Datentyps in die Repräsentation des anderen übertragen würden. Ein solches Umkopieren ist jedoch in der Regel so ineffizient, daß eine Kombination von Funktionalität verschiedener abstrakter Datentypen unterbleibt. Der Versuch scheitert an der in der Darstellung angedeuteten „Mauer der Ineffizienz".

Bei der Realisierung einer Bibliothek zur Verwaltung von Massendatentypen tritt diese Einschränkung der Kombinierbarkeit besonders drastisch in Erscheinung. Sowohl durch Zeiger verbundene Speicherungsstrukturen als auch die Speicherung von Elementen in Feldern sind in jedem Fall Bestandteil einer solchen Bibliothek. Für jede dieser Repräsentationsformen muß in der Bibliothek eine eigene vollständige und unabhängige Hierarchie von abstrakten Datentypen aufgebaut werden[5].

3.1.2.4 Fehlende Unterstützung der Standardisierung

In der Bibliothek einer modulorientierten Sprache finden sich zahlreiche Implementierungen von abstrakten Datentypen. Dabei fällt auf, daß ein großer Teil der Prozeduren und Funktionen in den Schnittstellen mehrerer abstrakter Datentypen enthalten ist (z.B. Einfügen, Löschen, usw.).

[4] Abschnitt 3.2 zeigt, daß objektorientierte Bibliotheken Funktionalität auch ohne Festlegung auf eine Repräsentation der Daten anbieten können und dadurch die Kombinierbarkeit von Funktionalität unterstützen.

[5] [Nord 92] kommt in seiner Dissertation zu ähnlichen Ergebnissen:„Consider the important problem of combining, or „integrating", modules that must share data. The possibility of sharing means that interacting modules must agree not only on the abstract interfaces, but also on the underlying data representations."

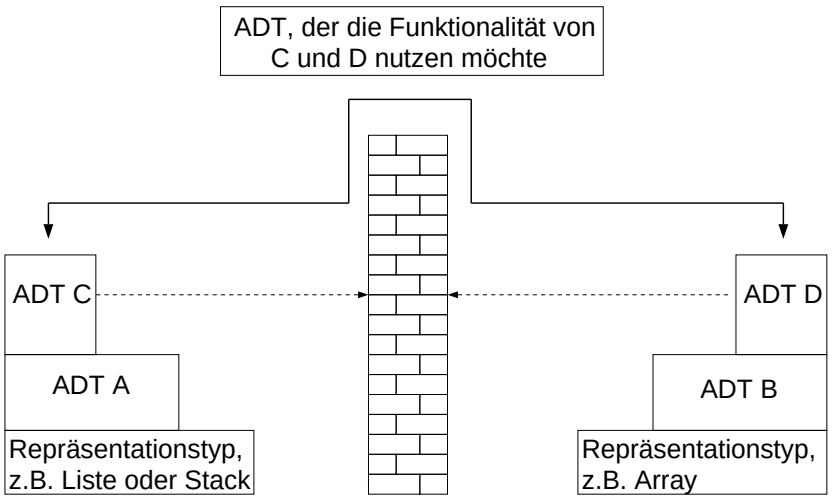

Abbildung 3.4: Ineffizienz der Kombination von Funktionalität

Wegen der getrennten Entwicklung der Modulhierarchien abhängig von der jeweiligen Repräsentation der Daten ist es dem Entwickler allein überlassen, eine Einheitlichkeit der Notationen über die Grenze einer einzigen Schnittstelle hinweg zu erreichen. Dieses Problem verschärft sich noch, wenn die Bibliothek in Gruppenarbeit entsteht. Die Einheitlichkeit basiert dabei nur auf Vereinbarungen zwischen den verschiedenen Entwicklern und auf selbst gesetzten Normen und Regeln.

3.1.2.5 Fehlende einfache Sicht auf komplexe Schnittstellen

Ein weiteres Problem modulorientierter Softwarebibliotheken besteht darin, daß der Benutzer alle Operationen auf seinen Daten einer einzigen Schnittstelle entnehmen muß. Wenn der Benutzer also in seinem Programm an einer Stelle eine komplexe Operation verlangt, muß er einen abstrakten Datentyp auswählen, dessen Schnittstelle komplex genug ist, diese Operation anzubieten. Auch bei den sehr viel einfacheren Operationen für das Einfügen oder Löschen von Elementen ist der Benutzer an die Verwendung der komplizierten und durch die Zahl der Methoden möglicherweise unübersichtlichen Schnittstelle gebunden. Modulorientierte Bibliotheken bieten dem Benutzer also nicht die Möglichkeit, die Funktionalität der komplexen Operationen aus der Schnittstelle eines spezialisierten abstrakten Datentyps zu entnehmen, aber für einfache Standardanfragen einen einfachere Schnittstelle zu benutzen. Spätere Teile der Arbeit zeigen, daß objektorientierte Sprachen bei der Lösung dieses Problems unterstützen.

3.1.3 Die Massendatentypen-Bibliothek von Tycoon

Die Bibliothek des Tycoon-Systems zur Verwaltung von Massendatentypen ist in der modulorientierten Programmiersprache TL entwickelt worden. Dieser Abschnitt zeigt beispielhaft anhand dieser Bibliothek, welche Auswirkungen die Eigenschaften modulorientierter Sprachen auf die Struktur von Softwarebibliotheken und den Programmierstil bei deren Erstellung haben.

3.1.3.1 Struktur der Bibliothek

Abbildung 3.5 zeigt die Benutztbeziehungen innerhalb der Bibliothek. Als benutzt gelten dabei die Module, die von einem Modul importiert werden. Die Abbildung enthält nur jene Module, die im Tycoon-System den Massendatentypen zugerechnet werden. Andere Importbeziehungen, z.B. Importe des Moduls Iter, sind nicht eingezeichnet. Es fällt auf, daß die Bibliothek in 5 nicht zusammenhängende Teile aufgeteilt ist:

- Teil 1: Die Bibliothek implementiert eine Schlange (Queue), ohne einen anderen abstrakten Datentyp der Bibliothek zu verwenden. Dies ist um so erstaunlicher, als die Bibliothek auch eine Liste (List) enthält, die bei der Realisierung der Schlange hilfreich sein könnte.
- Teil 2: Den zweiten Teil der Bibliothek bilden Assoziation (Assoc), Wörterbuch (Dictionary) und Menge (Set), die unter Verwendung der Liste realisiert wurden. Die Liste übernimmt die Speicherung der Daten, während die anderen abstrakten Datentypen ihre Operationen durch Aufrufe der Listenschnittstelle anbieten.
- Teil 3: Diesen Teil der Bibliothek bilden die abstrakten Datentypen, deren Elemente in einem Feld (Array) gespeichert sind. Eine Ausnahme stellt lediglich Graph dar, dessen Elemente in einer VarList gespeichert sind und der den Array-basierten Stack als Hilfsstruktur verwendet.
- Der vierte Teil der Bibliothek besteht aus dem B-Baum (BTree) und seinen Komponenten. Im Gegensatz zu den ersten drei Teilen der Bibliothek bildet in diesem Teil nur der B-Baum selbst einen abstrakten Datentyp, während die anderen Module durch eine Zerlegung der Funktionalität des B-Baumes entstanden sind und nicht selbständig verwendet werden können.
- Im letzten Teil der Bibliothek befindet sich das mehrdimensionale Wörterbuch (DDimDictionary). Wie beim B-Baum wurde auch hier die Gesamtfunktionalität in einzelne Module aufgeteilt. DDimDictionary ist die Schnittstelle dieses abstrakten Datentyps zur Mehrbereichssuche. Die Daten des Wörterbuches werden in einer speziell zu diesem Zweck entworfenen Liste DDimDicPartList gespeichert, die auch einzeln als abstrakter Datentyp verwendbar wäre. Die übrigen Module sind wie beim B-Baum nicht selbständig verwendbar.

3.1.3.2 Vorgehensweise bei der Implementierung von Bibliotheken in Tycoon

Innerhalb der Bibliothek lassen sich drei Programmierstile in Bezug auf die Realisierung der jeweiligen abstrakten Datentypen erkennen:

1. Ein abstrakter Datentyp, die Schlange (Queue), ist ohne Verwendung eines anderen abstrakten Datentyps entstanden. Diese völlige Neuentwicklung der gesamten Funktionalität läßt eine mögliche Wiederverwendung beispielsweise der Liste völlig außer acht. Die Schlange wird auch selbst innerhalb der Bibliothek nicht verwendet.
2. Der größte Teil der Bibliothek (Teil 2 und 3) besteht aus Paaren von abstrakten Datentypen, bei denen einer die Speicherung der Elemente übernimmt, während ein zweiter eine andere Schnittstelle für den Zugriff auf diese Daten unter Verwendung der Schnittstelle

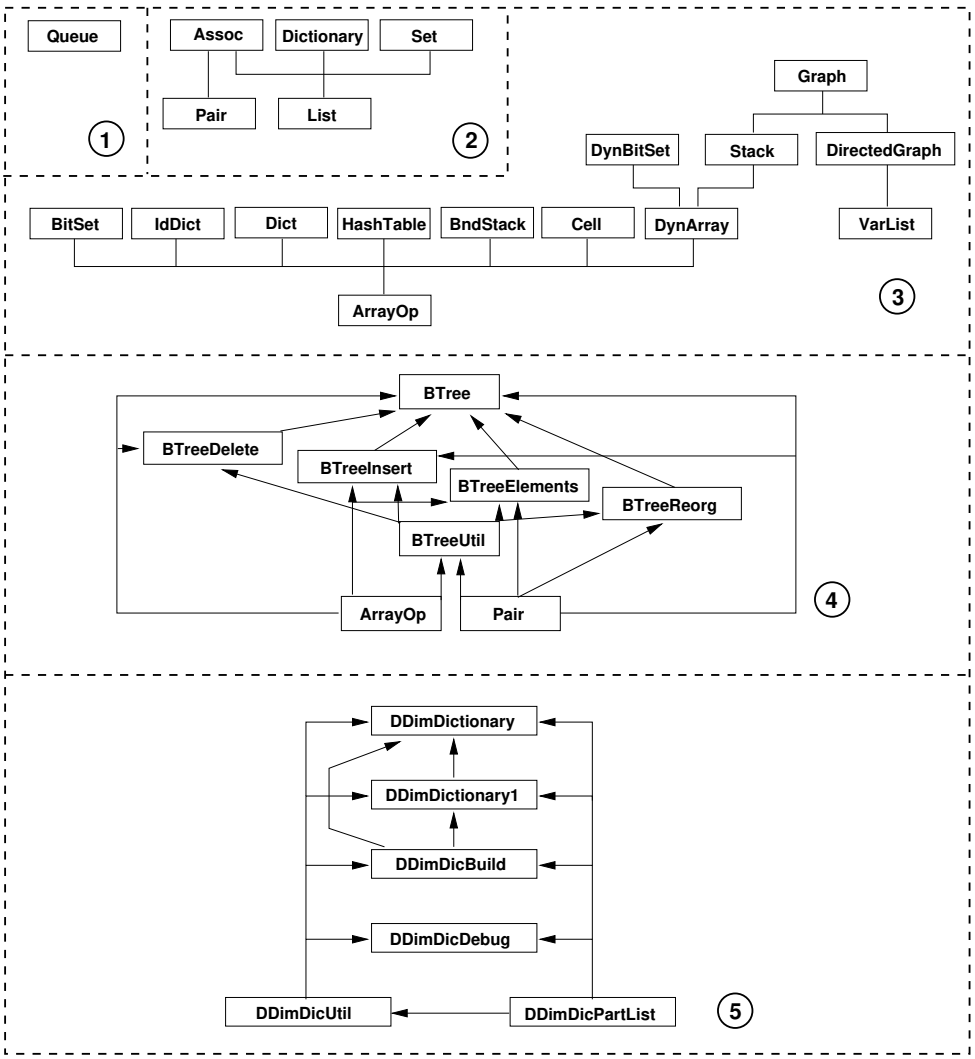

Abbildung 3.5: Bibliothek für Massendaten des Tycoon-Systems

des ersten abstrakten Datentyps realisiert. Diese Vorgehensweise entspricht der Darstellung in Abbildung 3.4. Allerdings zeigt sich in der Bibliothek des Tycoon-Systems, daß die Schichtung über dem Repräsentationstyp zur Speicherung der Elemente in der Regel auf einen weiteren abstrakten Datentyp beschränkt ist. Ausnahmen von dieser Regel sind nur die abstrakten Datentypen, die unter Verwendung des dynamischen Feldes (DynArray), welches auf einem Feld basiert, realisiert wurden.

3. Wiederum eine andere Art der Realisierung zeigt sich beim B-Baum und dem mehrdimensionalen Wörterbuch (Teil 4 und 5 in Abbildung 3.5). Die Vorgehensweise bei diesen beiden Teilen der Bibliothek entspricht einer Kombination aus der Aufteilung in Komponenten und der Speicherung der Daten in einem Repräsentationstyp analog zu der unter Punkt 2 beschriebenen Vorgehensweise.

Die Realisierung von abstrakten Datentypen dieser hohen Komplexität erfolgt vermutlich in zwei Schritten. Zunächst wird ein geeigneter abstrakter Datentyp zur Speicherung der Elemente des Ziel-Datentyps gewählt (B-Baum) oder neu erstellt (mehrdimensionales Wörterbuch). In einem weiteren Schritt erfolgt eine Aufteilung der umfangreichen und komplexen Funktionalität des Ziel-Datentyps in mehrere einzelne Komponenten. Nach der Realisierung der einzelnen Komponenten in jeweils einem Modul kann der Ziel-Datentyp unter Wiederverwendung der Komponenten aus diesen zusammengesetzt werden[6].

Typisch für diese Vorgehensweise ist das Entstehen von zahlreichen nicht selbständig verwendbaren Modulen. Da diese Module nicht als ungeeignet zum selbständigen Einsatz gekennzeichnet sind, kann ein im Verzeichnis nach einem passenden abstrakten Datentyp suchender Benutzer der Bibliothek durch diese Dateien verwirrt werden.

Die beschriebenen Beobachtungen in der Bibliothek für Massendaten lassen auf ein typisches Vorgehen der Softwareentwicklung in einer modulorientierten Bibliothek schließen. Ein Programmierer, der einen neuen abstrakten Datentyp realisieren möchte, sucht zunächst nach einem geeigneten und möglichst bereits implementierten abstrakten Datentyp zur Datenrepräsentation. Nach einem Vergleich der zur Verfügung stehenden abstrakten Datentypen und ihrer angebotenen Operationen erfolgt die Festlegung auf genau einen solchen Datentyp. Die Daten des neuen abstrakten Datentyps werden im bereits implementierten gespeichert. Der neue abstrakte Datentyp wird anschließend durch Operationen auf dem alten realisiert.

3.1.4 Bewertung modulorientierter Bibliotheken

Modulorientierte Programmierung scheint besonders geeignet zu sein, um Schichtenarchitekturen zu entwerfen und realisieren, bei denen an den Übergängen von einer Schicht zur nächsten ein Wechsel der Repräsentation der Daten erfolgt. Da dieser Wechsel gewünscht ist, führt die Kapselung der Daten nicht zu einer Ineffizienz des Systems.

Doch trotz der erheblichen Vorteile gegenüber einer monolithischen Softwareentwicklung ist Modulorientierung wegen der damit verbundenen Nachteile nicht für jede Anwendungsform das geeignete Paradigma. Softwarebibliotheken sollten u.a. übersichtlich und erweiterbar, ihre Funktionalität kombinierbar sein. Die Eigenschaften modularer Sprachen stehen diesen Zielen entgegen und verhindern ihre Realisierung in modulorientierten Bibliotheken durch den vorherrschenden Gedanken der Trennung und Kapselung und die Einschränkung der Wiederverwendung auf eine Benutztbeziehung. Es ist nicht möglich, auch Designentscheidungen oder Eigenschaften erneut zu verwenden, wie etwa beim Erben in objektorientierten Sprachen [Klint 86]. Zusätzlich fehlt modulorientierter Programmierung eine Unterstützung für die Kombination von Funktionalität.

Inkrementelle Erweiterung ist problematisch und der Grad der Wiederverwendung innerhalb der Bibliothek bleibt wegen der Aufteilung in disjunkte, stark spezialisierte Komponenten gering, so daß mehrfach ähnlich implementierte Funktionalität die Wartbarkeit reduziert. Fehlende Beziehungen zwischen ähnlichen abstrakten Datentypen bedeuten, daß Funktionalität unter verschiedenen Namen angeboten werden kann. Modulorientierten Bibliotheken fehlt

[6] In Bezug auf das mehrdimensionale Wörterbuch, das von den Autoren dieser Arbeit realisiert wurde [Lotter, Römer 94], handelt es sich bei diesem Absatz um einen Erfahrungsbericht, beim B-Baum [Shen 94] um eine Vermutung.

somit eine Unterstützung der Standardisierung. Der Benutzer einer modulorientierten Bibliothek wird bei der Auswahl des von ihm zu verwendenden abstrakten Datentyps nicht von der Struktur der Bibliothek unterstützt. Die abstrakten Datentypen der Bibliothek stehen nebeneinander und die Struktur der Bibliothek gibt keine Hinweise auf zu erwartende Eigenschaften. Weiterhin fehlt eine einfache Sicht auf komplizierte Schnittstellen und es sind nicht alle Module selbständig nutzbar. Modulorientierte Bibliotheken erfüllen somit die in Kapitel 2 geforderten Anforderungen nicht vollständig.

3.2 Objektorientierte Softwarebibliotheken

Dieses Kapitel beschreibt die grundlegenden Verfahren und Ziele der objektorientierten Programmierung. Ein besonderes Augenmerk wird auf Verfahren liegen, die für die Entwicklung von Behälterklassen verwendet werden.

3.2.1 Terminologie

Mit dem Aufkommen des Begriffs „objektorientiert“ entstand gleichzeitig eine neue Terminologie. Die Notwendigkeit der Einführung dieser neuen Terminologie ist nicht unumstritten[7]. Dennoch scheint es heute nahezu unmöglich, über das Thema Objektorientierung zu schreiben, ohne auf die weithin verbreiteten Begriffe zurückzugreifen. Aus diesem Grund erläutern die folgenden Abschnitten die in der objektorientierten Programmierung gebräuchlichen Begriffe.

3.2.1.1 Objekte, Eigenschaften und Methoden

Ein Objekt ist eine nach außen abgeschlossene Einheit aus Daten und Funktionalität. Die Daten des Objekts werden in lokalen, zum Objekt gehörenden Variablen gespeichert. Der Inhalt dieser Variablen bestimmt den Zustand des Objekts. Die lokalen Variablen werden nach außen verborgen und sind nur durch die Ausführung von Unterprogrammen veränderbar. Die auf die Daten eines Objekts anwendbaren Unterprogramme werden Methoden genannt.

Ein Objekt führt eine seiner Methoden aus, wenn es von einem anderen Objekt durch das Senden einer entsprechenden Nachricht hierzu aufgefordert wird. Voraussetzung ist, daß die in der Nachricht verlangte Methode zur Schnittstelle des die Nachricht empfangenden Objekts gehört [Otto 91][8]. Eine vereinfachende Terminologie spricht davon, daß eine Methode an ein Objekt gesendet wird. Dies ist jedoch kein wesentlicher Unterschied, entscheidend ist, daß Objekte von anderen zu Aktivitäten aufgefordert werden. Objekte stehen damit in einer „Benutztbeziehung“ zueinander, da sie Dienste erbringen.

Viele objektorientierte Programmiersprachen, z. B.Eiffel, C++ und die im Rahmen dieser Arbeit benutzte Sprache TooL, bieten eine weitergehende Verfeinerung der Kontrolle des Zugriffs auf Variablen und Methoden an, indem sie eine Unterscheidung in private und öffentliche Methoden bzw. Variablen erlauben. Private Methoden können dabei im Gegensatz zu

[7] [Reiser, Wirth 94]: „Mit dem Ziel, ein neues Paradigma für das Programmieren im Großen einzuführen, folgte eine Umbenennung vieler altbekannter Konzepte; so wurden aus Variablen Instanzen, aus Prozeduren Methoden und aus Prozeduraufrufen das Senden von Meldungen.“

[8] Ähnliche Darstellungen der Thematik finden sich auch in [Wirfs-Brock et al. 90; Coleman et al. 94; Kilberth et al. 94; Decker, Hirshfield 95; Coad, Yourdon 94; Coad, Yourdon 91; Beaudouin-Lafon 94; Cook 91; Booch 94].

den öffentlichen nicht von anderen Objekten aufgerufen werden, sondern nur vom Objekt selbst. Öffentliche Variablen können im Unterschied zu den bisher beschriebenen Variablen auch von anderen Objekten ohne den Aufruf einer Methode des Objekts direkt verändert werden.

Durch die Kapselung von Daten in privaten Variablen und den Zugriff über öffentliche Methoden, der *Schnittstelle* des Objekts, erhält ein Objekt die Mächtigkeit eines abstrakten Datentyps. Abbildung 3.6 verdeutlicht die im Zusammenhang mit dem Objektbegriff auftretende Terminologie.

Eine Unterstützung der Abstraktion bedeuten abstrakte Methoden (*deferred method*). Eine Abstrakte Methode ist ein Methodenrumpf, der die Signatur der Methode festlegt, aber noch keine Implementierung für die Funktionalität der Methode angibt.

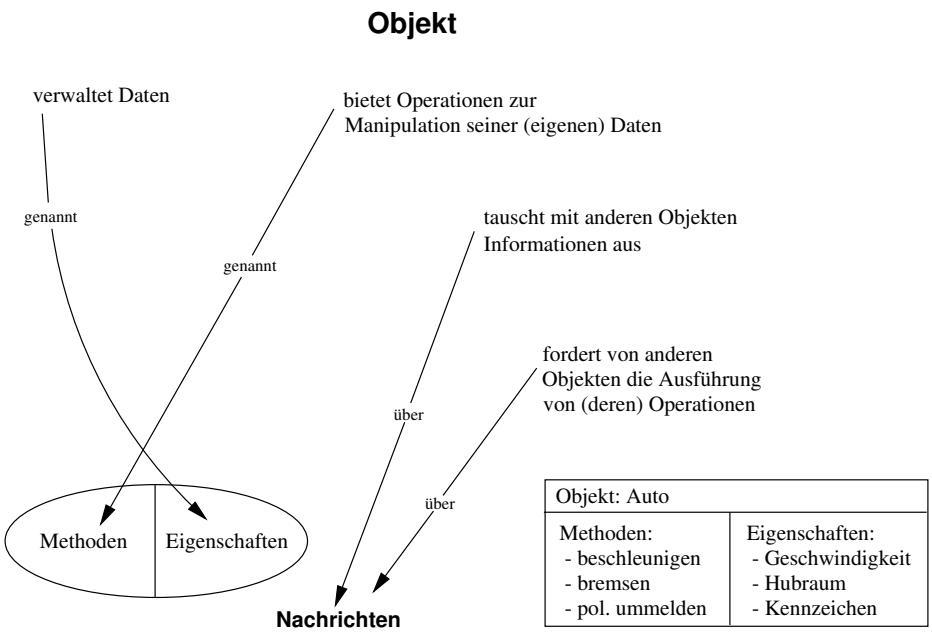

Abbildung 3.6: Begriffsklärung: Objekt (nach [Otto 91])

3.2.1.2 Klassen

Eine Klasse kann als Schablone zur Erzeugung von Objekten verstanden werden. Eine Klasse erzeugt dabei gleichartige Objekte, d. h. die Objekte einer Klasse besitzen dieselben Methoden und Variablen. Lediglich die in den Variablen gespeicherten Eigenschaften unterscheiden sich von Objekt zu Objekt. In Systemen, in denen Klassen ebenfalls Objekte sind, benötigen Klassen klassenspezifische Eigenschaften und Methoden, um Objekte erzeugen zu können. Die Klasse speichert die zu ihren Objekten gehörenden Methoden. Auf alle Objekte einer Klasse sind deshalb dieselben Methoden anwendbar. Objekte werden als Exemplar ihrer Klasse bezeichnet.

Klassen, die abstrakte Methoden enthalten oder erben und nicht implementieren werden als abstrakte Klasse bezeichnet (*abstract class*). Von abstrakten Klassen existieren keine Ex-

emplare, sie dienen der Unterstützung der Abstraktion. Die Ausdrucksmächtigkeit der in den abstrakten Klassen zu definierenden Bedingungen ist in den objektorientierten Sprachen unterschiedlich. In allen objektorientierten Sprachen können in den abstrakten Klassen abstrakte Methoden definiert werden. Durch die Definition einer Methode werden alle Subklassen gezwungen, diese Methode zu implementieren, wenn die Subklasse nicht selbst wieder abstrakt sein soll. Die Definition einer abstrakten Methode in einer Klasse stellt also die Bedingung dar, daß diese Methode zur Schnittstelle aller unter der abstrakten Klasse hängende Subklassen gehören muß.

Einige objektorientierte Sprache, z.B. Eiffel, besitzen die Möglichkeit, zu den Methoden auch Vor- und Nachbedingungen zu definieren. Diese Bedingungen werden wie die übrigen Eigenschaften einer Klasse auch vererbt. Durch die Definition von Vor- oder Nachbedingung in den abstrakten Methoden einer abstrakten Klasse kann also nicht nur festgelegt werden, daß die entsprechende Methode zur Schnittstelle aller Subklassen gehören muß, sondern der Entwickler der Bibliothek kann auch die Semantik der in den Subklassen zu einem späteren Zeitpunkt zu implementierenden Methoden spezifizieren.

Abbildung 3.7 verdeutlicht den Klassenbegriff der objektorientierten Programmierung. Die Abbildung zeigt zwei Exemplare der Klasse PKW. Beide Exemplare speichern zu den in der Klasse mit Namen angegebenen Variablen die jeweils zugehörigen Werte. Auf beide Exemplare sind die bei der Klasse abgelegten Methoden („beschleunigen“, „bremsen“ und „polizeilich ummelden“) anwendbar.

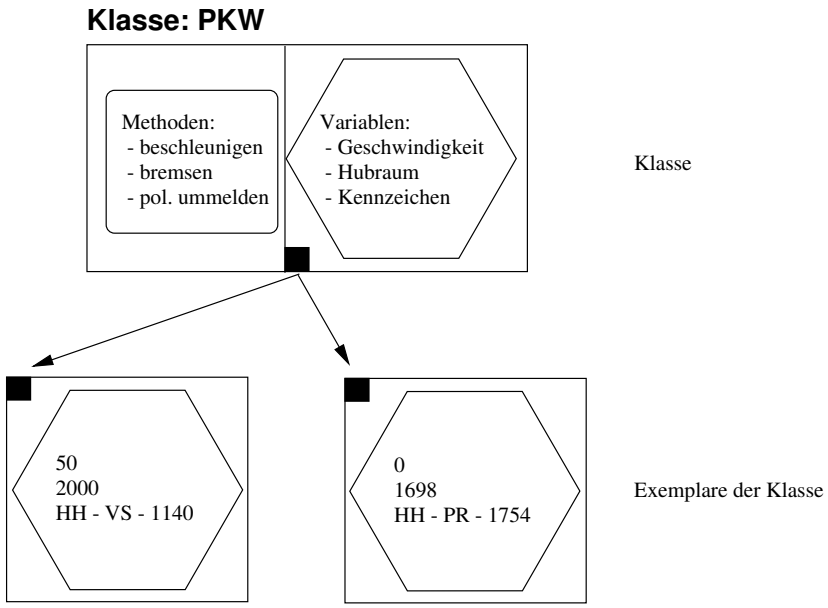

Abbildung 3.7: Objekte derselben Klasse (nach [Otto 91])

3.2.1.3 Vererbung und Klassenhierarchie

Vererbung bezeichnet den Vorgang der Ableitung einer Klasse von einer bereits bestehenden Klasse. Einige objektorientierte Programmiersprachen, z.B. Eiffel, C++ und TooL, bieten

Mehrfachvererbung an, die es erlaubt, eine Klasse von mehreren bereits bestehenden Klassen abzuleiten. Wenn eine Klasse von einer anderen erbt, bezeichnet man die beerbte Klasse als Oberklasse (*super class*) der erbenden Klasse. Die erbende Klasse ist Subklasse ihrer Oberklasse. Abbildung 3.8 zeigt ein Beispiel für eine Hierarchie von Klassen.

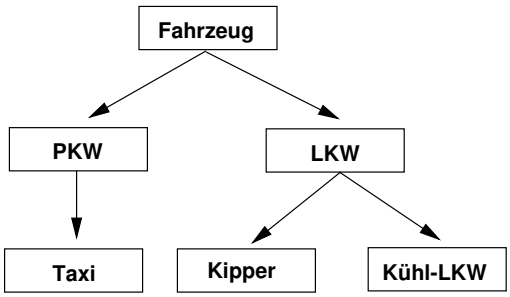

Abbildung 3.8: Beispiel einer Klassenhierarchie (nach [Otto 91])

Erhält ein Objekt eine Nachricht, so wird zunächst festgestellt, ob die zu dieser Nachricht gehörende Methode in der Klasse des Objekts selbst definiert worden ist. Wenn dies der Fall ist, kann die Klasse des Objekts die Methode selbst ausführen. Ist dies nicht der Fall, so handelt es sich um eine geerbte Methode[9]. Die entsprechende Methode der Oberklasse kann jedoch auf den Variablen eines Objekts der Subklasse arbeiten. Da die geerbten Methoden der Oberklassen auch für Objekte der Subklasse anwendbar sind, kann man sie als Teil der Schnittstelle der Objekte der Subklasse ansehen. Für das Objekt und seinen Benutzer bedeutet es keinen Unterschied, ob die Methode in der Klasse selbst definiert oder aus einer Oberklasse geerbt wird. Vererbung ist damit ein Mechanismus zur Erweiterung von Klassen. Exemplare der erbenden Klasse erhalten durch den Vorgang des Erbens alle Eigenschaften der Oberklasse, wodurch die Anwendung aller Methoden der Oberklasse auf die Variablen der Subklasse möglich ist. In einem weiteren Schritt ist es möglich, die geerbten Methoden für die Anwendung auf Exemplare der erbenden Klasse zu verändern oder der Schnittstelle neue, auf die Exemplare der Oberklasse nicht anwendbare Methoden hinzuzufügen.

Abbildung 3.9 zeigt das Senden einer Nachricht an das Objekt einer erbenden Klasse. Ein Exemplar (das Taxi mit dem Kennzeichen HH-BL-1007) der Klasse Taxi empfängt die Nachricht, die Geschwindigkeit um 20 km/h zu reduzieren. Da die Methode „bremse“ für alle Personenkraftwagen aufrufbar sein soll, wurde sie bereits in der Klasse PKW definiert. Auf das Exemplar der Klasse Taxi wird somit eine geerbte Methode aus der Klasse PKW angewendet.

3.2.2 Typisierung und Polymorphismus

Da Bibliotheken als wesentliches Ziel die Wiederverwendbarkeit ihrer Komponenten verfolgen, ist die Allgemeinheit der angebotenen Dienste von großer Bedeutung. Der Wert einer Bibliothek steigt beispielsweise auch dadurch, daß ihre Dienste auf eine Vielzahl von Argumenten anwendbar sind. Dieser Abschnitt beschreibt daher die Begriffe Typisierung und

[9]Wenn die Methode auch in den Oberklassen nicht definiert ist, gibt es einen Laufzeitfehler. Der größte Teil der Übersetzer für objektorientierte Sprachen erkennt solche Fehler jedoch bereits bei der Übersetzung der Programme.

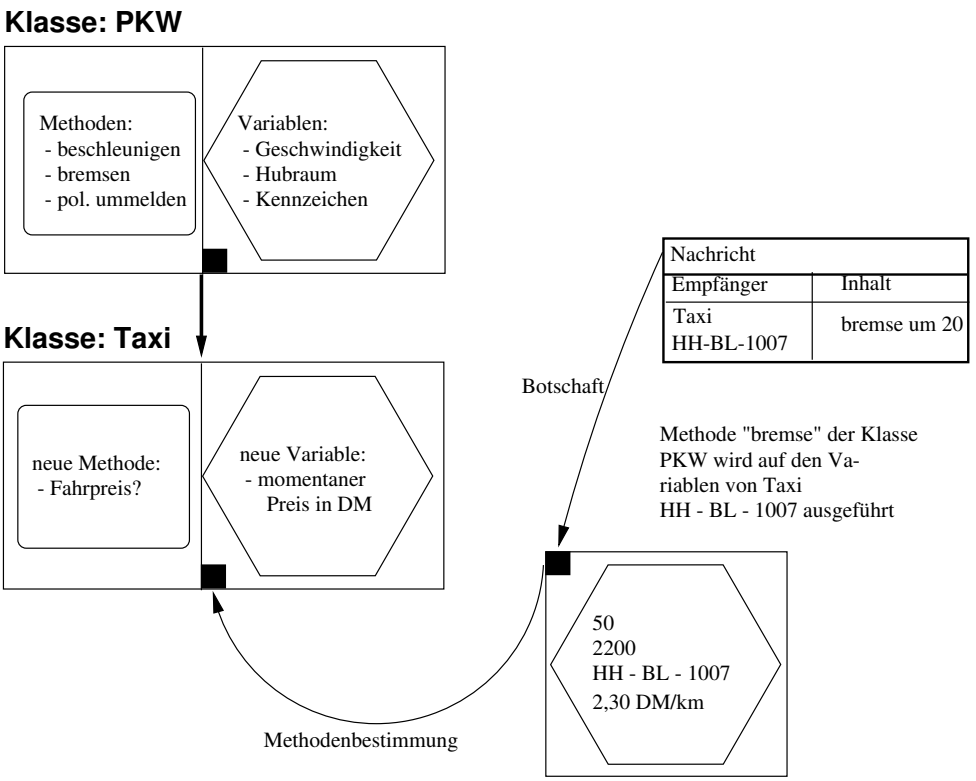

Abbildung 3.9: Vererbung (nach [Otto 91])

Polymorphismus. Durch die Anwendung dieser Techniken kann der Wert einer Bibliothek für Massendaten gesteigert werden.

3.2.2.1 Typisierung

Nach [Cardelli, Wegner 85] sind Typen Mengen von Werten, wobei die Werte ein „gleichartiges Verhalten“ zeigen. Bedingt durch den Begriff der „Gleichartigkeit“ sind die Wertmengen und damit die Typen nicht notwendig disjunkt. Werte können also mehreren Mengen angehören und damit auch mehrere Typen besitzen. Auf gleichartige Werte sind dieselben Operationen anwendbar. Typen fassen somit Werte zusammen, auf die dieselben Operationen anwendbar sind.

Ziel der Zuordnung von Werten und Typen ist die Möglichkeit einer Verträglichkeitsüberprüfung von Werten und Operationen, welche die Ausführung nicht vorgesehener Operationen auf der internen Repräsentation der Werte verhindert. Solche Operationen würden die interne Repräsentation in einen inkonsistenten Zustand versetzen. Um eine Überprüfung der Zulässigkeit von Operationen auf Werten durchführen zu können, müssen zunächst den Werten des Programms Typen zugeordnet werden, was durch Deklaration oder Typinferenz möglich ist. Programmiersprachen mit dieser Zuordnung werden als typisiert bezeichnet. Die Typisierung einer Sprache erhöht somit die Sicherheit von Software.

Wenn ein Typtest bereits zur Übersetzungszeit möglich ist, spricht man von einer stati-

schen Typisierung (*static typing*), während Typüberprüfungen zur Laufzeit als dynamische Typisierung (*dynamic typing*) bezeichnet werden. Strenge Typisierung (*strong typing*) bezeichnet eine Typüberprüfung, bei der ausgeschlossen ist, daß ein Programm mit einem Typfehler zur Laufzeit abbricht.

3.2.2.2 Polymorphismus

Konventionell typisierte Sprachen wie Pascal oder Modula-2 ordnen jedem Wert genau einen Typ zu. Im Gegensatz dazu können Werte in polymorphen Sprachen mehreren Typen angehören. Es gibt verschiedene Arten von Polymorphismus in Programmiersprachen. Bild 3.10 zeigt die verschiedenen Formen von Polymorphismus.

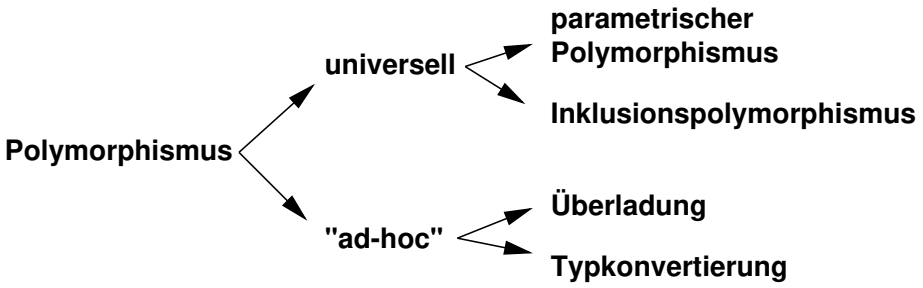

Abbildung 3.10: Formen von Polymorphismus (nach [Cardelli, Wegner 85])

Polymorphismus läßt sich zunächst in zwei Kategorien einteilen, den universellen und den „ad-hoc"-Polymorphismus. Diese beiden grundsätzlichen Formen unterscheiden sich durch die Zahl von verschiedenen Typen, auf die eine mit dem jeweiligen Polymorphismus behaftete Funktion anwendbar ist. Universell polymorphe Funktionen sind auf eine unendliche Zahl von verschiedenen Typen anwendbar, wobei die verschiedenen Typen eine gemeinsame Grundstruktur besitzen. Bei ad-hoc-Polymorphismus hingegen ist die Anzahl der verschiedenen zulässigen Typen begrenzt. Ad-hoc-Polymorphismus kann daher auch als eine endliche Menge monomorpher Funktionen aufgefaßt werden.

Eine universell polymorphe Funktion wendet somit auf alle Werte unabhängig von ihrem jeweiligen Typ dieselben Operationen an. Ad-hoc-Polymorphismus hingegen bedeutet die Ausführung verschiedener Operationen auf Werte abhängig vom jeweiligen Typ der Werte.

Universeller Polymorphismus kann in einem zweiten Schritt in parametrischen und Inklusionspolymorphismus aufgeteilt werden. Parametrischer Polymorphismus wird durch einen Typparameter realisiert. Durch den Typparameter erhält die entsprechende Funktion nicht nur den Wert zur Laufzeit als Parameter, sondern auch den Typ des jeweiligen Parameters. Funktionen mit parametrischem Polymorphismus heißen auch generische Funktionen (*generic function*). Die Generizität der Funktionen setzt allerdings voraus, daß alle Werte gleichförmig repräsentiert bzw. behandelt werden, z.B. durch Zeiger.

Inklusionspolymorphismus bezieht sich auf Objekte, die als zu mehr als einem Typ gehörend betrachtet werden können. Die Bezeichnung als Inklusion beruht auf dem Umstand, daß diese Typen nicht disjunkt sein müssen. Elemente vom Subtyp gehören ebenfalls zum Supertyp. Durch diese Eigenschaft können Elemente vom Subtyp im Kontext des Super-

typs benutzt werden. Dies bedeutet, daß alle Operationen des Supertyps auch auf Subtypen anwendbar sind.

Inklusionspolymorphismus spielt eine bedeutende Rolle bei der Entwicklung objektorientierter Softwarebibliotheken und bietet Vorteile sowohl für den Benutzer der Bibliothek als auch für deren Anbieter. Nachfolgende Abschnitte beschreiben die sich aus der Anwendung von Inklusionspolymorphismus ergebenden Vorteile.

Ad-hoc-Polymorphismus läßt sich ebenfalls in zwei Arten aufteilen. Diese Arten sind das Überladen (*overloading*) von Funktionsnamen und die explizite Typkonvertierung (*coercion*) von Werten. Bei der Überladung von Funktionsnamen wird derselbe Funktionsname verwendet, um eine Operation auf verschiedenen Typen von Werten auszuführen, wobei die durchzuführende Operation vom jeweils aktuellen Typ des Wertes abhängt. Ein Beispiel für eine überladene Operation kann die Addition sein. Die Operation + kann dabei sowohl ganzzahlige Argumente wie auch Fließkommazahlen addieren.

Eine Typkonvertierung wandelt den Typ eines Wertes in einen anderen Typ um. Das genannte Beispiel einer überladenen Funktion zur Addition zweier Zahlen läßt sich auch mit Hilfe der Typkonvertierung erklären. Unter der Annahme, es existiere nur eine Funktion zur Addition von Zahlen, müßte vor der Addition von ganzzahligen Argumenten und Fließkommazahlen die ganzzahlige Zahl in eine Fließkommazahl umgewandelt werden.

Polymorphismus erweitert die Flexibilität einer Sprache gegenüber monomorphen Sprachen dahingehend, daß Objekte sich in verschiedenen Situationen unterschiedlich verhalten können. Die von einem Objekt angebotene Funktionalität wird durch Polymorphismus somit kontextabhängig. In Bezug auf objektorientierte Bibliotheken zur Verwaltung von Massendatentypen entstehen aus dieser Kontextabhängigkeit einige Vorteile für Benutzer und Anbieter der Bibliothek.

3.2.3 Vorteile des objektorientierten Ansatzes für Bibliotheken

Die Einführung des Paradigmas objektorientierter Programmierung und Analyse sowie objektorientierten Designs bringt im Vergleich mit dem modulorientierten Ansatz Vorteile für die Erstellung qualitativ hochwertiger Software, die in den folgenden Abschnitten im Hinblick auf die Gestaltung und Entwicklung von Softwarebibliotheken für Massendatentypen untersucht werden.

3.2.3.1 Orientierung an den Daten statt an Funktionen

Die Objektorientierte Programmierung versteht die Programmausführung als eine Simulation des Verhaltens von realen oder imaginären Teilen der Welt. Objektorientierte Programmierung bedeutet die Suche nach geeigneten Abstraktionen für die Objekte der Anwendungsumgebung. Während bei der modulorientierten Programmierung eine funktionale Zerlegung und anschließende Schichtung vorherrscht, erfolgt die Abstraktion bei objektorientierter Programmierung über den Daten [Meyer 93]. Die Abstraktionen werden in Klassen zusammengefaßt. Abbildung 3.11 zeigt die unterschiedlichen Vorgehensweisen.

Ein wesentlicher Vorteil dieses Ansatzes ist es, Sichtweisen aus der Anwendungsumgebung in die Softwareerstellung übernehmen zu können. Das objektorientierte Paradigma ist daher durchgängig anwendbar auf Analyse, Design und Implementierung und bietet damit eine einheitliche Sicht auf die verschiedenen Phasen der Systementwicklung. Diese Vereinheitlichung läßt Programme verständlicher werden und erhöht Wartbarkeit und Erweiterbarkeit.

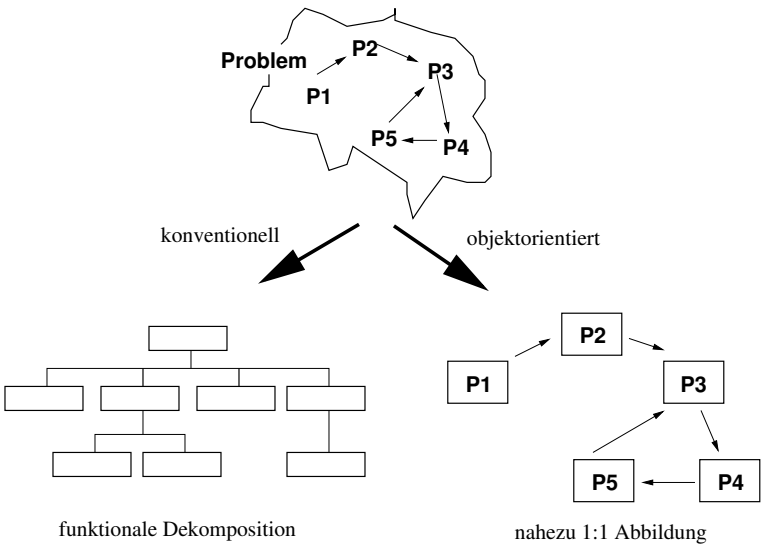

Abbildung 3.11: Konventionelle und objektorientierte Zerlegung eines Problems (nach [Otto 91])

Die Fokussierung des Paradigmas der Objektorientierung auf die Abstraktion über den (Daten-) Objekten der realen Welt bewirkt eine leichtere Anpaßbarkeit der Software an Veränderungen der Anforderungen von Benutzern. Die bei modulorientierter Programmierung gebildeten Abstraktionen über der Funktionalität des Systems werden mit großer Wahrscheinlichkeit bei Veränderungen der Benutzeranforderungen ebenfalls zu verändern sein, da neue Anforderungen der Benutzer sich in der Regel auf eine Veränderung der Funktionalität beziehen. Die an der Problemlösung beteiligten Daten und mit ihnen auch die Abstraktionen objektorientierter Programmierung bleiben jedoch oft über längere Zeit unverändert. Es ist daher in der objektorientierten Programmierung leichter möglich, das System durch eine Erweiterung der ursprünglichen Funktionalität auf denselben Datenabstraktionen an neue Anforderungen anzupassen.

3.2.3.2 Vorteile durch Vererbung

Neben die Benutztbeziehung tritt in objektorientierten Bibliotheken eine Erbbeziehung, woraus eine Verbesserung der Möglichkeiten zur Wiederverwendung resultiert. Vererbung bringt Vorteile in Bezug auf die Kombinierbarkeit von Funktionalität, die inkrementelle Erweiterung von Funktionalität und Repräsentation, die Wiederverwendung von Konzepten, sowie die Klassifizierung. Diese Vorteile werden im folgenden näher erläutert.

Kombinierbarkeit von Funktionalität

Eine Kombination von Funktionalität kann effizient erfolgen, wenn die Funktionalität noch nicht auf eine Repräsentationsform der Daten festgelegt ist. Eine derartige Entkoppelung ermöglichen objektorientierte Sprachen durch das Konzept der abstrakten Klasse, deren abstrakte Methoden bereits für andere Methoden benutzbar sind. Üblich ist daher, die Funk-

tionalität, die von der später erfolgenden Wahl der Repräsentationsform abhängig ist, als abstrakte Methode anzugeben, so daß darauf aufbauend der größte Teil der Funktionalität bereits in der abstrakten Klasse implementiert werden kann. Mehrfacherben ermöglicht die Kombination von Funktionalität mehrerer abstrakter Klassen, wobei eine anschließende Instantiierbarkeit der so entstehenden Klasse die Implementierung aller abstrakten Methoden verlangt. Mit der Implementierung aller Methoden ist in der Regel eine Festlegung auf eine Repräsentationsform der Daten nötig.

Für die Realisierung einer Bibliothek zur Verwaltung von Massendaten bedeutet dies, daß ein großer Teil der Funktionalität losgelöst von der Frage der Repräsentationsform der Daten in den weiter oben liegenden abstrakten Klassen der Klassenhierarchie implementiert werden kann und kombinierbar bleibt. Erst instantiierbare Klassen für Endbenutzer der Bibliothek müssen die Repräsentationsform festgelegt und die verbleibenden abstrakten Methoden realisieren.

Inkrementelle Erweiterbarkeit

Um die Funktionalität einer Subklasse gegenüber ihrer Oberklasse auszuweiten, ist es ausreichend, ihrer Schnittstelle Methoden hinzuzufügen. Weiterhin kann in der Subklasse die Verhaltensweise geerbter Methoden gegenüber ihrer Verhaltensweise in der Oberklasse verändert werden. Auch die inkrementelle Erweiterung der Repräsentation gegenüber der Oberklasse ist in der Subklasse möglich. Hierbei ist es ausreichend, die Veränderungen ausschließliche in der Subklasse durchzuführen. Da die ursprüngliche Klasse unverändert bestehen bleibt, ist auch die Funktionsfähigkeit von anderen Klassen, welche die ursprüngliche Klasse ebenfalls beerben oder sie benutzen, durch Änderungen in der abgeleiteten Klasse nicht gefährdet. Hinzu kommt, daß durch die Vererbung kein doppelter Code entsteht.

Die Verhaltensweise geerbter Methoden kann durch Überschreiben verändert werden. Hierbei ist ist die Möglichkeit des Aufrufs von Methoden der Oberklasse hilfreich, die auch besteht, wenn die gleichnamige Methode in der Subklasse verändert wurde. Da sich die Funktionalität beider Methoden oft nur in Nuancen unterscheidet, ist die ursprüngliche Methode in vielen Fällen bei der Implementierung der neuen Methode von Nutzen. Dadurch erhöht sich auch innerhalb der Bibliothek der Grad der Wiederverwendung. Objektorientierung vermeidet somit die in modulorientierten Bibliotheken auftretenden Probleme bei der inkrementellen Erweiterung.

Abstraktion und Wiederverwendung von Konzepten

Durch die Entkoppelung von Funktionalität und Repräsentation der Daten ist es in objektorientierten Bibliotheken möglich, in einer Klassenhierarchie so etwas wie abstrakte Konzepte zu implementieren, ohne eine konkrete Speicherungsform der Daten zu berücksichtigen. Die Kombination derartiger „Konzeptklassen“ durch Mehrfachvererbung erzeugt instantiierbare Klassen für den Benutzer. Kombinierbarkeit und Wiederverwendung von Konzepten stehen somit in einem engen Zusammenhang und erlauben Wiederverwendung auf einem höheren Abstraktionsniveau, als es in modulorientierten Bibliotheken der Fall ist [Knudsen et al. 93].

Klassifizierung der Funktionalität

Vererbung erleichtert nicht nur dem Anbieter die Erstellung änderungsfreundlicher Bibliotheken, sondern bringt auch dem Benutzer wichtige Vorteile. Von der Wurzel des Vererbungs-

graphen aus betrachtet, nimmt die Komplexität der Klassen mit jedem Vererbungsschritt zu. Diese Eigenschaft hilft dem Benutzer bei der Auswahl der für sein Problem zu verwendenden Klasse. Dabei kann sich der Benutzer von den Anforderungen an die Klasse leiten lassen.

Der Benutzer kann eine Klasse mit bestimmten Eigenschaften finden, indem er dem Vererbungsgraphen von der Wurzel aus folgt und dabei in den Pfad absteigt, dessen Klassen die gewünschten Eigenschaften haben. Vererbung, inkrementelle Verfeinerung und Spezialisierung der Eigenschaften der Klassen unterstützen den Benutzer also bei der Auswahl und Suche nach geeigneten Klassen als Leistungserbringer für die Lösung seiner Aufgaben.

Einfache Sicht auf komplexe Objekte

Eine objektorientierte Bibliothek bietet ihren Benutzern Schnittstellen verschiedener Komplexität an. Mit zunehmender Entfernung einer Klasse von der Wurzel des Vererbungsgraphen steigt der Grad der Mächtigkeit und Spezialisierung der Schnittstelle der Klasse und damit auch ihre Komplexität. Da die komplexeren Klassen durch Vererbung aus den allgemeineren hervorgehen, können die Objekte der Subklasse wegen des Inklusionspolymorphismus auch im Kontext der Oberklasse auftreten. Der Benutzer kann somit auf die Objekte der spezielleren Klassen auch die Methoden der allgemeineren Klassen anwenden. Im Gegensatz zu modulorientierten Bibliotheken ist daher bei einer Vielzahl von Operationen ein Ausweichen auf eine einfachere Schnittstelle möglich.

Gemeinsame Sicht auf verschiedene Klassen

Durch den mit der Vererbungsbeziehung der Klassen der Bibliothek einhergehenden Inklusionspolymorphismus gewinnt der Benutzer jedoch nicht nur eine einfachere, sondern auch eine gemeinsame Sicht auf Objekte verschiedener Klassen. Durch dieses einheitliche, auf verschiedene Klassen anwendbare Protokoll vereinfacht sich der Umgang mit Objekten verschiedener komplexer Klassen. Nur wenn eine Methode nötig ist, die in der gemeinsamen Sicht noch nicht vorhanden ist, muß die komplexe Sicht des jeweiligen Objekts selbst verwendet werden, in allen anderen Fällen ist jedoch die Verwendung der Oberklassen-Schnittstelle ausreichend.

3.2.3.3 Kapselung

Auch in der objektorientierten Programmierung werden Daten gekapselt. Der Zugriff auf die Daten eines Objekts ist nur über die Schnittstelle des Objekts möglich. Eine Ausnahme von dieser Regel ist jedoch, daß auch die Exemplare einer erbenden Klasse auf die von der Oberklasse definierten Variablen zugreifen dürfen, ohne die Methoden einer Schnittstelle verwenden zu müssen. Diese Einschränkung der Kapselung ist jedoch dann unschädlich, wenn die geerbten Variablen der Oberklasse als durch die Vererbung auch zur erbenden Klasse gehörend betrachtet werden. Diese Betrachtungsweise scheint sachgerecht, da die erbende Klasse von ihrer Oberklasse abgeleitet worden ist. Zwar müssen Subklassen die in den Oberklassen bereits definierten Variablen nicht neu aufführen oder definieren, doch diese Variablen gehören durch die Vererbung auch zu den erbenden Klassen und sind somit zu behandeln wie dort eingeführte Variablen. Es scheint daher angemessen, auch in der objektorientierten Programmierung von einer Kapselung der Daten zu sprechen. Probleme, die sich aus der Durchbrechung der Kapselung ergeben können, beschreibt der nächste Abschnitt.

3.2.4 Formen der Vererbung

Bei der Entwicklung von Behälterklassen muß zwischen der Sicht der Bibliotheksentwickler und der Sicht der Benutzer unterschieden werden. Besonders bei der Verwendung der Vererbung können bei unvorsichtiger Anwendung Nachteile für Benutzer entstehen. In der Literatur [Wetzel 94] werden im wesentlichen zwei Formen der Vererbung unterschieden. Dies sind *Implementierungsvererbung* und *Konzeptvererbung*. [Traub 95] nennt *Typvererbung* als dritte Form der Vererbung. Die folgenden Abschnitte erläutern die verschiedenen Formen der Vererbung. Dabei ist zu klären, welche Motivation hinter der jeweiligen Form steht und welche Auswirkungen sie auf die Qualität einer Bibliothek haben.

3.2.4.1 Implementierungsvererbung

Die Konzepte des abstrakten Datentyps und der Vererbung sind teilweise konträr. Bei der Vererbung werden die beim abstrakten Datentyp verborgenen Details für den Erben sichtbar. Somit setzt Vererbung eine Verletzbarkeit des Kapselungsprinzips voraus.

Die Durchbrechung der Kapselung ermöglicht der erbenden Klasse die Redefinition geerbter Methoden und die Implementierung neuer Methoden unter Verwendung geerbter Repräsentationen. Die Verletzung der strengen Kapselung birgt jedoch eine nicht zu unterschätzende Gefahr in sich. Wenn in der Oberklasse an der vererbten internen Repräsentation Veränderungen vorgenommen werden, müssen auch die Implementierungen der erbenden Klassen erneut auf Korrektheit untersucht werden, da sie durch die Veränderung ungültig geworden sein könnten [Wetzel 94].

Erbende Klassen können jedoch nur dann ungültig werden, wenn sie geerbte interne Repräsentationen nutzen. Wenn die erbende Klasse auf die internen Repräsentationen nur über ebenfalls geerbte Methoden der Oberklasse zugreift, bleibt sie korrekt, falls die Semantik der Methoden nicht verändert wird. Benutzende Wiederverwendung ist also von Veränderungen auf Seiten des Diensterbringers weniger betroffen als erbende Wiederverwendung [Snyder 86].

Eine Möglichkeit der Vermeidung direkten Zugriffs auf geerbte interne Repräsentationen liegt in der Einführung von Methoden in der Oberklasse, deren Aufruf in der Subklasse den direkten Zugriff ersetzt. Da diese Methoden für Benutzer der Klasse unerheblich sind, brauchen diese Methoden auch nur den erbenden Klassen zur Verfügung gestellt zu werden. Derartige private Methoden werden vorgeschlagen [Snyder 86; Wetzel 94], um die Idee des abstrakten Datentyps auch in der objektorientierten Programmierung zu verwenden. In [Snyder 86] wird dabei sogar von einer objektorientierten Programmiersprache verlangt, den Zugriff auf geerbte Variablen einer Klasse nur durch von der Oberklasse zur Verfügung zu stellende Methoden zu erlauben.

In [Traub 95] werden die Probleme, die beim Einsatz von Vererbung mit dem Ziel der Codewiederverwendung entstehen, am Beispiel einer Liste und einer sortierten Liste beschrieben. Problematisch ist die Ableitung der sortierten Liste mittels Implementierungsvererbung von der Liste. Zwar ist auch die sortierte Liste eine Liste, so daß es grundsätzlich möglich ist, die Eigenschaften der Liste zu erben und die Sortierung hinzuzufügen, wobei beispielsweise die Elemente einfügende Methode zu überschreiben wäre. Diese Methode muß in der Subklasse SortierteListe die Semantik dahingehend ändern, daß das neue Element an der der Sortierung entsprechenden Stelle in die Liste eingefügt wird.

Es werden allerdings in einer allgemeinen Liste weitere Methoden zum Einfügen von Elementen vorhanden sein. Dies könnte zum Beispiel eine Methode sein, die ein Element als

erstes Element in die Liste einfügt. Die Semantik einer solchen Methode ist aber in der sortierten Liste nicht mehr zulässig, da sie in einem direkten Widerspruch zu deren Eigenschaft der „Sortiertheit“ steht.

Wenn die sortierte Liste dennoch von der allgemeinen Liste abgeleitet werden soll, so wären die geerbten und nun nicht mehr zulässigem Methoden so zu überschreiben, daß sie zur Laufzeit eine Fehlermeldung ausgeben. Es ist offenkundig, daß dieses Vorgehen einen Widerspruch zum Ziel der Entwicklung sicherer Software darstellt. Die Zulässigkeit von Methodenaufrufen darf nicht erst zur Laufzeit überprüft werden.

Eine Alternative wäre es, die nicht mehr in das Konzept der durch Implementierungsvererbung gewonnenen Subklasse passenden geerbten Methoden aus der Schnittstelle der Subklasse zu entfernen. Einige objektorientierte Sprachen bieten zu diesem Zweck spezielle Mechanismen an, mit denen der Aufruf geerbter Methoden unterbunden werden kann. Doch diese scheinbare Alternative führt zu einem Verlust an Übersichtlichkeit. Es ist dann für einen Benutzer nicht mehr ohne weiteres feststellbar, welche Methoden in einer Klasse überhaupt zur Verfügung stehen.

Zwischen Klassen eine Vererbungsbeziehung mit dem Ziel der Wiederverwendung von Code einzuführen, liegt im Ermessen des Bibliotheksentwicklers. Die Vererbungsbeziehung sollte besser verborgen bleiben, um eine leichtere Revidierbarkeit der Entscheidung zu gewährleisten [Wetzel 94].

3.2.4.2 Konzeptvererbung

Alternativ zur Implementierungsvererbung kann Vererbung jedoch auch als ein Mechanismus zur Verfeinerung von Konzepten aufgefaßt werden, was als Konzeptvererbung bezeichnet wird [Snyder 86]. Da mit der Konzeptspezialisierung eine Klassifikation von Eigenschaften der Klassen einhergeht, die dem Benutzer die Verwendung der Bibliothek erleichtern soll, ist die Sichtbarkeit der Konzeptvererbung in der Schnittstelle der beteiligten Klasse erwünscht.

Wenn in der Schnittstelle einer Klasse nicht nur Methoden, sondern auch Variablen enthalten sind, werden natürlich auch diese vererbt. Deshalb ist in einem solchen Fall keine klare Trennung zwischen Implementierungsvererbung und Konzeptvererbung mehr möglich. Beide Formen der Vererbung treten dann gemischt auf [Wetzel 94].

Ist jeder Klasse ein Typ zugeordnet, so geht Konzeptvererbung mit einer Subtypbeziehung des Verhaltens der korrespondierenden Typen einher. Auf diese Weise wird die Subtypisierung völlig unabhängig von der Wiederverwendung der Implementierung. Wegen des Kapselungsprinzips kann eine durch Konzeptvererbung abgeleitete Klasse nach außen ein in Subtypbeziehung zum Verhalten ihrer Oberklasse stehendes Verhalten anbieten, obwohl die Implementierung von der Oberklasse völlig verschieden sein kann [Wetzel 94].

In [Snyder 86] findet sich als Beispiel die durch Konzeptvererbung hergestellte Subtypbeziehung zwischen dem Verhalten einer Multimenge (Bag) und dem einer Menge (Set), wobei die Implementierung beider Klassen kaum Übereinstimmungen aufweist.

In [Traub 95] wird Konzeptvererbung am Beispiel der Liste und der sortierten Liste erläutert. Beiden Listen legen die in ihnen enthaltenen Elemente geordnet ab. Bei der Liste ist die Ordnung extern, bei der sortierten Liste intern , also durch den Algorithmus der einfügenden Methode bestimmt. Nur bei der nicht sortierten Liste kann der Benutzer die Reihenfolge der Elemente in der Liste durch die Reihenfolge des Einfügens der Elemente selbst bestimmen. Eine die Eigenschaften klassifizierende Vererbungsbeziehung läßt sich daher herstellen, indem von der Klasse mit geordneten Elementen je eine Klasse mit intern bzw. extern

geordneten Elementen als Vorgänger für die Liste und die sortierte Liste abgeleitet wird.

3.2.4.3 Typvererbung

In [Traub 95] wird noch eine dritte Form der Vererbung eingeführt und als Typvererbung bezeichnet. Bei dieser Form der Vererbung wird wegen der Unverträglichkeit einer Erbbeziehung zwischen Liste und sortierter Liste auf eine Erbbeziehung verzichtet. Statt dessen wird eine gemeinsame Oberklasse gebildet, welche die gemeinsamen Eigenschaften beider Listen enthält. Von diesem Typ einer abstrakten Liste erben dann die Typen Liste und sortierte Liste. Abbildung 3.12 aus [Traub 95] zeigt die verschiedenen Formen der Vererbung angewendet auf Listen.

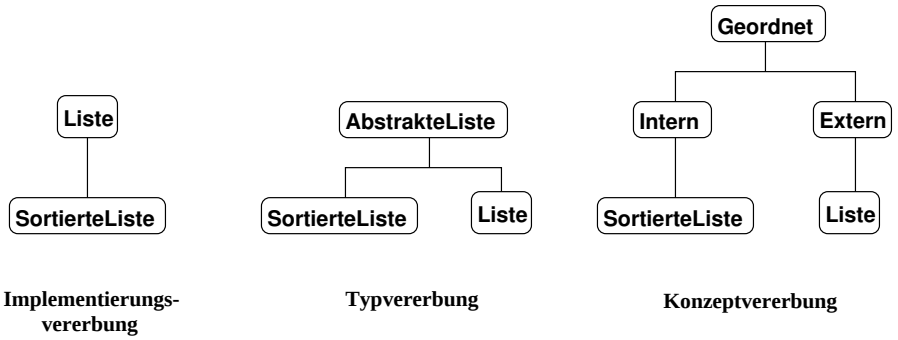

Abbildung 3.12: Drei Formen der Vererbung

Als Fazit bleibt festzustellen, daß aufgrund der in diesem Abschnitt beschriebenen Nachteile Vererbung nicht mit dem alleinigen Ziel der Implementierungs-Wiederverwendung benutzt werden sollte.

3.2.5 Ergebnis des Vergleichs der Grundstrukturen

Durch Vererbung bieten objektorientierte Bibliotheken eine Verbesserung von Wiederverwendung gegenüber modulorientierten Bibliotheken bei gleichzeitiger Betonung von Kombinierbarkeit und Erweiterbarkeit. Diese Eigenschaften führen sowohl für den Anbieter als auch für den Benutzer einer Bibliothek zu Vorteilen. Hervorzuheben sind die einfache Sicht auf verschiedene komplexe Objekte, Vereinheitlichung von Protokollen, Unterstützung inkrementeller Erweiterungen sowie Unterstützung der Standardisierung von Benennungen. Zusätzlich bedeutet Objektorientierung die Orientierung an den gegenüber der Funktionalität eines Systems dauerhafteren Objekten der realen Welt in allen Phasen der Systementwicklung.

Die Vorteile objektorientierter Bibliotheken legen nahe, für die Entwicklung einer Softwarebibliothek zur Verwaltung von Massendaten keine modulorientierte, sondern eine objektorientierte Sprache zu verwenden. Aus diesem Grund wurde im Rahmen dieser Diplomarbeit die bestehende modulorientierte Bibliothek des Tycoon-Systems nicht überarbeitet oder weiterentwickelt, sondern eine Bibliothek in der speziell zu diesem Zweck entworfenen objektorientierten Programmiersprache TooL neu gestaltet und implementiert.

Kapitel 4

Beispiele objektorientierter Softwarebibliotheken

Bereits im vorangehenden Abschnitt wurden die Vorteile objektorientierter Sprachen sowie Vorteile und Ziele der Entwicklung klassenorientierter Bibliotheken vorgestellt. Dieses Kapitel stellt verschiedene objektorientierte Sprachen vor und vergleicht in diesen Sprachen implementierte Bibliotheken für Massendatentypen. Bei der Betrachtung der Sprachen stehen die Eigenschaften im Vordergrund, die beim Design von Klassenbibliotheken von Bedeutung sind. Zu diesen Eigenschaften gehören insbesondere:

Möglichkeiten der Vererbung: Bei der Entwicklung einer objektorientierten Softwarebibliothek spielt das Ziel der möglichst freien Kombinierbarkeit von Funktionalität eine wichtige Rolle. Eine solche Kombinierbarkeit wird durch die Möglichkeit der Mehrfachvererbung unterstützt. Wenn eine Sprache nur eine Einfachvererbung zur Verfügung stellt, muß sich ihre Bibliothek von den Bibliotheken mehrfachvererbender Sprachen unterscheiden, wenn die Mehrfachvererbung in den Bibliotheken der entsprechenden Sprachen genutzt wird.

Kapselung: In rein objektorientierten Sprache kann die Kapselung der Objekte nur im Zuge der Vererbung durchbrochen werden, so daß auch die Exemplare erbender Klassen ohne den Umweg über eine Schnittstelle auf die interne Repräsentation der Daten zugreifen können. Einige objektorientierte Sprachen bieten Mechanismen an, mit denen auch in anderen Situationen eine vom Programmierer zu kontrollierende Durchbrechung der Kapselung von Objekten möglich ist. Wenn der Bibliotheksentwickler diese Möglichkeiten der Durchbrechung der Kapselung nutzen kann, ist es möglich, daß sich die Struktur von der Struktur der Bibliothek einer rein objektorientierten Sprache unterscheidet.

Eigenschaften des Typsystems: Die Unterstützung von Polymorphismus ist eine wichtige Spracheigenschaft. Einige objektorientierte Programmiersprachen bieten beispielsweise zusätzlich zum Inklusionspolymorphismus, der durch die Vererbung erreicht wird, auch die Möglichkeit der Typumwandlung von Objekten oder parametrischen Polymorphismus an.

Einschränkungen beim Überschreiben von Methoden: Die Signatur von geerbten Methoden kann beim Überschreiben nur mit starken Einschränkungen verändert werden. Manche objektorientierte Sprachen bieten dem Entwickler jedoch bezüglich der

Redefinition von Methoden eine größere Freiheit an, indem sie beispielsweise sogar eine Veränderung der Zahl der Parameter zulassen.

Abstraktion: Eine objektorientierte Sprache sollte den Entwickler bei der Abstraktionsbildung unterstützen. Im Rahmen einer objektorientierten Softwarebibliothek sind Abstraktionen sowohl von der konkreten Repräsentation der Daten der Objekte, als auch von den Eigenschaften speziellerer Klassen nötig. Abstraktion ist beispielsweise ein wichtiges Hilfsmittel, um eine einheitliche Sicht auf verschiedene Objekte zu erlangen, indem mit Hilfe von abstrakten Methoden eine konkret Festlegung einiger Details in den allgemeineren Klassen unterbleibt. Klassen müssen also von einigen Details ihrer Subklassen abstrahieren könne, ohne daß die Möglichkeit von Implementierungen in diesen Klassen vermindert wird.

Vor der Entwicklung der in diesem Kapitel untersuchten Bibliotheken der Sprachen C++, Smalltalk und Eiffel waren die Vorteile und Ziele bei der Entwicklung objektorientierter Softwarebibliotheken selbstverständlich bereits bekannt. Auch Veröffentlichungen mit Richtlinien zum „guten“ Design derartiger Bibliotheken waren den jeweiligen Entwicklern bekannt. Dennoch lassen sich bei der Analyse dieser Bibliotheken gravierende Unterschiede feststellen. Diese Unterschiede legen die Vermutung nahe, daß es Unterschiede in den Spracheigenschaften der jeweils verwendeten objektorientierten Programmiersprachen geben muß, die dem Softwareentwickler ein unterschiedliches Design der Bibliothek nahelegen.

Ziel dieses Abschnitts ist es daher, die Unterschiede der untersuchten Sprachen aufzuzeigen und die Bedeutung der Unterschiede für das Design von Softwarebibliotheken herauszuarbeiten. Auf diese Weise wird es nicht nur möglich, die Unterschiede in den Bibliotheken zu beschreiben, sondern auch zu begründen, warum die Verwendung verschiedener Sprachen zu einem abweichenden Resultat führen mußte. Aus dieser Analyse ergeben sich dann auch direkt Schlußfolgerungen über die nötigen Eigenschaften einer objektorientierten Programmiersprache, welche die Realisierung einer Softwarebibliothek für Massendaten unterstützt.

Nachdem bekannt ist, wie sich die Eigenschaften einer objektorientierten Programmiersprachen auf das Design ihrer Klassenbibliothek auswirken, lassen sich auch die Auswirkungen der Eigenschaften von TooL auf das Design der zur Sprachen gehörenden Massendatentypen-Bibliothek begründen.

4.1 C++

Die folgenden Abschnitte stellen zunächst die für die Entwicklung von Bibliotheken wesentlichen Eigenschaften der Sprache C++ dar. Diese Abschnitte der Arbeit sollen natürlich keine vollständige Einführung in die Sprache[1] sein.

Nach der Beschreibung der wichtigsten Eigenschaften der Sprache erfolgt eine beispielhafte Darstellung einer C++-Bibliothek für Massendaten. Dabei wird untersucht, welche Eigenschaften der Sprache sich auf das Design der Bibliothek ausgewirkt haben und wie dieser Einfluß sich in der Bibliothek niederschlägt.

[1] Für die Darstellung der Sprache verweist dieser Abschnitt auf [Stroustrup 92]. Weitere Darstellungen der Sprache finden sich unter anderem auch in [Stroustrup 94; Coplien 92; Lippman 89; Josuttis 94; Ammersaal 95].

4.1.1 Sprache

C++ ist eine objektorientierte Programmiersprache, die Mehrfachvererbung unterstützt. Eine Klasse, die von mehreren Oberklassen erbt, erhält alle ihre Methoden und Variablen. Von dieser Regelung sind nur die mit private gekennzeichneten Teile der Oberklasse ausgenommen. Mit private gekennzeichnete Teile von Klassen sind weder für Objekte einer abgeleiteten Klasse noch für andere Objekte zugreifbar.

4.1.1.1 Mehrfachvererbung

Bei der Mehrfachvererbung kann es zu Mehrdeutigkeiten kommen. Eine Mehrdeutigkeit liegt vor, wenn in beiden Oberklassen eine Methode mit gleichem Namen vorhanden ist. Für einen solchen Fall gibt es in C++ keinen eingebauten Mechanismus, um diese Mehrdeutigkeit aufzulösen. Es muß in der erbenden Klasse diese Methode neu definiert werden. Dies ist ein Unterschied zu TooL. In TooL wird vom Übersetzer mit Hilfe eines Algorithmus[2] eine Methode ausgewählt. Ein möglicher Nachteil des in C++ gewählten Verfahrens liegt in der möglichen Codeverdopplung und Verminderung der Wiederverwendung, wenn eine der geerbten Methoden auch in der erbenden Klasse hätte verwendet werden können. Ein Vorteil der in der Sprache C++ gewählten Vorgehensweise zur Auswahl der zu verwendenden Methode der Oberklassen liegt in der einfacheren Überschaubarkeit und Steuerungsmöglichkeit durch den Programmierer, da es nicht nötig ist, einen in die Sprache eingebauten Algorithmus zur Auswahl der Methode zu berücksichtigen.

Da Vererbung eine Erweiterung der Konzepte einer Klasse und damit auch eine Spezialisierung ihrer Eigenschaften bedeutet, muß der Vererbungsgraph auch in C++ wie in jeder objektorientierten Sprache azyklisch und gerichtet sein [Stroustrup 92, S. 201]. Zyklische Abhängigkeiten zwischen Klassen lassen sich nur mit Hilfe von friend Klassen in einer Benutztbeziehung realisieren [Stroustrup 92, S. 12].

4.1.1.2 Kapselung

Im Gegensatz zu anderen objektorientierten Programmiersprachen kennt C++ 3 Abstufungen der Zugriffskontrolle[3] auf Methoden und Variablen. Die Steuerung der Zugriffskontrolle erfolgt durch die Einteilung der Klassen in 3 Bereiche. Diese Bereiche werden durch die Schlüsselworte public, protected und private abgegrenzt. Ihre Bedeutung ist [Stroustrup 92, S. 47]:

- **Public**. Die durch public gekennzeichneten Variablen und Methoden können von allen anderen Objekten aufgerufen werden. Dieser Teil der Objekte einer Klasse unterliegt also keinerlei Zugriffsbeschränkungen.

- **Protected**. Methoden und Variablen, die nur von Objekten aus abgeleiteten Klassen und der Klasse selbst aufgerufen werden dürfen, kennzeichnet man in C++ als protected. Methoden, die protected sind, entsprechen damit den private Methoden in TooL.

- **Private**. Der als private gekennzeichnete Teil einer Klasse ist nur der Klasse selbst zugänglich. Auch von dieser Klasse abgeleitete Klassen haben keinen Zugriff.

[2] Eine Beschreibung des Algorithmus befindet sich in Abschnitt 6.2

[3] Die im Rahmen dieser Arbeit verwendete Sprache TooL kennt nur 2 Abstufungen.

Eine solche Dreiteilung der Zugriffsberechtigungen gibt es in anderen objektorientierten Programmiersprachen, wie z.B. TooL und Eiffel, nicht. Um Verwirrungen zu vermeiden, hier eine tabellarische Gegenüberstellung:

Schlüsselwort	Bedeutung in C++	Bedeutung in TooL
private	Nur die Objekte der definierenden Klasse haben Zugriff	kein Äquivalent in TooL vorhanden
protected	Objekte aus definierender und erbenden Klassen haben Zugriff	entspricht private von TooL
public	Objekte aller Klassen haben Zugriff	entspricht public von TooL

Tabelle 4.1: Zugriffskontrolle in C++ und TooL

Die Deklaration als private bewirkt nicht nur eine Kapselung gegenüber benutzenden Klassen, sondern auch gegenüber Erben. Ein Vorteil hiervon ist, daß die betreffenden Elemente nur innerhalb der deklarierenden Klasse benutzt werden können und daher eine Veränderung der Semantik weder benutzende noch erbende Klassen fehlerhaft werden läßt. Allerdings reduziert sich der Grad der Wiederverwendung.

Die durch die Deklaration als protected oder private erreichte Kapselung der Objekte kann durchbrochen werden. Als friend deklarierte Methoden haben Zugriff auf alle privaten Teile eines Objekts. Es ist auch möglich, alle öffentlichen Methoden einer anderen Klasse als friend einer Klasse zu definieren.Die Umgehung der Schnittstelle des Objekts ermöglicht in manchen Fällen eine erhebliche Steigerung der Effizienz. In vielen Fällen führen Methoden bestimmte Test, beispielsweise von Bereichsgrenzen, vor dem Zugriff auf die Variablen durch. Der direkte Zugriff ohne Verwendung der Schnittstelle umgeht diese Tests und ist daher schneller.

Dem Effizienzgewinn steht jedoch eine nicht unerhebliche Gefahr des unkontrollierten Zugriffs gegenüber. Auch müßte in einem guten Design erst geklärt werden, ob nicht ein geeignetes Design der Bibliothek oder des Systems die friend-Deklaration vermeiden kann. Hier käme beispielsweise eine gemeinsame Oberklasse der durch friend verbundenen Klassen in Betracht, da diese ohnehin Zugriff auf die internen Strukturen hätte. Interessant ist, daß in [Stroustrup 92, S. 13] empfohlen wird, friend möglichst zu vermeiden.

In [Stroustrup 92, S. 34] wird noch der Begriff des Moduls eingeführt. In der Terminologie der Sprache C++ weicht die Bedeutung des Begriffs jedoch deutlich von der Interpretation in Sprachen wie beispielsweise Modula ab. Während Modula-2 durch Module Definition und Implementierung trennt, sind Module in C++ nur Einheiten der Übersetzung. Von einer inhaltlichen Trennung oder einer Kapselung ist nicht die Rede. Der Begriff Modul ist also in diesem Zusammenhang eher irreführend.

4.1.1.3 Typsystem

Wesentliche Eigenschaften des Typsystems von C++ sind:

- Statische Typisierung
- Typumwandlung

- Überladen von Methoden
- Optionale Argumente:
  Die Angabe der obligatorischen Parameter ist ausreichend.
- Polymorphismus

Statische Typisierung

C++ ist statisch typisiert [Stroustrup 92, S. 44]. Deshalb muß über den Typ eines Objekts ein Minimum an Wissen zur Übersetzungszeit bereits vorhanden sein. So ist beispielsweise auch kein allgemeiner Stack möglich, der Objekte beliebigen Typs als Element aufnehmen kann. In einem solchen Stack wäre über die Elemente nicht bekannt, welche Methoden sie verstehen. Eine Angabe des Elementtyps der Behälterklassen ist daher unverzichtbar. C++ bietet jedoch die Möglichkeit, eine Stack-Schablone (*template*) zu schreiben. Diese Schablone braucht den Typ ihrer Elemente noch nicht zu spezifizieren. Der Typtest der Schablone findet allerdings auch erst bei ihrer Instantiierung mit dem Elementtyp statt. Die Schablone selbst kann also nicht wie eine Klasse verwendet werden, sondern ist nur ein Hilfsmittel, mit dem Klassen erzeugt werden können.

Typumwandlung

C++ kennt mehrere Formen der Typumwandlung (*type cast*). Eine Form der Typumwandlung verändert lediglich den vom Übersetzer angenommenen Typ eines Ausdrucks, während bei einer zweiten Form zusätzlich eine Umwandlung der internen Repräsentation des Objekts erfolgt. Dieser Abschnitt beschreibt beide Formen der Typumwandlung.

Im Rahmen der inkrementellen Erweiterung von Klassen einer objektorientierten Bibliothek werden häufig Objekte im Kontext ihrer jeweiligen Oberklasse verwendet. Für den Vorgang der Übersetzung bedeutet dies, daß Objekte in einem Kontext auftreten, in dem Objekte eines anderen Typs, nämlich des Supertyps, erwartet werden. Wenn der Subtyp im Kontext des Supertyps auftritt, führt der Übersetzer bzw. Typtester eine Konvertierung des Objektes vom Subtyp in den Supertyp durch. Diese Konvertierung bedeutet jedoch nicht, daß zur Laufzeit des Programmes die interne Repräsentation der Objekte verändert würde. Lediglich der vom Übersetzer zur Übersetzungszeit angenommene Typ des Objekts wird verändert. Die interne Repräsentation des Subtyps ist wegen der Spezialisierung des Subtyps bereits für die Operationen im Kontext des Supertyps ausreichend.

Eine typische Anwendung für die Typumwandlung sind die Parameter und Rückgabetypen von Methoden. Diese Parameter sind in der Bibliothek von C++ in der Regel vom Typ Object. Ein Beispiel ist die Methode copy. Auch copy ist so definiert, daß es Object liefert. Erbende Klassen erwarten jedoch ein Objekt des jeweiligen Typs der Klasse selbst. Da der Typ des Ergebnisses als Object angegeben ist, können eigentlich nur die Methoden der Klasse Object auf das Ergebnis angewendet werden. Um die Ergebnisse der jeweiligen Kopiermethode dennoch im normalen Kontext der Klasse des kopierten Objekts verwenden zu können, werden die Rückgabewerte vom Compiler in den jeweiligen Typ umgewandelt. Die interne Repräsentation bleibt dabei unverändert.

Typumwandlung mit gleichzeitiger Veränderung der internen Repräsentation hingegen ist beispielsweise durch explizite Angabe im Quelltext möglich [Stroustrup 92, S. 106f]. Zusätzlich zur expliziten Typumwandlung erfolgt in C++ eine Typumwandlung implizit durch den

Übersetzer, wenn Werte verschiedenen Typs miteinander verknüpft werden sollen [Stroustrup 92, S. 40].

Überladen von Methoden

Wenn der Name zweier Methoden identisch ist, sie jedoch Argumente unterschiedlichen Typs erwarten, spricht man von überladenen Methoden (*overloading*) [Stroustrup 92, S. 142]. Überladene Methoden dienen dazu, ähnliche Aktionen auf Objekten verschiedenen Typs auszuführen. Um den Benutzer Übersicht zu geben, kann der Entwickler der Bibliothek Methoden den gleichen Namen geben. Allerdings erzwingt die Sprache C++ nicht, daß die Methoden wirklich ähnliche Aktionen auf den Daten ausführen. Die Kontrolle liegt hier also allein beim Entwickler. Er kann auch völlig verschiedene Methoden mit Hilfe des Überladens mit dem gleichen Namen versehen. Das Überladen braucht im Quelltext nicht besonders deklariert zu werden. Der Übersetzer sucht abhängig vom Typ der Parameter eines konkreten Aufrufs die jeweils passende Funktionalität der überladenen Methode heraus. Im Fall von Mehrdeutigkeiten weist er den Code zurück.

Optionale Argumente

Es ist in C++ möglich [Stroustrup 92, S. 145], bei der Deklaration einer Methode nur die von ihr mindestens verlangten Parameter anzugeben. Ein konkreter Aufruf einer solchen Methode kann jedoch eine beliebige Zahl weiterer Parameter enthalten. Zum Zeitpunkt der Übersetzung sind weder die Anzahl noch der Typ dieser Parameter bekannt. Zusätzlich gibt es auch optionale Parameter, bei denen ein Standardparameter (*default parameter*) angegeben ist.

Polymorphismus

Die Sprache C++ bietet drei Möglichkeiten, Polymorphismus zu realisieren:

1. Mit Hilfe von (Typ-) Schablonen (*templates*).
2. Durch Zeiger auf Funktionen.
3. Inklusionspolymorphismus

Polymorphismus mit Hilfe von Schablonen

Schablonen ermöglichen die Abstraktion von Elementtypen, indem mit Hilfe eines Typparameters ein neuer Typ erzeugt wird [Stroustrup 92, S. 12, S. 275ff]. Schablonen sind besonders bei der Realisierung von Behälterklassen von Bedeutung. Behälterklassen sollen Funktionalität unabhängig davon anbieten, welchen Typ die in ihnen gespeicherten Objekte zur Laufzeit haben[4].

Da der einer Schablone übergebene Typ unbekannt ist, sind Annahmen über den Typ und damit die Fähigkeiten der Elemente des Behälters unmöglich. Es sind deshalb im Unterschied zur Sprache TooL, die eine Einschränkung des Elementtyps auf eine Menge von Typen erlaubt, keine Mindestanforderungen bekannt, welche die Elemente des Behälters erfüllen.

[4]Ein Verzicht auf diese Forderung führt zu einer Vervielfachung der Behälter, z.B. IntegerStack, RealStack oder CharStack.

Allerdings birgt die in C++ gewählte Regelung auch große Vorteile. Da der Zwang zur Angabe der zulässigen Elementtypen entfällt, erlaubt C++, zunächst die Funktionalität der Schablone zu implementieren, ohne sich Gedanken über den Elementtyp zu machen. Erst bei der Instantiierung einer Schablone wird vom Übersetzer geprüft, ob die Elemente des dann angegebenen Typs den Anforderungen der Schablone genügen.

Dieser Test des Übersetzers ist möglich, weil sich die Anforderungen der Schablone an den zu übergebenden Typ aus dem Code der Schablone selbst herleiten lassen. Der Übersetzer analysiert also den Code der Schablone und erhält auf diese Weise eine Liste von Zugriffen auf die Objekte des Elementtyps. Durch einen Vergleich dieser Anforderungsliste mit der Schnittstelle der Objekte kann die Verträglichkeit von Elementtyp und Schablone zum Zeitpunkt der Instantiierung entschieden werden. Da für die Entscheidung der Typverträglichkeit nur die tatsächlich an die Elemente gestellten Anforderungen herangezogen werden, stellt der Typtest von C++ lediglich Mindestanforderungen an den Typ der Elemente der Schablone.

Polymorphismus mit Hilfe von Zeigern auf Funktionen

Mit Hilfe von Zeigern auf Funktionen lassen sich polymorphe Routinen realisieren [Stroustrup 92, S. 149]. Das dortige Beispiel offenbart jedoch, daß der Polymorphismus eigentlich nicht auf der Fähigkeit der Sprache beruht, Zeiger auf Funktionen zu verwenden. Vielmehr entsteht dieser Polymorphismus aus der Möglichkeit, Funktionen zu verwenden, bei denen der Typ der Parameter nicht spezifiziert werden muß. Diese Art von Polymorphismus ist daher nicht typsicher. Es handelt sich nicht um den in Abschnitt 3.2.2.2 beschriebenen parametrischen Polymorphismus, da dieser einen Typ als Parameter verlangt, der die Typüberprüfung ermöglicht.

Inklusionspolymorphismus

Eine weitere in C++ angebotene Form von Polymorphismus ist der in Abschnitt 3.2.2.2 beschriebene Inklusionspolymorphismus. Dies bedeutet, daß Objekte auch im Kontext ihrer Oberklasse auftreten können.

4.1.1.4 Überschreiben von Methoden

Das Überschreiben von Methoden ist zulässig, wenn sie bei ihrer ersten Definition als virtuell (*virtual*) definiert wurden [Stroustrup 92, S. 204f]. Der Typ einer solchen virtuellen Methode darf bei der Redefinition nicht verändert werden. Allerdings ist es in C++ zulässig, beim Überschreiben die Methode in eine andere Kategorie einzuordnen. So ist es beispielsweise erlaubt, durch Überschreiben einer öffentlichen Methode diese in eine private Methode umzuwandeln. Durch diese Umwandlung ist die Methode nicht mehr Bestandteil der Schnittstelle der erbenden Klasse und kann deshalb dort nicht mehr direkt aufgerufen werden. Der Aufruf der Methode ist nur noch nach vorheriger Umwandlung des entsprechenden Objekts in den Typ der Oberklasse möglich. Diese Möglichkeit ist ein wesentlicher Unterschied zur Programmiersprache TooL. In TooL ist es nicht zulässig, eine geerbte Methode einer anderen Kategorie zuzuordnen.

Bei einer unvorsichtigen Gestaltung der Bibliothek kommt es vor, daß Methoden geerbt werden, die in der erbenden Klasse nicht mehr implementierbar sind, da sie den Eigenschaften der erbenden Klasse widersprechen. Bei der Verwendung von C++ kann in einer solchen Situation eine Veränderung des Entwurfs der Bibliothek vermieden werden, indem die nicht

mehr „passende“ Methode aus der Schnittstelle entfernt wird. Das Entfernen von Methoden aus der Schnittstelle stellt einen Verstoß gegen das Prinzip der inkrementellen Erweiterung und Spezialisierung dar und wird daher von TooL nicht unterstützt.

4.1.2 Die C++-Bibliothek NIHCL

Die Bibliothek NIHCL[5] ist eine Sammlung von Klassen, die denen der Bibliothek von Smalltalk-80 ähnlich sind [Gorlen 90]. Die Bibliothek enthält sowohl allgemein nützliche Datentypen wie String, Date und Time als auch die meisten der Smalltalk-80 Behälterklassen wie z.B. indizierte Felder (OrderedCltn) , einfach verkettete Listen (LinkedList), Hashtabellen (Set) und assoziative Felder (Dictionary). In dieser Arbeit wurde die Version 3.1.4 der Bibliothek untersucht. Einen Überblick über die Bibliothek bietet Abbildung 4.1.

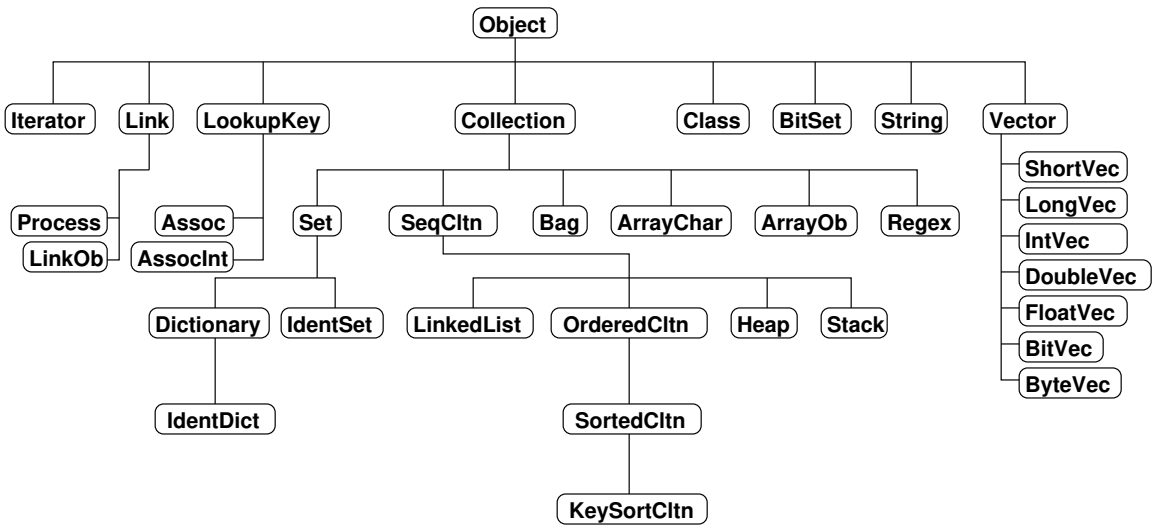

Abbildung 4.1: Ausschnitt der NIHCL-Klassenbibliothek

Die folgenden Abschnitte beschreiben die Organisation der Bibliothek. In NIHCL wird von der Möglichkeit der Mehrfachvererbung kein Gebrauch gemacht. Alle Klassen haben ihren gemeinsamen Ursprung in der Klasse Object. Von besonderem Interesse ist, ob und auf welche Weise die Bibliothek Wiederverwendung von Funktionalität und eine gemeinsame Sicht auf komplexere Objekte verbindet.

Um erkennen zu könne, wie gut die Bibliothek diese Ziele der objektorientierten Programmierung erreicht, müssen zunächst die Behälterklassen der Bibliothek analysiert werden. Eine wichtige Rolle bei dieser Analyse spielt die Frage, in welcher Klasse der Behälterhierarchie welche Methoden eingeführt werden und ob andere Klassen von diesen Methoden durch Wiederverwendung profitieren.

Abbildung 4.2 gibt einen Überblick über die Verteilung der in NIHCL implementierten Methoden auf die Klassen der Bibliothek.

[5] **N**ational **I**nstitute of **H**ealth **C**lass **L**ibrary

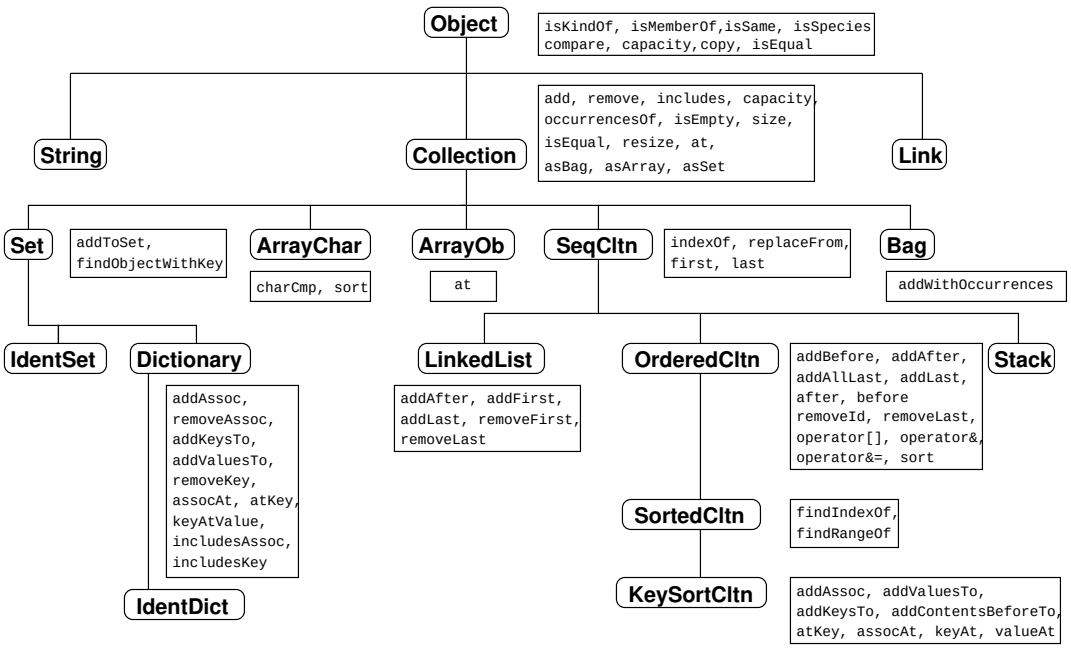

Abbildung 4.2: Verteilung der Methoden in NIHCL

Object

In der Klasse Object werden eine Vielzahl von allgemeinen Definitionen vorgenommen. So findet sich hier die Definition von Vergleichsmethoden (compare, isEqual) und kopierenden Methoden (copy, deepenShallowCopy und shallowCopy) . Auch Tests auf Zugehörigkeit eines Objekts zu einer bestimmten Klasse (isKindOf, isMemberOf, isSame und isSpecies) werden bereits in Object definiert. Überraschend ist die bereits in dieser Klasse erfolgende Einführung einer Kapazität (capacity).

Collection

Collection ist eine abstrakte Basisklasse für die Behälterklassen der NIHCL. Collection implementiert die virtuellen Methoden, die auf alle Behälter angewendet werden können [Gorlen 90, S. 37]. In Collection findet man deshalb Methoden zum Einfügen (add, addAll, addContentsTo) und Löschen (remove removeAll, removeCltn) von Elementen. Mit Hilfe von printOn kann der Inhalt des Behälters auf einen Stream ausgegeben werden. Zusätzlich enthält Collection Methoden zum Vergleichen von Behältern (compare, sameContentsAs, isEqual). Einige weitere Methoden geben Informationen über den Zustand des Behälters (capacity, includes, occurrencesOf, isEmpty, size).

Auffallend ist die Implementierung von Methoden zur Konvertierung der Behälter in einen anderen Behältertyp (asArrayOb, asBag asHeap, asOrderedCltn, asSet und asSortedCltn). Für jeden gewünschten Ziel-Datentyp muß eine eigene Methode für die Konvertierung implementiert werden. Die in NIHCL gewählte Lösung unterscheidet sich deutlich von der in der TooL-Bibliothek gewählten (Abschnitt 7.6). In der TooL-Bibliothek werden für die Konver-

tierung von Behältern Ströme (*streams*) von Elementen verwendet. Durch die Verwendung von Strömen brauchen in jeder Klasse nur zwei Methoden realisiert zu sein, um eine Konvertierung zu ermöglichen. Dies ist eine Methode, die einen Strom von Elementen des Behälters erzeugt, sowie eine Methode, die aus einem Strom von Elementen einen Behälter des jeweiligen Typs erzeugt. Durch diesen Mechanismus bleiben die Klassen erweiterbar in Bezug auf die Methoden zur Konvertierung. Auch wenn neue Klassen hinzukommen, braucht in bestehende Klassen keine weiter Methode zur Konvertierung in die neue Klasse eingefügt zu werden.

Mit den bisher beschriebenen Möglichkeiten liefert Collection aus NIHCL ungefähr die Mächtigkeit einer Kombination der TooL-Klassen Boxed und Accessible. Es gibt jedoch auch deutliche Abweichungen. Collection definiert eine Methode zur Veränderung der Kapazität (reSize). Ergänzend werden Faktoren deklariert (DEFAULT_CAPACITY, EXPANSION_INCREMENT, EXPANSION_FACTOR), die auf reSize Einfluß haben. Eine weitere Abweichung vom Design der TooL-Bibliothek stellt die in Collection erfolgende Einführung einer Methode zum indizierten Zugriff auf die Elemente des Behälters dar (at). In NIHCL sollen also alle Behälter

- eine Kapazität kennen. Dies ist eine recht überraschende Annahme, denn beispielsweise eine durch Zeiger verbundene Liste benötigt keine Kapazität. Ein Benutzer erwartet bei einer solchen Liste, daß unabhängig von der Zahl der bereits enthaltenen Elemente stets neue eingefügt werden können. Allein die Größe des vorhandenen Speichers begrenzt damit die Liste. Eine durch Zeiger verbundene Liste hat also entweder keine oder eine unendliche Kapazität. Eine solche Liste ist auch als Subklasse von Collection in NIHCL vorhanden. Durch die frühzeitige Einführung von Kapazitäten in der Vererbungshierarchie ergibt sich in der Klasse LinkedList ein Widerspruch.

- in der Kapazität veränderlich sein. Die Einführung von Kapazitätsänderungs-Faktoren unterbindet eine Unterscheidung zwischen Behältern mit fester und solcher mit variabler Kapazität.

- einen indizierten Zugriff bieten. Es ist zweifelhaft, ob ein indizierter Zugriff auf die Elemente bei jedem Behälter sinnvoll ist. Zum Beispiel bei rekursiven Listen oder einer Hashtabelle scheint ein solcher Zugriff nicht sinnvoll.

Set

„Ein Set ist eine ungeordnete Menge von Objekten, in der kein Objekt doppelt enthalten ist. „Doppelt“ vorhanden wäre ein Objekt, wenn zwei Objekte von isEqual als gleich erkannt würden. Mengen werden unter Verwendung einer Hashtabelle realisiert. Die Methode capacity gibt die Hälfte der Kapazität der Hashtabelle zurück, size liefert die Anzahl der aktuell in der Menge enthaltenen Objekte. Aus Gründen der Effizienz ist die Kapazität immer eine Potenz von 2. Die Kapazität wird automatisch verdoppelt, wenn die Menge die Kapazitätsgrenze erreicht“[6]. Wie in TooL werden auch in der NIHCL Methoden für die typischen Operationen auf Mengen implementiert.

In NIHCL wird kein Unterschied zwischen Set und Hashtable gemacht. Im Kommentar von Set.h findet man: „Set.h - declarations for hash tables“. Dies ist ein Unterschied zur TooL-Bibliothek. In TooL werden sowohl Set als auch HashTable realisiert. Der Grund für

[6] Übersetzung aus einem Kommentar des Quelltextes.

dieses Vorgehen sind Methoden wie Vereinigung oder Durchschnitt, die man bei einer Menge erwartet, nicht jedoch bei einer Hashtabelle. In TooL verwendet der Set die HashTable.

In der Klasse Set finden sich einige Hinweise auf zweifelhafte Design-Entscheidungen. So sind für das Einfügen von Elementen nun zwei Methoden vorhanden (addToSet und add). Die Methode add wurde bereits in Collection definiert. Aus nicht ersichtlichen Gründen wird ihr nun addToSet hinzugefügt. Unverständlich bleibt diese zusätzliche Methode besonders dadurch, daß add nur als Aufruf von addToSet definiert wird. Die Funktionalität beider Methoden ist also absolut gleich. Deshalb hätte man auch gleich add mit der gewünschten Funktionalität überschreiben können.

Völlig isoliert von anderen Design-Entscheidungen wird in Set.c noch der Begriff des Schlüssels eingeführt (findObjectWithKey). Dies ist doppelt irreführend, da von findObjectWithKey keine Schlüssel verwendet werden. Es wird lediglich die Menge nach einem Objekt durchsucht, das die Eigenschaft isEqual(ob) erfüllt.

Einen Hinweis auf Designfehler liefert auch die Überschreibung von compare. Um den Aufruf von geerbten Methoden zu unterbinden, die in einer Klasse nicht mehr vorhanden sein sollen, benutzt der Autor von NIHCL die Methode shouldNotImplement. Diese Methode gibt eine Fehlermeldung aus. Die in einer Klasse störende, weil nicht mehr passende oder schwer zu implementierende Methode wird dann einfach als Aufruf von shouldNotImplement realisiert:

```
int Set::compare(const Object&) const
{ shouldNotImplement("compare");
  return 0;
}
```

Gleichzeitig wird die Methode noch von einer öffentlichen in eine private Methode umgewandelt. In Collection.h war compare noch als öffentliche Methode eingeführt worden. Wie in Abschnitt 4.1.1.4 beschrieben, wird compare nun in Set.h in eine private Methode umgewandelt:

```
private:   // shouldNotImplement()
  virtual int compare(const Object&) const;
```

Arraychar

Arraychar implementiert Byte-Felder. Der größte Teil des Codes ist über eine Makroschablone realisiert. Diese Makroschablone definiert Makros, die für Felder von eingebauten Typen char, int, short, long, unsigned, float und double verwendet werden können. Arraychar bietet eine Methode zum Sortieren der Elemente an. Dabei stammt die Funktionalität der Vergleichsfunktion nicht wie bei TooL direkt aus den Elementen des Arrays, sondern wird in Arraychar selbst implementiert. Auch in Arraychar werden Methoden in private umgewandelt. Im einzelnen sind dies add, at, doNext, occurrencesOf und remove.

ArrayOb

ArrayOb realisiert eine Klasse, deren Elemente Zeiger auf Objekte sind. Bei der Initialisierung wird das Feld mit Zeigern auf das Nil-Objekt gefüllt. In ArrayOb wird auch reSize implementiert, so daß das Feld in der Kapazität angepaßt werden kann. Eine Methode zum Entfernen aller im Array enthaltenen Elemente kann in TooL durch die Verwendung von Streams bereits in der allgemeineren Klasse Accessible definiert werden. Bei NIHCL wird sie erst in ArrayOb

implementiert (removeAll). Wie viele andere Klassen bleibt auch ArrayOb nicht vom „Privatisieren“ von Methoden verschont. Hier sind es die Methoden add, occurrencesOf und remove.

SeqCltn

SeqCltn ist eine abstrakte Klasse, die Behälter repräsentiert, deren Elemente geordnet sind und über ganzzahlige Indizes zugegriffen werden können[7]. Obwohl bereits in Collection at zum indizierten Zugriff auf die Elemente des Behälters enthalten ist, soll SeqCltn gegenüber dem Vorgänger Collection Indizes als Eigenschaft hinzubekommen. Darum werden in SeqCltn allgemeine, mit einem Index zusammenhängenden Methoden implementiert (indexOf, replaceFrom, first und last). Die Eigenschaften von SeqCltn kollidieren nicht mit den geerbten Methoden. Deshalb ist es hier nicht notwendig, den Aufruf von Methoden durch Einordnung als private Methode zu erschweren.

Bag

Ein Bag entspricht einem Set, nur daß in Bags gleiche Elemente auch mehrfach auftreten können. Bei der Implementierung von Bag wird ein Dictionary verwendet, um jedes Objekt in Bag mit der Anzahl seines Auftretens zu assoziieren[8]. Die Gleichheit von Objekten wird dabei von isEqual festgestellt. Beim Einfügen eines bereits vorhandenen Objekts oder beim Löschen eines Objekts wird der entsprechende Zähler aktualisiert [Gorlen 90, S. 19].

Bag führt keine wesentlichen Erweiterungen gegenüber seiner Oberklasse Collection durch. Lediglich eine Methode zum Einfügen mehrerer Elemente, die beim Einfügen den Zähler nicht auf 1, sondern gleich auf einen vorgegebenen Wert erhöht, kommt hinzu (addWithOccurrences). Unklar ist, warum die Methode compare nicht implementiert, sondern als private Methode überschrieben wird.

LinkedList

Verkettete Listen sind durch die Reihenfolge des Einfügens und Löschens der Elemente geordnet. Die Elemente sind über Indizes zugreifbar[9]. LinkedList ist eine einfach vorwärtsverkettete Liste. Jedes in ihr enthaltene Objekt muß ein Exemplar einer von Link abgeleiteten Klasse sein. Benutzer können eine solche Klasse selbst implementieren oder die Klasse LinkOb verwenden, um Zeiger auf Objekte in die LinkedList einzufügen [Gorlen 90, S. 105]. Diese Wahlmöglichkeit führt dazu, daß einige Methoden in zwei Versionen vorhanden sind, die sich durch den Typ ihres Parameters unterscheiden (Link bzw. Object). Die Methoden addFirst, addLast addAfter, removeFirst und removeLast werden in LinkedList neu eingeführt.

Der in Collection eingeführte indizierte Zugriff auf die Elemente der Behälter wird in LinkedList aufrechterhalten. Unverständlich ist jedoch, weshalb die geerbte Methode at in eine private umgewandelt wird und statt dessen ein Operator eingeführt wird (operator[](int i)). Da durch diesen Operator aber ebenfalls das Element mit dem als Parameter übergebenen Index zurückgegeben wird, bleibt die Veränderung der Bezeichnung der Methode unverständlich.

Auch die im direkten Vorgänger SeqCltn eingeführte Methode atAllPut, die alle Elemente gegen ein anderes austauscht, wird in LinkedList privat. Es ist unklar, warum diese Methode

[7] Übersetzung eines Kommentars aus SeqClt.c
[8] Übersetzung aus dem Quelltext von Bag.c
[9] Übersetzung aus dem Quelltext von LinkedList.c

hier nicht implementiert werden soll. Auch bei den ebenfalls in private Methoden umgewandelten indexOfSubCollection und replaceFrom scheint eine Implementierung in LinkedList ohne Schwierigkeiten möglich.

Die bereits bei der Beschreibung von Collection in Abschnitt 4.1.2 geäußerte Vermutung bezüglich der Kapazität bestätigt sich in LinkedList. In einer verketteten List gibt es in NIHCL keine oder unendliche Kapazität, so daß reSize keine Funktionalität erhält. Aus nicht ersichtlichen Gründen verwendet NIHCL hierzu nicht die bereits an zahlreichen Stellen verwendete Umwandlung in eine private Methode, sondern reduziert die Funktionalität von reSize einfach auf eine Nicht-Operation.

OrderedCltn

Geordnete Behälter sind durch die Reihenfolge geordnet, in der Objekte in sie eingefügt oder aus ihnen gelöscht werden. Die Elemente einer OrderedCltn sind über einen Index zugreifbar[10].

OrderedCltn implementiert ein Feld veränderlicher Größe, das Zeiger auf Objekte enthält. Die Zeiger sind über einen bei Null beginnenden Index zugreifbar, wobei Bereichsüberschreitungen durch Tests entdeckt werden. Mit add werden Objekte am Ende des Feldes beim Index size abgelegt [Gorlen 90, S. 169].

OrderedCltn erweitert die Funktionalität gegenüber dem Vorgänger SeqCltn um Methoden zum Einfügen nach zu suchenden Objekten (addBefore, addAfter) und hinter allen Elementen (addAllLast, addLast). Mit after bzw. before werden die Elemente hinter bzw. vor einem anderen gesucht. Die zum Löschen eingeführte Methode removeId verwendet zur Identifikation des Zielelements nicht isEqual, sondern die Identität. Ein Löschen des letzten Elements bietet removeLast. Zusätzlich zum geerbten indizierten Zugriff (at) wird ein Operator (operator[](int i)) mit gleicher Funktionalität angeboten. Weitere Operatoren bieten ein funktionales Anfügen aller Elemente einer anderen OrderedCollection sowie diese Funktionalität mit Seiteneffekt. Mit sort ist eine Sortierung der Elemente in aufsteigender Reihenfolge mögliche.

Unklar bleibt der Grund für die Aufteilung in SeqCltn und OrderedCltn, da die in OrderedCltn gegenüber SeqCltn hinzugekommenen Methoden nicht im Widerspruch zu den Eigenschaften einer SeqCltn stehen. Diese Methoden hätte man auch in der Klasse SeqCltn bereits einführen können. Es werden in OrderedCltn keine geerbten öffentlichen Methoden in private umgewandelt.

Auch der Begriff einer geordneten Sammlung von Objekten ist etwas irreführend, da von der OrderedCltn keine Reihenfolge der Objekte erzwungen wird, die in einem Zusammenhang mit den Eigenschaften der Objekte steht. Es stimmt auch nicht, daß die Reihenfolge der Elemente der SortedCltn der Reihenfolge ihres Einfügens entspricht. Die Methoden addBefore, addAfter, removeId und sort bieten dem Benutzer der OrderedCltn die Möglichkeit, eine beliebige Reihenfolge der Elemente unabhängig von der Reihenfolge des Einfügens zu erreichen. Eigentlich handelt es sich deshalb bei der OrderedCltn nicht um einen geordneten, sondern um einen sortierbaren Behälter.

[10] Übersetzung aus dem Quelltext von OrderedCltn.c

SortedCltn

Eine SortedCltn ist ein Behälter, dessen Elemente gemäß der Funktion compare geordnet sind. Die Methode compare muß von den einzufügenden Objekten selbst bereitgestellt werden. Die Methode add stellt durch eine binäre Suche den Index fest, an dessen Position ein einzufügendes Objekt positioniert werden muß. Anschließend ruft add die private Methode OrderedCltn::addAtIndex auf, um das Objekt an der entsprechenden Stelle einzufügen. Um an der gewünschten Stelle Platz zu schaffen, schiebt addAtIndex alle dahinter liegenden Elemente im Feld einem Index weiter nach hinten. Aus diesem Grund ist SortedCltn für große Datenmengen ineffizient[11].

SortedCltn führt eine neue öffentliche Methoden ein (findIndexOf), die durch eine binäre Suche den Index eines Objekts im Behälter sucht. Da in SortedCltn gleiche Objekte mehrfach vorkommen dürfen, wird nicht immer das erste entsprechende Element gefunden. Eine andere neue Methode (findRangeOf) liefert einen Bereich von Objekten, die dem Suchkriterium genügen.

Wegen der Ordnung der Elemente ist es unzulässig, Elemente an beliebigen Positionen einzufügen. Diese Sortiertheit ist mit zahlreichen geerbten Methoden unverträglich. Aus diesem Grund wurden die Methoden addAfter, addAllLast, addBefore, addLast, atAllPut, indexOfSubCollection, replaceFrom und sort in die Kategorie der privaten Methoden verschoben.

KeySortCltn

Eine KeySortCltn unterscheidet sich von der SortedCltn dadurch, daß die Elemente nach einem separaten Schlüssel sortiert werden. Jedes Element ist ein Paar aus Schlüssel und Wert. Die Sortierung der Elemente erfolgt jedoch unter ausschließlicher Berücksichtigung der Schlüssel [Gorlen 90, S. 95]. Wenn die Kapazität durch sukzessives Einfügen erschöpft ist, wird die Kapazität durch resize automatisch erhöht, so daß der Benutzer sich nicht um die Kapazität zu kümmern braucht.

In KeySortCltn werden neue Methoden eingeführt, die den Umgang mit Kombinationen aus Schlüssel und Wert in einem sortierten Behälter ermöglichen. Dies sind Methoden zum Einfügen und Suchen (addAssoc, addValuesTo, addKeysTo, addContentsBeforeTo, atKey, assocAt, keyAt und valueAt).

In KeySortCltn werden keine geerbten öffentlichen Methoden in private Methoden umgewandelt. Dies ist erstaunlich, da deshalb auch die geerbte Methode add(Object& ob) noch zur Schnittstelle gehört. Bei add wird jedoch nur ein Parameter übergeben, während die KeySortCltn beim Einfügen Schlüssel und Wert erwartet.

Dictionary

Dictionary unterscheidet sich von Set dadurch, daß die Elemente Assoziationen aus Schlüssel und Wert sind. Das Dictionary gibt zu einem gegebenen Schlüssel den Wert zurück[12]. Die Gleichheit von Schlüsseln wird durch die Anwendung der virtuellen Methode isEqual festgestellt [Gorlen 90, S. 53].

Die geerbte Methode add erwartet als Parameter einen Zeiger auf ein Objekt. Diese Signatur darf nicht verändert werden. Da aber in Dictionary nur Elemente eingefügt werden

[11] Übersetzung aus dem Quelltext von SortedCltn.c
[12] Übersetzung aus dem Quelltext von Dictionary.c

dürfen, die Kombinationen aus Schlüssel und Wert sind, muß die Verträglichkeit des Arguments zur Laufzeit getestet werden. Dies stellt eine Schwäche von C++ gegenüber TooL dar. In TooL hätte man bereits bei der Definition der Vererbungsbeziehung den Typ der Elemente spezifiziert. In add wird deshalb eine Zusicherung eingebaut, die diesen Test zur Laufzeit übernimmt.

Da sich der Elementtyp gegenüber der Oberklasse Set geändert hat, werden in Dictionary einige neue Methoden eingeführt. Diese Methoden realisieren das Einfügen (addAssoc, addKeysTo und addValuesTo), Löschen (removeAssoc, removeKey) und Suchen (assocAt, atKey, keyAtValue und includesAssoc) im Zusammenhang mit Schlüsseln. Die Methoden addKeysTo und addValuesTo wurden auch in KeySortCltn eingeführt. Somit muß hier Code doppelt implementiert werden. Wenn atKey als zweiter Parameter noch ein neuer Wert übergeben wird, überschreibt die Methode den alten mit dem neuen Wert. Diese Methode ist damit ein Beispiel für die Anwendung der in Abschnitt 4.1.1.3 beschriebenen optionalen Parameter. Die Methode compare wird aus nicht ersichtlichen Gründen in eine private Methode umgewandelt.

Stack

Stack implementiert einen Stapel, dessen Elemente Zeiger auf Objekte sind. Elemente können mit push auf den Stapel gepackt werden. Durch pop wird das oberste Element entfernt, dessen Index 0 ist. Die Methode top liefert das oberste Element. Auch Elemente, die innerhalb des Stapels liegen, können über ihren Index direkt zugegriffen werden [Gorlen 90, S. 215].

In Stack werden weitere Methoden eingeführt, die jedoch lediglich bereits vorhandene Funktionalität unter einem anderen Namen anbieten. Im einzelnen sind dies removeLast (wie pop), operator[](int i) (wie at) und last (wie top). Auch in Stack müssen wieder einige geerbte Methoden in private umgewandelt werden (atAllPut, indexOfSubCollection, remove, replaceFrom und startAt).

4.1.3 Bewertung der NIHCL

Verwirrend bei der Untersuchung der Bibliothek ist, daß in den C++-Deklarationsdateien (*header files*) die Schlüsselworte public, protected und private mehrfach auftreten dürfen. Leider verzichtet C++ hier auf die Übersichtlichkeit der Sprache TooL, die entsprechende Schlüsselworte nur einmal zuläßt.

Übersichtlichkeit der Bibliothek

In den Kapiteln 2 und 3.2 wurden als Ziele des Entwurfs einer klassenorientierten Softwarebibliothek Übersichtlichkeit und Verständlichkeit besonders betont. Bibliotheken sollten danach durch ihr Angebot einer Hierarchie von Klasse dem Benutzer ein einfache Sicht auch auf die Schnittstellen der komplizierteren Klassen erlauben.

Verständlichkeit und Übersichtlichkeit einer Bibliothek setzen beispielsweise voraus, daß die Eigenschaften der Klasse in der Bibliothek in einer klar gegliederten Form durch Spezialisierung entwickelt sind. Nur wenn diese Bedingung erfüllt ist, kann de Benutzer durch ein Absteigen in der Vererbungshierarchie eine Klasse mit den gewünschten Eigenschaften finden. Um zu überprüfen, wie weit die vorliegende Bibliothek dieses Ziel erreicht, zeigt Abbildung 4.3, in welche Kategorien man die Klassen der NIHCL einteilen kann. Bei der Betrachtung wird deutlich, daß die Gliederung der Bibliothek dem Benutzer an vielen Stellen nur geringe Hilfestellungen anbietet.

Zwischen den unterhalb der Klasse Collection liegenden Klassen ArrayChar, ArrayOb und SeqCltn kann der Benutzer auf den ersten Blick keine Unterscheidung feststellen. Alle 3 Klassen greifen mit Hilfe von Indizes auf die Elemente der Klasse zu. Ungünstig scheint auch die Unterscheidung unterhalb der SeqCltn, da hier auf den unterschiedlichen Elementtyp (Link bzw. Object) zwischen LinkedList auf der einen und Stack bzw. OrderedCltn auf der anderen Seite abgestellt wird. Eine ebensolche Unterscheidung nach dem Elementtyp erfolgt auch zwischen IdentSet und IdentDict.

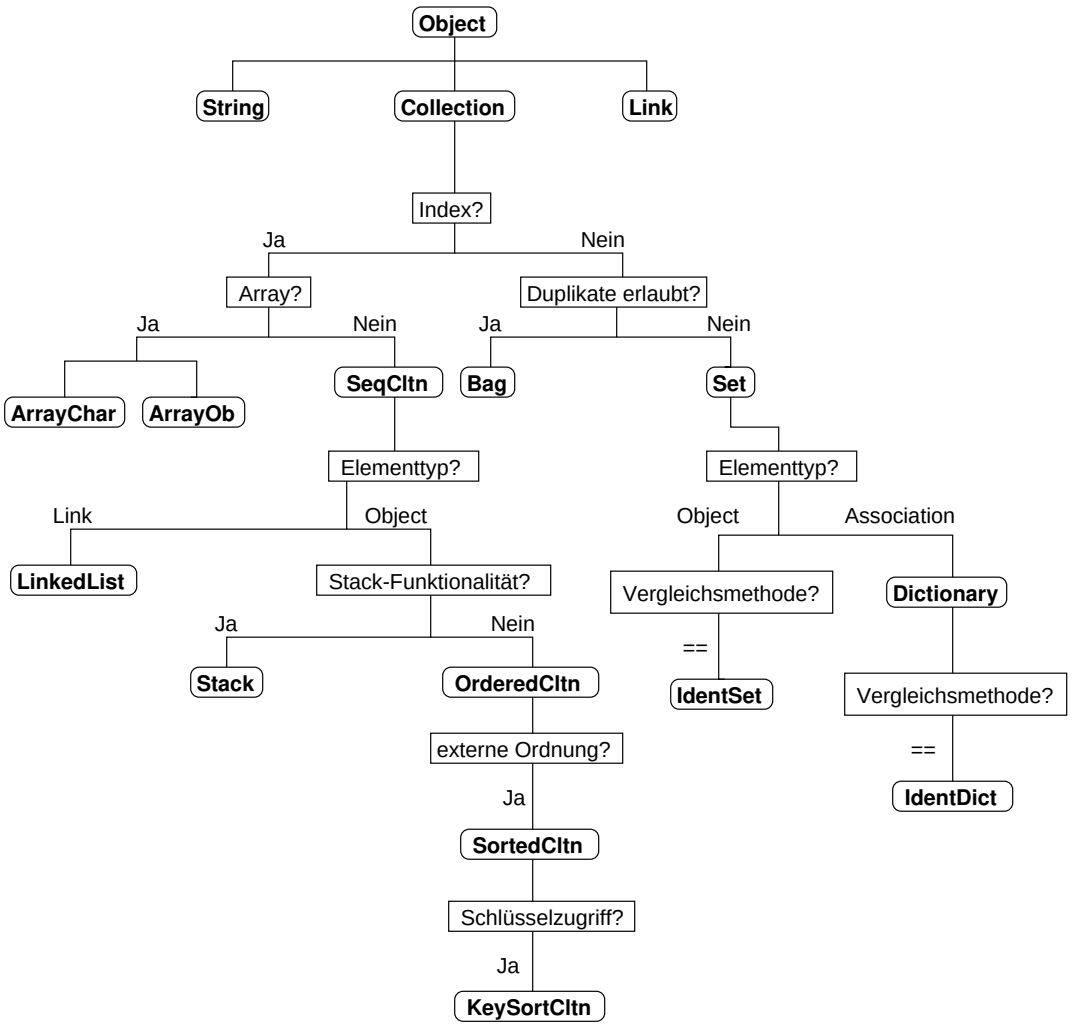

Abbildung 4.3: Kategorisierung der Klassen in NIHCL

Einfache Sicht auf komplexe Schnittstellen

Auffällig ist der Verzicht auf die Verwendung der von der Sprache angebotenen Mehrfachvererbung. In Anlehnung an die Smalltalk-Bibliothek, das Vorbild von NIHCL, wurde NIHCL als Klassenbaum realisiert. Jede Klasse aus der Bibliothek hat also genau einen Vorgänger.

Um den Grad der Wiederverwendung innerhalb der Bibliothek zu erhöhen, liegt es nahe, Methoden so dicht wie möglich an der Wurzel des Klassenbaums zu implementieren. Allerdings stößt diese Vorgehensweise auch schnell auf ein Problem. Die Summe der öffentlichen Methoden eines Objekts bestimmt nämlich gerade die Eigenschaften eines Objekts. Die Eigenschaften eines Objekts sind also untrennbar mit der Funktionalität verbunden. Eine Implementierung dicht an der Wurzel eines Klassenbaums bedeutet also gleichzeitig eine frühzeitige Festlegung der Eigenschaften der darunter liegenden Klassen.

Gerade in dieser frühzeitigen Festlegung von Eigenschaften liegt das Problem beim Design einer Bibliothek aus erbenden Klassen. Durch den Verzicht auf die Verwendung von Mehrfachvererbung wird dieses Problem noch verschärft. Je früher eine Eigenschaft in der Hierarchie für alle erbenden Klassen festgelegt wird, um so größer ist die Gefahr, bei der Realisierung erbender Klassen mit dieser dann geerbten Eigenschaft in Konflikt zu geraten.

Oft zeigt sich dann bei der Entwicklung speziellerer Klassen im fortgeschrittenen Stadium der Bibliotheksentwicklung, daß öffentliche Methoden geerbt werden, die nicht mehr zu den geplanten Eigenschaften der tieferliegenden Klassen passen. In der Bibliothek NIHCL ist dies in zahlreichen Klassen zu beobachten. Es handelt sich dabei um folgende bereits besprochenen (1.) bzw. hier nicht beschriebene Klassen (2.):

1. Set, Arraychar, Array_c.m4 und Array_h.m4 (Schablonen für Felder), ArrayOb, Bag, LinkedList, SortedCltn, Dictionary und Stack

2. Iterator, Nil, Random, Range, Rectangle, Regex, BitBoard, Bitset, Class, FDSet und Heap

Alle diese Klassen erben öffentliche Methoden, die im Widerspruch zu ihren Eigenschaften stehen. Abbildung 4.4 zeigt, an welchen Stellen in der Bibliothek geerbte öffentliche Methoden mit den Eigenschaften der Klasse kollidieren.

Nicht mehr „passende“ geerbte öffentliche Methode können in C++ durch Redefinition in eine private Methode umgewandelt und dadurch aus der Schnittstelle der betreffenden Klasse entfernt werden. Es ist jedoch nicht unbedenklich, beim Design einer Klassenbibliothek von diesem Werkzeug Gebrauch zu machen. Es zeigt sich ein Zielkonflikt zwischen den beiden Zielen der klassenorientierten Bibliotheksentwicklung, der Erhöhung der Wiederverwendung und der gewünschten gemeinsamen einfachen Schnittstelle für Objekte erbender Klassen:

- Einerseits bietet die Möglichkeit der nachträglichen Einordnung als private Methode den Vorteil, auch Klassen beerben zu können, deren Eigenschaften nicht vollständig zu denen der erbenden Klasse passen, was die Wiederverwendung auch an dieser Stelle der Bibliothek ermöglicht.

- Andererseits verliert der Benutzer der Bibliothek die gemeinsame Schnittstelle für verschiedene kompliziertere Objekte. Eine gemeinsame Sicht auf Objekte verschiedener Klassen ist nur gegeben, wenn der Benutzer bei der Verwendung einer Klasse darauf vertrauen kann, auch alle öffentlichen Methoden der darüber liegenden Oberklassen aufrufen zu können. In NIHCL kann der Benutzer sich angesichts der massiven Verwendung des „Versteckens“ von geerbten Methoden aber gerade nicht mehr auf die Anwendbarkeit einer Methode aus einer Oberklasse verlassen. Der Benutzer muß sich vor der geplanten Verwendung einer Methode einer Oberklasse vergewissern, daß diese Methode nicht auf dem Weg des Vererbungsgraphen von der einführenden Klasse zur aktuellen Klasse „versteckt“ wurde.

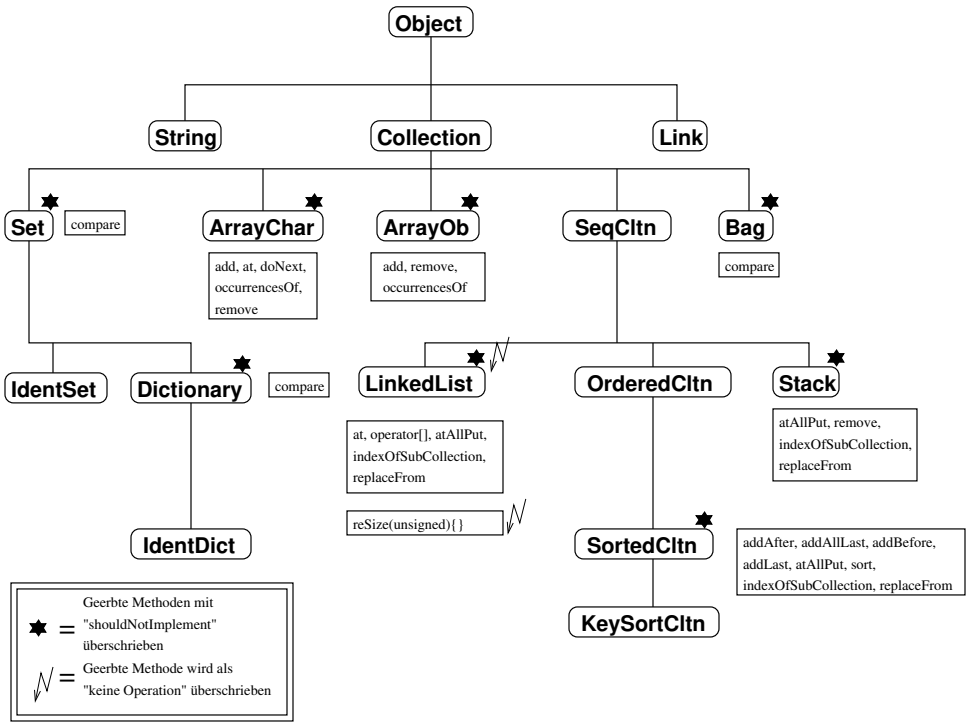

Abbildung 4.4: Verstecken geerbter Methoden in NIHCL

Die Sprache TooL bietet keine Möglichkeit, geerbte öffentliche Methoden in private umzuwandeln. Geerbte Funktionalität bleibt damit auch immer Bestandteil der Schnittstelle einer erbenden Klasse. Durch diese Eigenschaft erzwingt TooL eine Gleichrangigkeit der Ziele „Wiederverwendung“ und „gemeinsame Schnittstelle für Objekte verschiedener Klassen“. Die Bibliothek NIHCL ließe sich in TooL nicht implementieren. Dies zeigt, wie stark die Auswirkungen des Sprach-Designs auf das Design und den Stil einer in dieser Sprache verwirklichten klassenorientierten Softwarebibliothek sind.

4.2 Vererbung: Ein „Produzentenmechanismus“?

Die Möglichkeit, geerbte öffentliche Methoden aus der Schnittstelle einer erbenden Klasse wieder zu entfernen, durchbricht das Prinzip der Erweiterung und Spezialisierung von Konzepten im Rahmen der Implementierung einer Klassenhierarchie[13]. Dennoch bieten mehrere objektorientierte Programmiersprachen diese Möglichkeit an[14]. Bei der Entwicklung einer Klassenbibliothek ergibt sich somit die Möglichkeit, nur einen Teil einer Klasse zu erben. Ein anderer Teil der Oberklasse, der eigentlich mit geerbt werden würde, kann aus der Schnittstelle der Subklasse wieder entfernt werden, so als ob diese Methoden nie geerbt worden wären. Dieser

[13] Einige Autoren halten eine Durchbrechung des Prinzips der Spezialisierung für unumgänglich: *„Von Zeit zu Zeit müssen Sie aber eine Unterklasse erzeugen, die nur einen Teil der Oberklasse enthält, obwohl das eine Verletzung des Prinzips von Verallgemeinerung und Spezialisierung ist.“* [Coad, Nicola 94].

[14] Dies sind unter anderem die Sprachen C++, Smalltalk und Eiffel.

Vorgang kann deshalb auch als „partielles Erben“ bezeichnet werden. Die Beschreibung der NIHCL zeigt, daß von der Möglichkeit des partiellen Erbens bei ihrer Entwicklung umfangreich Gebrauch gemacht wurde. Im folgenden wird beschrieben, aus welchem Grund partielles Erben in einigen objektorientierten Programmiersprachen enthalten ist und welche Auswirkungen die Verwendung dieser Möglichkeiten auf die Benutzbarkeit einer objektorientierten Bibliothek hat.

Die Beobachtungen der bisherigen Abschnitte dieses Kapitels lassen die Frage aufkommen, weshalb C++ als objektorientierte Programmiersprache partielles Erben zuläßt, obwohl diese Möglichkeit eigentlich eine Durchbrechung objektorientierter Prinzipien darstellt. Der Grund hierfür ist, daß Vererbung gemäß einer wachsenden Übereinkunft[15] in erster Linie dem Anbieter einer Bibliothek Nutzen bringt. Diese Betrachtung läßt sich in folgender These zusammenfassen:

These:

> Der Mechanismus des Vererbens stellt in erster Linie eine Unterstützung des „Produzenten“ einer Bibliothek dar, während die Benutzer, die „Konsumenten“, eine Bibliothek auf völlig andere Weise verwenden [Meyer 90].

Die These, Vererbung sei ein „Produzentenmechanismus“, kann dadurch begründet werden, daß der Nutzer einer Bibliothek sich in erster Linie für die Schnittstellen der angebotenen Klassen interessiert. Für den Benutzer einer Bibliothek wäre deshalb weniger wichtig, von welcher Klasse eine zu verwendende Klasse abgeleitet wurde, sondern welche Methoden die Klasse bietet. Der Benutzer würde sich also für das Angebot interessieren, das ihm eine Bibliothek macht, nicht für die Art und Weise, in der dieses Angebot vom Bibliotheksentwickler mit Hilfe der Vererbung realisiert wird.

In den bisherigen Kapiteln wurden deutliche Hinweise darauf gefunden, daß Vererbung bei geschicktem Einsatz auch wesentliche Vorteile für den Benutzer einer Bibliothek bietet. Diese Überlegungen legen nahe, Vererbung nicht als “Produzentenmechanismus“ aufzufassen und führen zur Gegenthese:

Gegenthese:

> Geschickt angewendet ist Vererbung eine Möglichkeit, die Übersichtlichkeit der Bibliothek zu steigern. Vererbung kann durch ihre Möglichkeit der Klassifizierung von Eigenschaften der Behälter bei der Auswahl des zu verwendenden Behälters unterstützen. Darüber hinaus bietet Vererbung in Form einer Konzept- oder Typspezialisierung eine gemeinsame Sicht auf die Objekte verschiedener komplexer Klassen, wodurch sich der Umgang des Benutzers mit den Klassen wesentlich vereinfacht.

Die folgenden Abschnitte beschreiben die Konsequenzen der Anwendung partiellen Erbens für den Benutzer klassenorientierter Bibliotheken. Hierbei ist besonders zu beachten, daß partielles Erben zu einem Unterschied von Vererbungs- und Schnittstellenhierarchie der Bibliothek führt. Beide Formen von Hierarchien werden in den folgenden Abschnitten erläutert.

[15] [Cook 92]:„Yet there is a growing consensus that inheritance is a „producer's mechanism“ [Meyer 90] that has little to do with client's use of classes.“

4.2.1 Vererbungshierarchie

Die Vererbungsstruktur einer Bibliothek ist geeignet, die Klassen zu dokumentieren und den Benutzern beim Verständnis der Bibliothek zu helfen[16] [Cook 92]. In einer ohne Verwendung partiellen Erbens entwickelten Bibliothek könnte der Benutzer mit Hilfe der Vererbungsbeziehungen sehr leicht feststellen, welche Methoden eine Klasse anbietet. In einer solchen Bibliothek bietet eine Klasse folgende Methoden an:

- Alle in der Klasse selbst definierten öffentlichen Methoden.
- Alle Methoden, die transitiv von von den Oberklassen der jeweiligen Klasse als öffentlich deklariert wurden.

Bei Verzicht auf partielles Erben ist die Vererbungsbeziehung also für den Bibliotheksentwickler genau wie für den Konsumenten ein strukturierendes Hilfsmittel. Der Nutzen der Vererbungsbeziehung für den Benutzer der Bibliothek ist noch größer, wenn die Vererbungsbeziehungen in der Bibliothek nicht nur zum Zwecke der Wiederverwendung von Implementierungen (Abschnitt 3.2.4.1), sondern mit dem Ziel der Spezialisierung und Verfeinerung von Konzepten (Abschnitt 3.2.4.2) eingesetzt werden.

Die Verwendung partiellen Erbens bleibt indes für den Benutzer der Bibliothek nicht ohne negative Folgen, gerade weil er sich in erster Linie für das Protokoll einer Klasse interessiert. Der Benutzer muß wissen, welche Methoden in diesem Protokoll enthalten sind. Im Falle partiellen Erbens kann der Benutzer die Schnittstelle einer Klasse nicht durch ein transitives Zurückverfolgen der Vererbungsbeziehungen feststellen. Festzustellen, welche Methoden von einer Klasse angeboten werden, gerät zu einem komplexen Algorithmus, bei dem eine Orientierung an der Vererbungshierarchie nur bedingt möglich ist[17] [Cook 92]. Partielles Erben trägt also selbst wesentlich dazu bei, die Vererbung zu einem „Produzentenmechanismus“ werden zu lassen, da die Vorteile der Vererbung für den Konsumenten ausgeschaltet werden.

4.2.2 Schnittstellenhierarchie

Neben dem Begriff der Vererbungshierarchie steht in Sprachen, die partielles Erben zulassen, noch der Begriff der Protokoll- oder Schnittstellenhierarchie[18]. Unter einer Schnittstellenhierarchie wird die logische Organisation aller Schnittstellen einer Bibliothek verstanden. Unter der Schnittstelle einer Klasse versteht man die gesamte Menge der von Exemplaren dieser Klasse ausführbaren Methoden, also die von der Klasse angebotene Funktionalität. Die Schnittstellenhierarchie ist eine partielle Ordnung bezüglich der Konformität, die gemeinsame Teile von Schnittstellen in der Menge der Klassen einer Bibliothek aufzeigt [Cook 92; Cardelli 84; Cardelli, Mitchell 89; Black et al. 87]:

[16] [Cook 92]: „The inheritance structure is used to document the classes and is presented to users as an aid to understanding the library.“

[17] [Cook 92] weist darauf hin, daß sich besonders die Untersuchung der in abstrakten Klassen teilweise implementierten Methoden durch partielles Erben verkompliziert. Gerade abstrakte Klasse können aber durch ihr Angebot einer gemeinsamen Sicht auf komplizierte Objekte bei einem Verzicht auf partielles Erben eine Hilfe für den Benutzer sein, so daß die Schwierigkeiten, eine Schnittstelle zu erkennen, in diesem Bereich besonders schwerwiegend sind.

[18] Bei einem Verzicht auf partielles Erben sind Vererbungs- und Schnittstellenhierarchie identisch.

Konformität:

> Eine Schnittstelle B ist zu einer Schnittstelle A konform, wenn jedes Objekt, das die Schnittstelle B anbietet, auch die Schnittstelle von A anbietet. Die Menge der die Schnittstelle B anbietenden Objekte ist also eine Teilmenge der die Schnittstelle A anbietenden Objekte. Betrachtet man die Schnittstellen der beiden Klassen als Mengen von angebotenen Methoden, so stellt man fest, daß die speziellere Klasse B in der Regel über eine größere Menge von Methoden verfügt. Es sind jedoch gegenüber der Menge der Methoden von A keine Methoden entfernt worden. Die Menge der Methoden von A ist also eine Teilmenge der Menge der Methoden von B. Dies bedeutet, daß ein Objekt vom Typ B überall dort verwendet werden kann, wo ein Objekt vom Typ A erwartet wird.

Im Rahmen einer Untersuchung der Smalltalk-Bibliothek wurde ein Algorithmus entworfen und implementiert, der aus den Klassen einer Bibliothek eine Schnittstellenhierarchie extrahiert [Cook 92]. Abbildung 4.5 veranschaulicht, wie aus den einzelnen Schnittstellen der Klassen eine Protokollhierarchie gewonnen werden kann [Cook 92]. Im linken Teil des Bildes sind die von den drei Klassen A, B und C angebotenen Methoden aufgeführt. Der rechte Teil des Bildes zeigt die sich aus diesen einzelnen Schnittstellen ergebende Protokollhierarchie. In der Protokollhierarchie sind nur die jeweils hinzukommenden Methoden einer Klasse angegeben. Für zwei Schnittstellen, die einen Teil ihres Protokolls gemeinsam haben, fügt der Algorithmus eine abstrakte Schnittstelle ein. In der Abbildung sind dies die Schnittstellen X und Y.

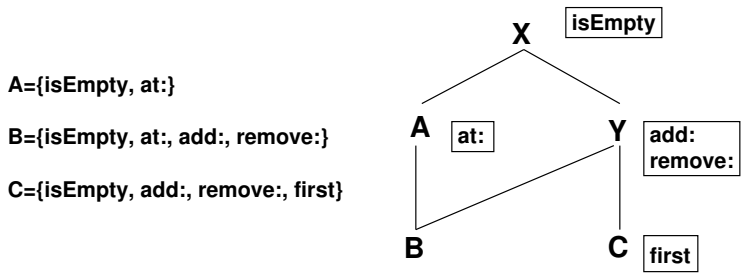

Abbildung 4.5: Entwicklung der Protokollhierarchie (nach [Cook 92])

Eine partiell erbende Klasse mag zwar eine größere Menge von angebotenen Methoden aufweisen, doch die Bedingung, daß die Menge der Methoden der Oberklasse eine Teilmenge der Menge der Methoden der Subklasse sein muß, ist beim partiellen Erben verletzt. Partielles Erben zerstört also den Zusammenhang von Vererbung und Konformität der Schnittstellen.

4.2.3 Ergebnis

Zusammenfassend ist festzustellen, daß in Bibliotheken, die unter Verwendung partiellen Erbens entwickelt werden, die Hierarchie der Schnittstellen der Klassen unabhängig von der Vererbungshierarchie ist. Diese Unabhängigkeit führt dazu, den Nutzen von Vererbung auf die Anbieter der Bibliothek zu beschränken, da der Benutzer der Bibliothek die ihn interessierende Schnittstellenhierarchie nur unter Anwendung eines komplexen Algorithmus entwickeln

kann. Die Schnittstelle einer Klasse ist nicht mehr auf den ersten Blick zu erkennen. Oberklassen bieten dem Benutzer dann keine einheitliche Sicht auf die Objekte komplexer Klassen.

Um beim Design einer Bibliothek diese Nachteile zu vermeiden, sollten die Unterschiede zwischen Vererbungs- und Schnittstellenhierarchie minimiert werden. Beim Entwurf der Klassen sollte deshalb darauf geachtet werden, daß Methoden nicht zu hoch in der Bibliothek angesiedelt werden. Methoden dürfen erst in Klassen eingeführt werden, deren Nachfolger alle ihre Methoden implementieren können. Die Ausführungen dieser Arbeit zeigen, daß Vererbung kein „Produzentenmechanismus“ ist. Vererbung wird erst durch ihre unvorsichtige Anwendung vom Entwickler der Bibliothek zu einem „Produzentenmechanismus“ degradiert.

4.3 Smalltalk

Bei der Beschreibung der Bibliothek NIHCL wurde bereits erwähnt, daß die Bibliothek der Sprache Smalltalk das Vorbild für NIHCL ist. Um die beim Design beider Bibliotheken getroffenen Entscheidungen besser nachvollziehen zu können, beschreiben die nachfolgenden Abschnitte kurz die Eigenschaften der Sprachen Smalltalk und ihrer Bibliothek. Da das Entfernen von Methoden aus der Schnittstelle einer Klasse bei der Gestaltung der NIHCL eine wesentliche Rolle spielt, wird auch die Bibliothek der Sprache Smalltalk besonders im Hinblick auf diese Problematik untersucht.

4.3.1 Sprache

Die objektorientierte Programmiersprache Smalltalk-80 [Goldberg, Robson 83] ist eine interpretative Sprache. Ohne daß der Benutzer einen Übersetzer aufrufen muß, werden die Methoden von Klassen im Hintergrund übersetzt und bei Bedarf auf einer virtuellen Maschine durch einen Interpretierer (*interpreter*) ausgeführt. Teil des Smalltalk-Systems ist eine graphische Benutzeroberfläche [Goldberg 84]. Innerhalb dieser Oberfläche kann der Benutzer Objekte interaktiv mit Hilfe des Mauszeigers manipulieren.

Der Objektbegriff von Smalltalk ist allgemeiner als in anderen objektorientierten Sprachen. So kann ein Objekt in Smalltalk sowohl eine Kapsel aus Datenstruktur und Funktionalität als auch ein Prozeß oder die Beschreibung einer Klasse (*meta class*) sein. Alle Bestandteile eines Programmes sind in Smalltalk somit Objekte [Hoffmann 87].

Smalltalk ist eine untypisierte Sprache. Eine Entscheidung über die Zulässigkeit von Operationen auf Werten erfolgt somit erst zur Laufzeit. Der Versuch, eine Methode aufzurufen, die nicht Teil der Schnittstelle einer Klasse ist, führt zu einem Fehler zur Laufzeit. Auch Smalltalk bietet abstrakte Klassen an. Methoden, deren Funktionalität in einer Klasse noch nicht vollständig implementiert werden kann, werden als subclassResponsibility mit einer vorläufigen Implementierung versehen:

```
size
  self subclassResponsibility
```

Die Methode size kann in Subklassen mit einer anderen Implementierung überschrieben werden. Sollte ein Überschreiben der Methode in der Subklasse unterbleiben, so führt der Aufruf von size durch die Implementierung als subclassResponsibility zu einem Laufzeitfehler:

```
subclassResponsibility
  self error: 'My subclass should have overridden one of my messages.'
```

Ähnlich der Sprache C++ kann auch in Smalltalk eine geerbte Methode aus der Schnittstelle entfernt werden. Nach einer allgemeinen, nicht zwingend vorgeschriebenen Konvention werden hiervon betroffene Methoden wie in C++ als shouldNotImplement überschrieben:

```
capacity
  self shouldNotImplement
```

Die Methode shouldNotImplement ist bereits in der Klasse Object so implementiert, daß ihr Aufruf zur Laufzeit eine Fehlermeldung erzeugt:

```
shouldNotImplement
  self error: 'This message is not appropriate for this object.'
```

Das Löschen von Methoden aus der Schnittstelle in Smalltalk geschieht also so ähnlich wie in C++. In C++ wurden zu löschende Methoden in private Methoden umgewandelt. Versuche, entsprechende Methoden trotzdem aufzurufen, können somit bereits im Zuge der statischen Überprüfung zur Übersetzungszeit verhindert werden. In Smalltalk kann die Methode zwar noch zur Laufzeit aufgerufen werden, da eine statische Überprüfung fehlt, doch die Methode löst eine Fehlermeldung aus. Es kann also auch in Smalltalk von der Möglichkeit partiellen Erbens gesprochen werden. Vergleicht man die beiden Varianten, geerbte Methoden aus der Schnittstelle zu entfernen, so hat das in Smalltalk verwendete Verfahren den Nachteil, die Unmöglichkeit der Anwendung der Methode erst zur Laufzeit des Programms zu entdecken.

4.3.2 Bibliothek

Teil des Smalltalk-System sind zahlreiche bereits vordefinierte Klassen. Da Smalltalk dem Bibliotheksentwickler nur eine Einfachvererbung[19] bietet, sind die Klassen in Form eines Baumes unter der Wurzel Object organisiert. Einen Überblick über die Bibliothek bietet Abbildung 4.6.

Durch die Anwendung partiellen Erbens differieren auch in der Smalltalk-Bibliothek Vererbungs- und Schnittstellenhierarchie. Mit Hilfe des in Abschnitt 4.2 beschriebenen Algorithmus von [Cook 92] kann aus der Vererbungshierarchie die Schnittstellenhierarchie entwickelt werden. Abbildung 4.7 zeigt die mit Hilfe dieses Algorithmus entwickelte Hierarchie der Schnittstellen der Bibliothek. Die in der Abbildung mit kursiver Schrift und eckiger Umrandung abgebildeten Klassen stellen abstraktes Protokoll dar, das im Zuge der Analyse der Bibliothek vom Algorithmus automatisch der Darstellung hinzugefügt worden ist.

4.3.3 Bewertung der Smalltalk-Bibliothek

Ein Vergleich der beiden abgebildeten Hierarchien zeigt so große Unterschiede, daß auch in dieser Bibliothek Vererbung nur dem Produzenten Vorteile bringt. Die gemeinsame Sicht des Konsumenten ist durch die Anwendung des partiellen Erbens wie in der NIHCL zerstört. Eine Hilfe bei der Auswahl des zu verwendenden Behälters bietet die Vererbungshierarchie der Bibliothek nicht. Es gilt also hier dieselbe Kritik, die auch im Abschnitt 4.1.2 bereits gegenüber der NIHCL geäußert wurde.

Da sich die Entwickler der NIHCL an der Bibliothek von Smalltalk orientiert haben, ist es nicht verwunderlich, daß die Unterschiede zwischen Vererbungs- und Schnittstellenhierarchie die Qualität beider Bibliotheken vermindern. Unverständlich ist es jedoch, eine Bibliothek

[19] Eine Erweiterung von Smalltalk mit Mehrfachvererbung beschreibt [Borning 82].

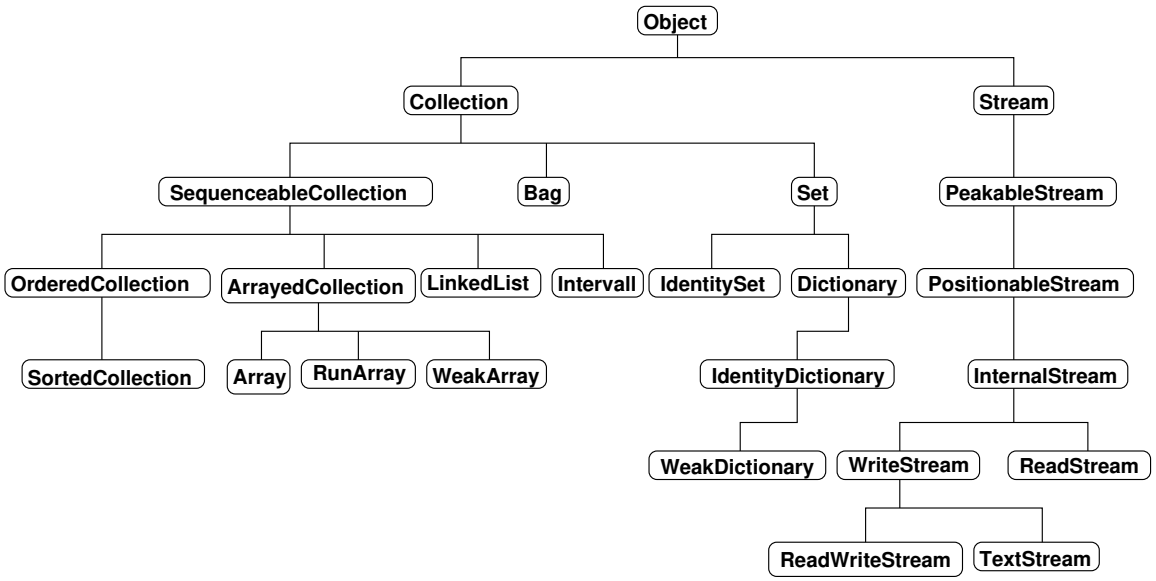

Abbildung 4.6: Ausschnitt der Smalltalk-Klassenbibliothek

in einer Sprache zu entwickeln, die Mehrfachvererbung anbietet und sich gleichzeitig an der Bibliothek einer Sprache zu orientieren, welche diese Möglichkeit nicht anbietet. Bei der Erstellung der Smalltalk-Bibliothek mußte wegen der Spracheigenschaften von Smalltalk auf die Verwendung von Mehrfachvererbung verzichtet werden. Bei der Entwicklung von NIHCL hätte von dieser zusätzlichen Freiheit bei der Modellierung von Beziehungen zwischen Klassen jedoch Gebrauch gemacht werden können.

4.4 Eiffel

Die in der Programmiersprache Eiffel [Meyer 92; Switzer 92] entwickelten Klassenbibliotheken zeichnen sich durch eine große Anzahl angebotener Klassen und starke Strukturierung aus. Die Entwürfe der unterschiedlichen Anbieter orientieren sich alle stark an dem ursprünglichen Entwurf, wie er in [Meyer 94b] beschrieben ist. Abschnitt 4.4.1 beschreibt kurz die wesentlichen Eigenschaften der Sprache, der darauf folgende Abschnitt 4.4.2 die Struktur der Behälterklassen-Hierarchie der Klassenbibliothek von Eiffel. Abschnitt 4.5 faßt die Ergebnisse zusammen.

4.4.1 Sprache

Der Entwurf der Programmiersprache Eiffel zielt direkt auf die sprachliche Unterstützung qualitätssichernder objektorientierter Bibliotheksentwicklung. Der Zielsetzung der Wiederverwendbarkeit und Erweiterbarkeit von Softwarebibliotheken entsprechend, bietet Eiffel Mehrfachvererbung, wiederholte Vererbung und eine Form des gebundenen parametrischen Polymorphismus für Klassendefinitionen. Die Entwicklung zuverlässiger Systeme wird durch strenge Typisierung und vielfältige Möglichkeiten zur Formulierung von Invarianten sowie

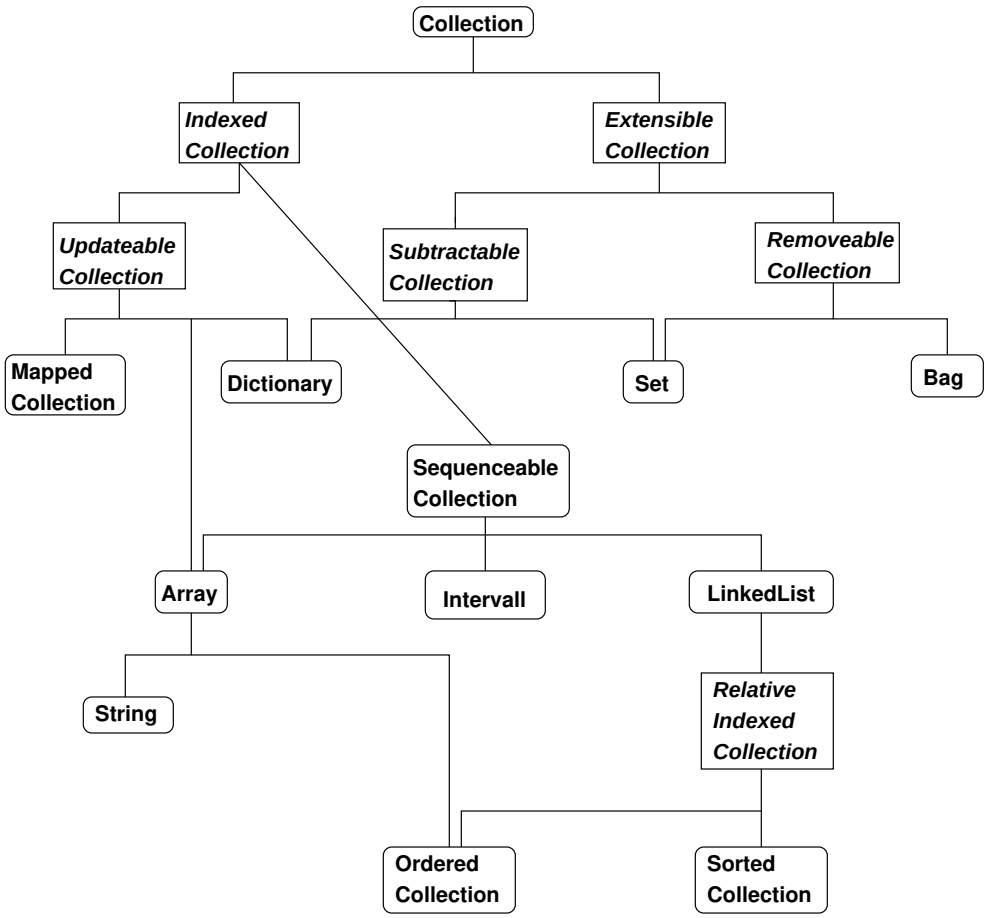

Abbildung 4.7: Schnittstellenhierarchie der Smalltalk-Bibliothek (nach [Cook 92])

Vor- und Nachbedingungen unterstützt (*design by contract* [Meyer 94a, S.23]). Für letztere wird auch der Zugriff auf den Zustand vor Ausführung einer Methode angeboten.

Zur weiteren Flexibilisierung des Vererbungsmechanismus und zur Auflösung von Namenskonflikten können Methoden umbenannt (rename) werden. Ebenfalls der Flexibilisierung dient die Möglichkeit, bei der Redefinition (redefine) von Methoden auch eine Veränderung der Signaturen vorzunehmen. Zusätzlich ermöglicht Eiffel, Methoden auch vollständig aus der Schnittstelle einer Klasse zu entfernen (undefine). Der Einsatz der Vererbung zur Abstraktion und Klassifizierung wird durch die Möglichkeit unterstützt, abstrakte Klassen und Methoden zu definieren.

Während C++ mit Hilfe der Schlüsselworte public, protected und private nur die Unterscheidung zuließ, eine Methode für alle Benutzer anzubieten oder ihre Zugreifbarkeit auf die Klasse oder die Klasse und ihre Erben zu beschränken, bietet Eiffel eine flexiblere Zugriffskontrolle an. Die Kapselung von Methoden gegenüber Klienten einer Klasse kann für jede Methode explizit durch Angabe der Klassen, denen der Zugriff erlaubt ist, festgelegt werden. Dabei ist es sowohl möglich, allen anderen Klassen den Zugriff zu erlauben (entspricht public in C++), als auch den Zugriff aller anderen Klassen zu unterbinden (entspricht private in

C++). Werden keine Angaben gemacht, so sind Methoden für alle Klienten zugreifbar. Der so festgelegte Exportstatus einer Methode wird prinzipiell von Subklassen übernommen. Er kann aber von diesen beliebig verändert werden.

Das Typsystem basiert auf der Klassendefinition. Jeder Typ wird direkt oder indirekt von einer Klasse abgeleitet. Auf den Typen ist eine der Subtyprelation ähnliche Konformitätsrelation definiert, welche die Konsistenzbedingung für z.B. Zuweisungen, Parameterübergaben und Redefinitionen von Methodensignaturen darstellt [Meyer 92, S.117ff]. Diese Relation entspricht für nicht parametrisierte Typen der Vererbungsrelation. Für von generischen Klassen beschriebene parametrisierte Typen wird die Konformitätsbeziehung zwischen den durch Instantiierung abgeleiteten Typen betrachtet. Die Konformität dieser Typen ist daher zusätzlich zur Vererbungsbeziehung noch abhängig von den für die Instanttierung verwendeten Typen.

Bei der Redefinition von Methoden lassen die Regeln zur Einhaltung der Konformitätsbeziehung die kovariante Spezialisierung von Parametertypen zu. Aufgrund dessen und wegen der Möglichkeit zur Veränderung des Exportstatus einer Methode benötigt die abschließende Typüberprüfung eine potentiell systemweite Analyse zum Bindungszeitpunkt [Meyer 92, S.364].

Funktionen haben in Eiffel keinen '*first class object*'-Status, d.h. sie können nicht als Aktualparameter oder Rückgabewert auftreten. Ein Metaklassenkonzept existiert ebenfalls nicht. Objekterzeugungsmethoden sind Bestandteil der Klassendefinition und beziehen sich auf eine konkret deklarierte Variable, welche das erzeugte Objekt aufnimmt.

4.4.2 Die Eiffel-Bibliothek

Einen Überblick über die Bibliothek bietet Abbildung 4.8. Die Abbildung zeigt die Massendatentypen-Hierarchie der Eiffel-Klassenbibliothek. Abstrakte Klassen sind in der Abbildung durch einen Stern neben dem Namen der Klasse gekennzeichnet. An einigen Stellen sind, um die Übersichtlichkeit zu erhöhen, nicht alle Erbbeziehungen eingezeichnet. In solchen Fällen wird die fehlende Erbbeziehung nur durch einen kurzen Strich angedeutet.

In der Abbildung zeigt sich eine von der Organisation der bisher betrachteten Bibliotheken abweichende Struktur der Eiffel-Bibliothek. Während die Ableitung der Behälter von Collection den anderen Bibliotheken nachempfunden ist, zeigt sich mit den in anderen Bibliotheken nicht vorhandenen Klassen Box und Traversable ein neues Vorgehen bei der Organisation objektorientierter Bibliotheken. Die Eiffel-Bibliothek teilt die angebotene Funktionalität in 3 Bereiche ein:

- Speicherung: Die unter Box beginnende Hierarchie enthält Funktionalität, die im Zusammenhang mit der Speicherung der Elemente der Behälter steht. Hier finden sich beispielsweise Methoden zur Feststellung der Größe der Behälter, ihrer Kapazität und für die Anpassung der Kapazität verwendete Variablen.

- Zugriff: Unter Collection ist die auf den Zugriff bezogene Funktionalität der Bibliothek angeordnet. In diesem Teil der Hierarchie befinden sich unter anderem Methoden zum Einfügen und Löschen von Elementen sowie zum Suchen von Elementen in den jeweiligen Behältern.

- Traversierung: Protokoll zum Traversieren der Behälter bietet die Hierarchie unter der Klasse Traversable. Bei der Traversierung gibt es ein aktuelles Element des Behälters, auf das mit item zugegriffen werden kann.

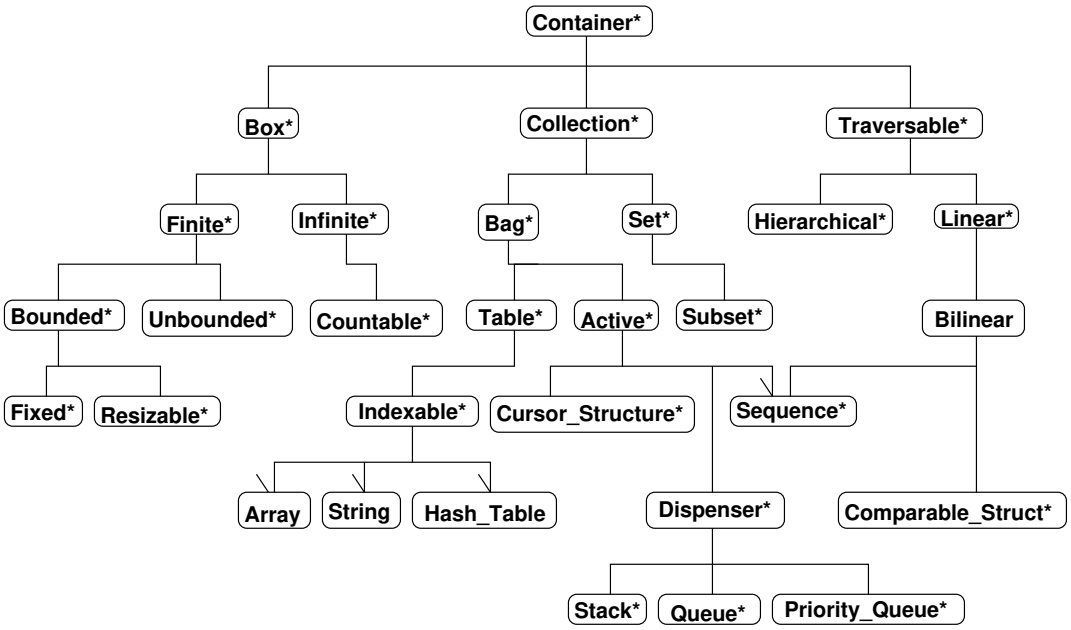

Abbildung 4.8: Ausschnitt der Eiffel-Klassenbibliothek

Der größte Teil der Klassen der Bibliothek ist abstrakt. Dies unterstützt die These, in objektorientierten Bibliotheken könne ein Großteil der Funktionalität entkoppelt von der Repräsentation der Daten in den abstrakten Klassen der Vererbungshierarchie realisiert werden. Klassen, die instantiierbar sein sollen, kombinieren die Funktionalität der 3 Pfade und implementieren die noch abstrakten Methoden in Abhängigkeit von der dann jeweils gewählten konkreten Repräsentationsform der Daten. Die Eiffel-Bibliothek ist damit ein Beispiel für die gegenüber modulorientierten Bibliotheken überlegene Kombinierbarkeit objektorientierter Bibliotheken. Abbildung 4.9 zeigt die Verteilung der Methoden auf die Klassen der Eiffel-Bibliothek.

Um die Qualität der Eiffel-Bibliothek zu beurteilen, muß auch diese Bibliothek daraufhin untersucht werden, ob und in welchem Maß die abstrakten Klassen der Bibliothek eine gemeinsame und einfache Sicht auf das Protokoll der komplizierteren Klassen bieten. Die thematische Aufteilung des Protokolls in 3 Bereiche könnte das Erreichen dieses Ziels erleichtern, da die Klassen der Eiffel-Bibliothek nicht die Spezialisierung ganzer Behälter, sondern lediglich die Spezialisierung von Protokollausschnitten realisieren müssen. Die Gefahr von Konflikten zwischen geerbten und geplanten Eigenschaften einer Subklasse ist durch diese Konstellation geringer als in den anderen Bibliotheken.

Eine Durchbrechung der einheitlichen Sicht der abstrakten Klassen auf die komplexen instantiierbaren Klassen für die Benutzer ist in Eiffel durch die Anwendung der in Abschnitt 4.4.1 beschriebenen Möglichkeiten zur Umbenennung von Methoden und zur Veränderung der Sichtbarkeit der Methoden möglich.

Eine Analyse des Quelltextes der Bibliothek zeigt, daß auch in Eiffel ein Vielzahl von Methoden umbenannt (rename) wird. Umbenennungen erfolgen in den Klassen Linear, Indexable, Hash_table und Table sowie zusätzlich noch in den nicht in der Abbildung enthalte-

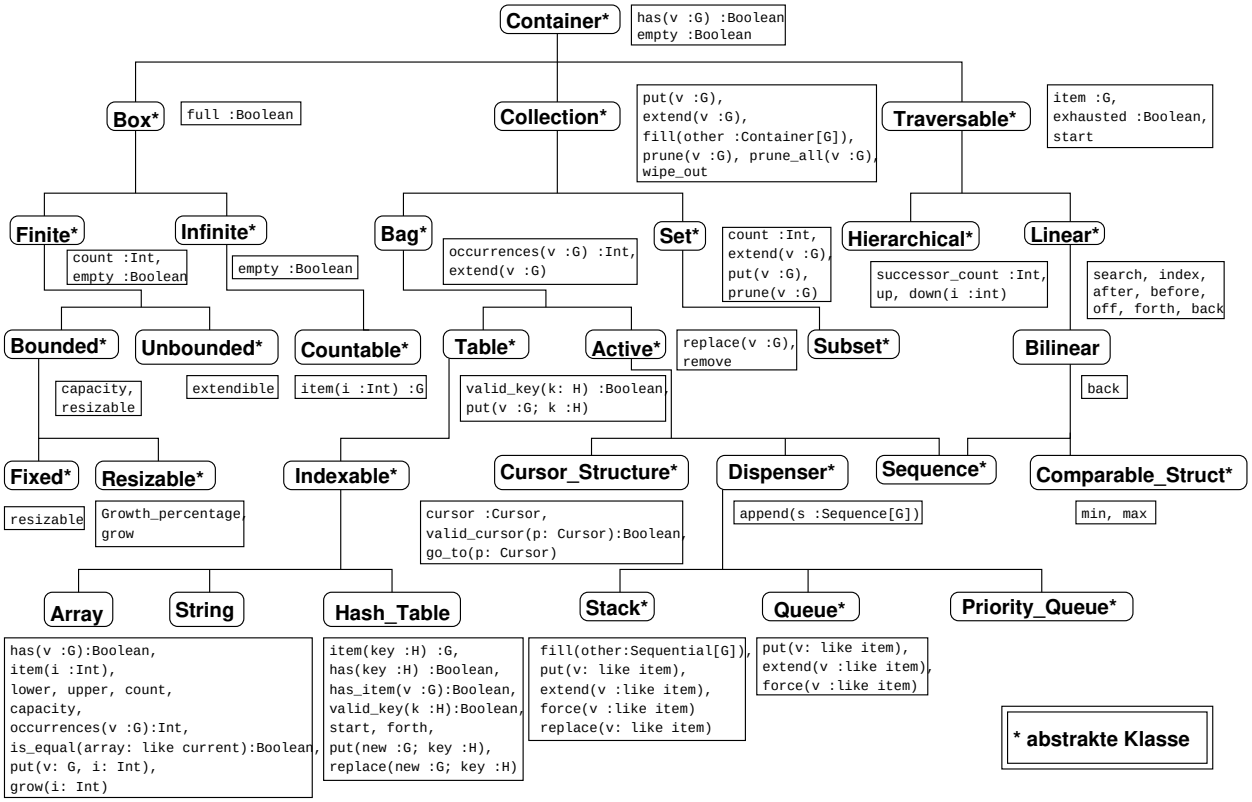

Abbildung 4.9: Verteilung der Methoden der Eiffel-Klassenbibliothek

nen Klassen Arrayed_circular, Arrayed_list, Chain, Dynamic_list, Linked_circular, Two_way_circular, Two_way_list, Fixed_list, Arrayed_tree, Binary_search_tree, Linked_tree, Two_way_tree, Comparable_set, Arrayed_queue, Arrayed_stack, Linked_queue, Linked_stack, Sorted_priority_queue, Heap, Fixed_queue, Fixed_stack, Array2, Sorted_struct, Compact_cursor_tree, Cursor_tree, C_tree_iter und Twc_iter.

Auch von der Möglichkeit, geerbte Methoden zu Löschen (undefine) wird in Eiffel ebenfalls in vielen Klassen Gebrauch gemacht. Im einzelnen sind dies die Klassen Seq_string, Active, Arrayed_circular, Arrayed_list, Chain, Circular, Dynamic_chain, Dynamic_circular, Dynamic_list, Linked_circular, List, Sorted_list, Sorted_two_way_list, Two_way_circular, Fixed_list, Part_sorted_two_way_list, Arrayed_tree, Linked_tree, Two_way_tree, Comparable_set, Linked_set, Sorted_set, Part_sorted_set, Arrayed_queue, Arrayed_stack, Linked_queue, Linked_stack, Sorted_priority_queue, Fixed_queue, Fixed_stack, Array2 und Sorted_struct.

Abbildung 4.10 zeigt das Entfernen einer Methode aus der Schnittstelle am Beispiel der Klasse Table. In Collection (hier nur verkürzt dargestellt) wird die Methode put mit einem Parameter eingeführt:

```
deferred class COLLECTION [G] inherit
    CONTAINER [G]
feature -- Element change
    put, extend (v: G) is  -- Ensure that structure includes 'v'.
        require extendible: extendible deferred
```

```
        ensure item_inserted: has (v) end;
    end -- class COLLECTION
```

Über die Eigenschaften von Table sagt die im Quelltext der Klasse enthaltene Beschreibung jedoch, daß der Zugriff auf die Elemente einer Table mit Hilfe eines Index erfolgt. Die Eigenschaft, Elementen einen Index zuzuordnen, ist jedoch nicht konform mit der Möglichkeit, ein Element ohne Angabe eines solchen Index in die Table einfügen zu können. Die Methode put wird daher in Table (ebenfalls verkürzt dargestellt) umbenannt (in bag_put) und eine neue Methode zum Einfügen unter Verwendung des alten Namens put mit 2 Parametern eingeführt. Das aus der Umbenennung des geerbten put entstandene bag_put wird so überschrieben, daß sein Aufruf keinerlei Aktion zur Folge hat. Gleichzeitig wird die Zugriffskontrolle bezüglich der Methode bag_put so geändert, daß der Aufruf der Methode für Objekte anderer Klassen nicht mehr möglich ist:

```
deferred class TABLE [G, H] inherit
    BAG [G] -- Containers whose items are accessible through keys
      rename put as bag_put end;
feature -- Element change
    put (v: G; k: H) is -- Associate value 'v' with key 'k'.
      require valid_key: valid_key (k) deferred
      ensure insertion_done: item (k) = v end;
feature NONE -- Inapplicable
    bag_put (v: G) is do end
end -- class TABLE
```

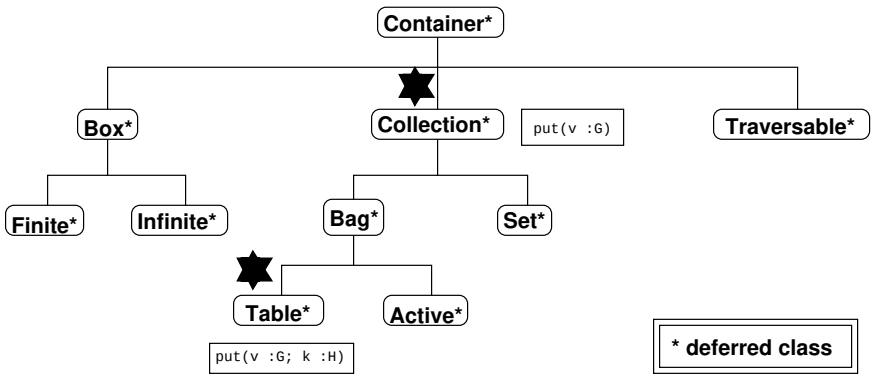

Abbildung 4.10: Beispiel für das Entfernen von Methoden in der Eiffel-Bibliothek

4.5 Fazit

In Kapitel 2 wurde Übersichtlichkeit als wesentliches Qualitätsmerkmal für Softwarebibliotheken neben den allgemeinen Merkmalen wie Korrektheit herausgestellt. Der Vergleich der vorherrschenden Grundstrukturen für Softwarebibliotheken in Kapitel 3 zeigt, daß die Verwendung einer objektorientierten Bibliothek die Entwicklung einer übersichtlichen Bibliothek begünstigt. Zusätzlich bietet Objektorientierung die Möglichkeit der Kombination von Funktionalität durch die Entkoppelung der Funktionalität von der Repräsentation der Daten.

In diesem Kapitel wurden C++, Smalltalk und Eiffel daraufhin verglichen, welche Eigenschaften der Sprachen Auswirkungen auf die Gestaltung der jeweiligen Behälterbibliothek haben. Dabei ist festzustellen, daß alle untersuchten Sprachen über die Möglichkeit partiellen Erbens verfügen. Bei allen untersuchten Bibliotheken dieser Sprachen wurde von dieser Möglichkeit ausgiebig Gebrauch gemacht. Somit ist partielles Erben eine wesentliche Spracheigenschaft in Bezug auf die Gestaltung von Bibliotheken.

Das mit partiellem Erben verbundene Entfernen von Methoden aus der Schnittstelle erbender Klassen geschieht entweder mit Hilfe von Schlüsselworten (rename oder undefine in Eiffel, private in C++) oder durch Überschreiben mit einer durch Konvention festgelegten Methode, deren Aufruf eine Fehlermeldung erzeugt. Durch diese Verfahren partiellen Erbens ergeben sich erhebliche Unterschiede zwischen der Vererbungs- und der Schnittstellenhierarchie der Bibliotheken. Die Fähigkeit abstrakter Klassen, eine einfache und gemeinsame Sicht auf verschiedene komplexe Klassen anzubieten, wird in allen untersuchten Bibliotheken zerstört, so daß die Bibliotheken die in Kapitel 2 genannten Qualitätsanforderungen bezüglich der Übersichtlichkeit nicht erfüllen.

Entgegen der in der Literatur vertretenen These, Vererbung sei nur ein Produzentenmechanismus [Meyer 90] und die Durchbrechung des Prinzips der Spezialisierung nötig [Coad, Nicola 94], verfolgt diese Arbeit das Ziel, Vererbung zum Nutzen von Produzenten und Konsumenten der Bibliothek einzusetzen. Die im Rahmen der Diplomarbeit verwendete objektorientierte Programmiersprache TooL bietet daher konsequenterweise auch keinerlei Unterstützung partiellen Erbens an. In der TooL-Behälterklassenbibliothek sind daher Vererbungs- und Schnittstellenhierarchie identisch, so daß jede Klasse eine einfache und gemeinsame Sicht auf die Schnittstellen ihrer Subklassen anbietet.

Kapitel 5

Bibliotheksentwurf in TL

Die in den vorhergehenden Kapiteln herausgearbeiteten Qualitätsmerkmale für Softwarebibliotheken im allgemeinen und Bibliotheken für Massendatentypen im besonderen (Kapitel 2), sowie die in Abschnitt 3.1 dargelegten Schwächen einer modulorientiert entworfenen Massendatenbibliothek führen zur Bevorzugung eines objektorientierten Entwurfsansatzes. Dies wirft die Frage auf, inwieweit sich ein solcher Ansatz durch einen objektorientierten Programmierstil in der Programmiersprache TL, der hochsprachlichen Schnittstelle des Tycoon-Systems umsetzen läßt. Dabei treten einige grundlegende Schwierigkeiten auf. Diese sollen hier anhand einiger Beispiele zusammengefaßt werden. Der direkt anschließende Abschnitt skizziert aber zunächst kurz die Eigenschaften von TL.

5.1 Die Programmiersprache TL

Dieser Abschnitt stellt einige der für die weiteren Ausführungen dieses Kapitels benötigten Sprachkonzepte von TL vor. Eine ausführliche Beschreibung findet sich in [Matthes et al. 94; Matthes 92].

TL ist eine strikt typisierte, funktionale Sprache, die aber auch imperative Eigenschaften wie destruktive Zuweisungen und Seiteneffekte in Funktionen beinhaltet. Der imperativen Seite ist z.B. die explizite Deklaration von veränderbaren Variablen mit Hilfe des Schlüsselwortes **var** zuzurechnen. TL definiert eine strukturelle Subtypbeziehung durch statische Überprüfung induktiv formulierter Typregeln. Funktionen und Typen sind in TL gleichberechtigte Sprachkomponenten (*first class status*), das heißt, sie können beispielsweise auf Variablen zugewiesen oder auf Funktionsparametern übergeben werden. TL unterstützt Subtyppolymorphismus und gebunden parametrischen Polymorphismus. Weiterhin bietet die Sprache benutzerdefinierte Typoperatoren zur Formulierung generischer Typausdrücke. Die Typhierarchie der nicht parametrisierten Typen wird durch den Supertyp aller Typen **Ok** nach oben hin und durch den Subtyp aller Typen **Nok** nach unten hin abgeschlossen. Zur Modularisierung der Programme wird die Aufteilung in Schnittstelle (**interface**) und Implementierung (**module**) unterstützt.

Zur Datenmodellierung bietet TL vordefinierte Typ- und Wertkonstruktoren für Tupel, Tupel mit Varianten, Records und Arrays. Auch für Funktionen stehen entsprechende Operatoren zur Verfügung. Rekursive Typ- oder Wertbindungen erfolgen explizit mit Hilfe entsprechender Schlüsselworte (**Rec**, **rec**). Der aggregierende Typkonstruktor **Tupel** beispielsweise ähnelt einem *Record* in Pascal oder einer *Structure* in C. Seine Felder werden durch eine geordnete Folge von Signaturen beschrieben.

```
Let IntegerList = Tuple RepT <:Ok  first():Int end
```

Der Typ IntegerList ist ein Tupel mit einer Typsignatur RepT und einer Funktion first. Typsignaturen in Tupeltypen sind die Grundlage für die Darstellung abstrakter Datentypen in TL. Der Typ RepT ist (semi)abstrakt. Über einen Wert dieses Typs sind nur die Informationen bekannt, welche sich aus der deklarierten Subtypbeziehung ergeben.

Zur Unterstützung inkrementeller Softwareentwicklung bietet TL die Konstrukte **Repeat** und **open**, mit deren Hilfe Typdefinitionen bzw. Wertbindung wiederholt werden können. So ist beispielsweise die Definition eines Typs Student unter Verwendung des Typs Person sowie die Erzeugung der korrespondierenden Werte mit Hilfe dieser Schlüsselworte möglich.

```
Let Person  = Tuple name :String age :Int end
Let Student = Tuple Repeat Person semester :Int end

let paul :Person = tuple let name = "Paul" let age = 27 end
let paulAsStudent : Student = tuple open paul let semester = 6 end
```

Der Aufruf von **Repeat** führt zur Wiederholung der Signaturdefinitionen des Typs Person, während mit **open** die Wertbindungen der Feldbezeichner des Tupelwertes paul unter Erzeugung gleichnamiger Bezeichner in den Tupelwert paulAsStudent übernommen werden.

Für die Beschreibung weiterer Spracheigenschaften und Ausdrucksmittel von TL sei auf die bereits zu Anfang des Abschnitts genannten Quellen verwiesen.

5.2 Beispiele

Soll der objektorientierte Programmierstil in TL etabliert werden, so scheint es zunächst naheliegend zu sein, die Klassen durch abstrakte Datentypen zu beschreiben. Auf diese Weise wäre die an der Schnittstelle des abstrakten Datentyps angebotene Funktionalität als Protokoll des Objekts aufzufassen.

5.2.1 Subtypbeziehung zwischen abstrakten Datentypen

Zur Beschreibung des durch eine Klasse definierten Objekts, muß an der Schnittstelle des die Klasse modellierenden abstrakten Datentyps ein Objekttyp angeboten werden. In dieser Konstellation kommt es zum Konflikt zwischen zwei Zielvorstellungen. Einerseits soll bei der Definition dieses Types die eigentliche Implementierung zumindest teilweise versteckt werden, was die Definition des Objekttyps als abstrakte Typvariable begründet. Andererseits ist es aber beabsichtigt, weitere Klassen zu formulieren, deren Objekte als Subtypen der ersten Klasse beschrieben sind. Ein solcher Subtyp kann zwar in der Schnittstelle noch definiert werden, die Implementierung scheitert jedoch an der Kapselung der anderen Klasse. Der Supertyp ist nur über die Schnittstelle zugänglich, wo er als abstrakte Typvariable deklariert wurde. Bei der Implementierung des ebenfalls abstrakt definierten Subtyps werden nun Informationen über die Beschaffenheit des Supertyps benötigt, um die Subtypbeziehung einhalten zu können. Dies bedeutet aber eine Durchbrechung der Kapselung, die bei der Definition des Supertyps durchgeführt wurde. Formal scheitert die Definition des Subtyps an der von TL verlangten Namensäquivalenz für abstrakte Typvariablen.

In Abbildung 5.1 wird die Situation mit Hilfe von Tupeltypen und ihren Werten veranschaulicht, welche sich in TL hinsichtlich der Kapselung genauso verhalten wie Schnittstelle

```
Let PointD1 = Tuple T1 <: Ok end;

let pointD1 :PointD1 = tuple Let T1 = Tuple x :Int end end;

Let PointD2 = Tuple T2 <: PointD1.T1 end;

let b:PointD2 = tuple Let T2 <:pointD1.T1 = Tuple  x,y :Int end end;
```

Abbildung 5.1: Subtypisierung von abstrakten Datentypen

und Modul. Der Typ PointD1 definiert einen in ein Tupel gekapselten abstrakten Typ T1. Die Variable pointD1 hält einen Wert des Typs PointD1 mit dem als Tupeltyp implementierten Typ T1. Typ PointD2 fordert nun einen Subtyp T2 des abstrakten Typs PointD1.T. Die Implementierung des Typs T2 wird aber aus den oben genannten Gründen vom Typüberprüfungsalgorithmus von TL zurückgewiesen.

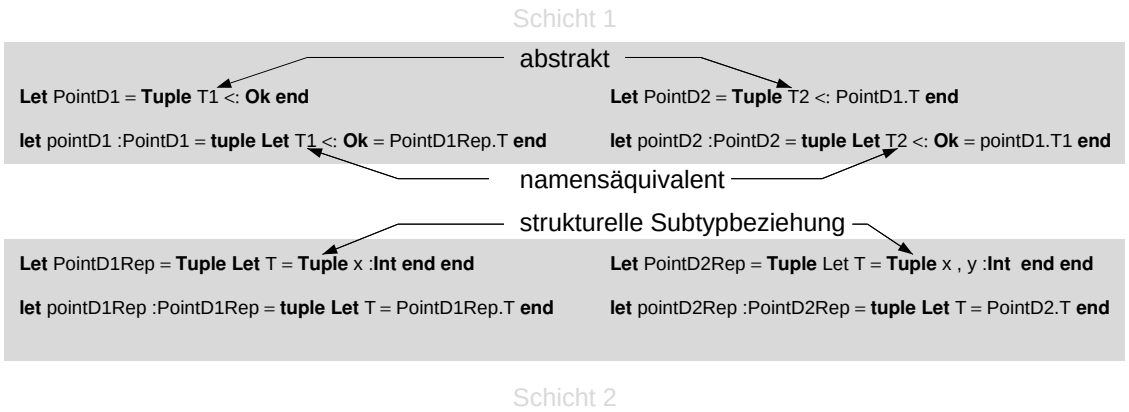

Abbildung 5.2: Schichtung von Spezifikation und Repräsentation

Eine Lösung des Problems ist möglich [Wetzel 94, S.101], indem eine zweite Schicht von Schnittstellen eingeführt wird, so daß jede Klasse schließlich durch zwei Schnittstellen und die entsprechenden Implementationen repräsentiert wird Die obere Schnittstelle dient nur dazu, den Objekttyp zu kapseln und gleichzeitig eine Subtypbeziehung zwischen den Objekttypen verschiedener Schnittstellen zuzulassen. Dies wird erreicht, indem die Objekttypen dieser ersten Schicht in den Implementationsmodulen gleichgesetzt werden. Damit wird nach außen hin die Subtypbeziehung aufrechterhalten. Die erste Schicht entspricht in Abbildung 5.2 den Typen PointD1 und PointD2 sowie den zugehörigen Variablen.

Die Schnittstelle der zweiten Schicht legt nun den eigentlichen Objekttyp offen. Dadurch ist auf dieser Schicht die Kapselung kein Hindernis mehr für die Formulierung von Subtypbeziehungen zwischen den Objekttypen. In Abbildung 5.2 findet man die zur zweiten Schicht korrespondierenden Typen PointD1Rep und PointD2Rep. Da nun nach außen hin über die erste Schicht andere Objekttypen angeboten werden als die letztlich für die Zustandsrepräsentation notwendigen Objekttypen der zweiten Schicht, muß bei der Implementation des Protokolls der

```
module superClass
  import
  export
  Let T <:Ok = Tuple
    var size() :Int
    var empty() :Bool
    end
  let create() :T =
    tuple
      let var size() :Int =
        raise exception 'virtuelle Methode'
      let var empty() :Bool = size() == 0
    end
end;
```

```
module subClass
  import superClass
  export
    Let T <:Ok = Tuple
       Repeat superClass.T
      end
  let create() :T =
    begin
      let super = superClass.create()
      let result =
        tuple
          open super
        end
      result.size := fun() :Int ...
      result
    end
end;
```

Abbildung 5.3: Virtuelle Methoden I.

ersten Schicht die Abbildung auf diese Repräsentationstypen geleistet werden. Insgesamt wird der Klassenentwurf und die Verwaltung der Objekte einer Klasse durch diese Konstruktion komplizierter.

5.2.2 Späte Bindung

Die sogenannte späte Bindung ist für den objektorientierten Entwurf von entscheidender Bedeutung, insbesondere für die Wiederverwendung von Implementierungen. Durch die späte Bindung wird es möglich, gemeinsames Protokoll in abstrakten Oberklassen zu spezifizieren. Dieses Protokoll kann dann durch die Erben der Klasse durch Implementierung der abstrakten Methoden individuell angepaßt und damit wiederverwendet werden.

Eine weitere Möglichkeit ist die Optimierung von Methoden in Subklassen. Durch Redefinition der Methode unter Ausnutzung genauerer Informationen in der Subklasse kann eine Methode optimiert werden. Das abstrakte Protokoll der Oberklasse bleibt unberührt.

Die späte Bindung sollte deshalb bei der Umsetzung eines objektorientierten Programmierstils in TL auch abgebildet werden. Um die Redefinition von Methoden zu ermöglichen, werden diese als auf modifizierbare Variablen zugewiesene Funktionen modelliert (Abbildung 5.3). Damit überhaupt die Bindungen der Oberklasse genutzt werden können, muß ein Objekt dieser Klasse explizit erzeugt werden. Die Bindungen können dann mit Hilfe des TL-Konstruktes **open** in das Subklassenobjekt importiert werden.

In Abbildung 5.3 wird im Modul superClass die Methode size als virtuelle Methode zunächst nicht implementiert. Sie wird aber von Methode empty genutzt, welche auf diese Weise bereits in der Oberklasse definiert werden kann. Die Definition der Methode size im die Subklasse modellierenden Modul subClass führt aber nicht zu dem gewünschten Ergebnis, da durch *open* neue Variablen angelegt werden. Es bleibt für die Methode empty die Bindung an

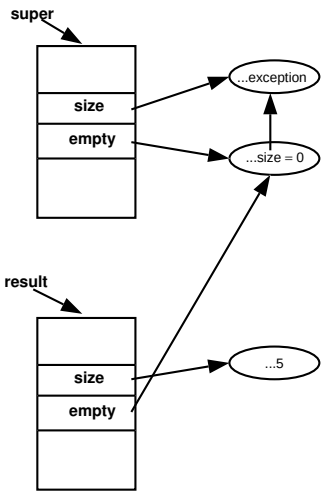

Abbildung 5.4: Fortbestand der alten Bindung

den alten, eine Ausnahme auslösenden Funktionsrumpf erhalten. Die Bindungsverhältnisse werden noch einmal in Abbildung 5.4 skizziert.

Die Zuweisung der neuen Implementierung von size ist aber erfolgreich, wenn sie direkt nach Erzeugung des Objektes der Oberklasse auf die Funktionsvariable dieses Objekts erfolgt. Alle Bindungen die innerhalb des Objektes der Oberklasse an die Funktionsvariable bestehen, übernehmen jetzt die neue Implementierung. Problematisch wird es allerdings, wenn die Bindungen bereits aus einem Vorgängerobjekt des Objektes der Oberklasse stammen. Die Variablen dieses Vorgängers sind nicht mehr erreichbar. Alle Methoden die Bindungen an diese Variablen enthalten, müßten dann ebenfalls überschrieben werden.

5.2.2.1 Ausnutzung der dynamischen Parameterbindung

Eine Vereinfachung kann jedoch durch Ausnutzung der dynamischen Bindung von Funktionsparametern erreicht werden. Dazu ist es notwendig, einen Self-Parameter einzuführen. Statt nun bei der Implementierung einer Funktion direkt auf die anderen Funktionen oder Variablen des Objekts zuzugreifen, geht man den indirekten Weg über den Self-Parameter der zu implementierenden Funktion. Beim Aufruf einer Funktion mit Self-Parameter muß auf diesem das Empfängerobjekt übergeben werden: obj.empty(obj). Auf diese Weise wird sichergestellt, daß die aktuellen Bindungen des Objekts berücksichtigt werden.

Als Beispiel dient Abbildung 5.5 in der nun die abgewandelte empty-Methode zu sehen ist. Sie hat den angekündigten **Self**-Parameter. In der Implementierung von empty wird nun die size-Methode über den **Self**-Parameter angesprochen.

Durch eine solches Vorgehen entstünden eine Vielzahl von Funktionen mit Parameter vom Objekttyp. Derartige Funktionen werden aber zum Problem, wenn der Objekttyp in der Subklasse spezialisiert wird und eine Redefinition der Funktion erfolgen soll. Die Zuweisung auf die in der Oberklasse definierte Funktionsvariable ist nur möglich, wenn der Typ der zuzuweisenden Funktion Subtyp des Variablentyps ist. Die Spezialisierung des Objekttyps in der Subklasse und damit des Parametertyps verletzt jedoch die in TL gültige Kontravari-

```
module superClass
  import
  export
  Let Rec T <:Ok = Tuple
    var size(self :T) :Int
    var empty(self :T) :Bool
    end
  let create() :T =
    tuple
      let var size(self :T) :Int =
        raise exception 'virtuelle Methode'
      let var empty(self :T) :Bool = self.size(self) == 0
    end
end;
```

```
module subClass
  import superClass
  export
    Let RecT <:Ok = Tuple
      Repeat superClass.T
    end
  let create() :T =
    begin
      let super = superClass.create()
      let result =
        tuple
          open super
        end
      result.size := fun(self :T) :Int ...
      result
    end
end;
```

Abbildung 5.5: Methoden mit **Self**-Parameter

anzregel für die Subtypbeziehung von Funktionstypen. Ein Verzicht auf die Spezialisierung des Parametertyps würde eine zu starke Restriktion darstellen, da die Information über die zusätzlichen Funktionen des Objekttyps nicht mehr für die Implementation der Funktionen zur Verfügung stünde (alle Zugriffe müßten über den Self-Parameter erfolgen).

Ein Lösungsansatz für diese Problematik ist es, den Self-Parameter nicht über den Objekttyp des jeweiligen Moduls zu definieren, sondern für ihn den Supertyp aller Typen in TL, den Typ :Ok zu wählen. Dieser könnte dann innerhalb des Funktionsrumpfes durch Typumwandlung auf den Objekttyp des Moduls spezialisiert werden. Dadurch wäre aber der Vorteil der statischen Typisierung, Laufzeitfehler zu vermeiden, aufgehoben, denn bei Übergabe eines falschen, nicht spezialisierten Objekts würde die fehlerhafte Übergabe nicht mehr zur Übersetzungszeit erkannt. Beim Zugriff auf nicht vorhandene Informationen müßte es dann zwangsläufig zu Laufzeitfehlern kommen.

In Abbildung 5.6 wird diese Technik für die schon bekannten Funktionen empty und size angedeutet. Der Self-Parameter ist nun vom Typ Ok und in der Implementierung der Funktion empty wird die Funktion typecast des Moduls unsafe genutzt, um die Funktion size wieder aufrufen zu können. Im Modul subClass wird für die Realisierung der Funktion size die gleiche Technik benutzt.

Akzeptabel scheint die geschilderte Technik allenfalls für die **Self**-Parameter, welche ja immer mit dem 'Empfängerobjekt' selbst instantiiert würden. Dieser Vorgang wäre möglicherweise durch Einsatz der Syntaxerweiterungsmechanismen von TL automatisierbar, so daß die Gefahr eines Laufzeitfehlers durch falsche Übergabe nicht so groß wäre. Soll auch die Spezialisierung anderer, 'normaler' Objekttypparameter ermöglicht werden, wie dies in manchen objektorientierten Sprachen der Fall ist, so ist die Technik nicht verwendbar. Ohne den Schutz der Syntaxerweiterung ist die Gefahr, daß Typfehler erst zur Laufzeit bemerkt werden, zu groß.

```
module superClass
  import unsafe
  export
  Let T <:Ok = Tuple
    var size(self :Ok) :Int
    var empty(self :Ok) :Bool
    end
  let create() :T =
    tuple
      let var size(self :Ok) :Int =
        raise exception 'virtuelle Methode'
      let var empty(self :Ok) :Bool =
        begin
          let my = unsafe.typecast(self :T)
          my.size(my) == 0
        end
    end
end;
```

```
module subClass
  import superClass
  export
    Let T <:Ok = Tuple
        Repeat superClass.T
      end
  let create() :T =
    begin
      let super = superClass.create()
      let result =
        tuple
          open super
        end
      result.size :=
       fun(self :Ok) :Int
       ...unsafe.typecast(self :T) ...
      result
    end
end;
```

Abbildung 5.6: Methoden mit Self-Parameter und Typanpassung

5.2.2.2 Ausnutzung rekursiver Bindung

Eine weitere Möglichkeit, die späte Bindung der Methoden zu simulieren, besteht darin, statt des **Self**-Parameters eine explizite rekursive Bindung zu verwenden, wie in Abbildung 5.7 zu sehen. Im Modul superClass greift die Funktion empty bei ihrer Implementierung nun über die rekursiv gebundene Variable self auf die Funktion size zu. Der Vorteil liegt in dem von der Reihenfolge der innerhalb des Tupels durchgeführten Deklarationen unabhängigen Zugriff auf diese Deklarationen.

Für die späte Bindung ergibt sich aber kein Unterschied gegenüber der Variante ohne Self-Parameter, die zu Beginn des Abschnitts 5.2.2 vorgestellt wurde. Die Spezialisierung durch eine weitere Funktion erfordert bei der Objekterzeugung die Definition eines neuen Tupelwertes. In diesem können dann die Bindung des Superobjekts mittels **open** importiert werden. Allerdings werden dabei aber neue Funktionsvariablen angelegt, die nur die Wertbindung des Superobjekts übernehmen. Dies bedeutet für das Modul subClass in Abbildung 5.7, daß die Zuweisung der Implementierung von size auf diese neue Funktionsvariable erfolgt. Die vom Superobjekt übernommene empty-Funktion ist aber mit ihrem Aufruf von size weiterhin rekursiv an die Funktionsvariable des Superobjekts gebunden. Daher wird der gewünschte Effekt also nicht erzielt.

Bei der Verwendung von **Self**-Parametern trat dieses Problem nicht auf, da die Bindung über den Parameter immer zum Empfänger des Funktionsaufrufes selbst und damit zu den aus Sicht des Empfängers aktuellsten Variablenbelegungen erfolgte.

```
module superClass
  import
  export
  Let T <:Ok = Tuple
    var size() :Int
    var empty() :Bool
    end
  let create() :T =
    let rec self :T =
    tuple
      let var size() :Int =
        raise exception 'virtuelle Methode'
      let var empty() :Bool = self.size() == 0
    end
end;
```

```
module subClass
  import superClass
  export
    Let T <:Ok = Tuple
      Repeat superClass.T
      capacity() :Int
      end
  let create() :T =
    begin
      let super = superClass.create()
      let result =
        tuple
          open super
        let var capacity() = ...
        end
      result.size := fun() :Int ...
      result
    end
end;
```

Abbildung 5.7: Rekursive Bindung statt **Self**-Parameter

5.3 Folgerungen

Es ist generell möglich die benötigten objektorientierten Konzepte in TL abzubilden. Die Simulation aller Konzepte gleichzeitig läßt aber die Modellierungsvorgänge komplizierter werden. Dadurch wird die Übersichtlichkeit reduziert und die Wahrscheinlichkeit technisch fehlerhafter Modellierung erhöht. Wie weit sich dieses durch Nutzung der Syntaxerweiterungsmechanismen von TL vermeiden läßt, wurde nicht abschließend untersucht. Ein solches Vorgehen würde es aber auch erschweren, spezifisch auf die objektorientierten Konzepte zugeschnittene Optimierungen vorzunehmen. Die oben geschilderten Überlegungen und das Verlangen nach einer besseren Unterstützung der objektorientierten Programmierung im Tycoon-System führten zur Entwicklung der objektorientierten TL-Variante TooL. Welche als Implementierungssprache für die im Rahmen dieser Arbeit entwickelte Behälterklassenbibliothek genutzt wurde. Eine Einführung in diese Sprache und die Vorstellung der Behälterklassenbibliothek sind die Themen der nächsten Kapitel.

Kapitel 6

TooL: Eine objektorientierte Variante von TL

Dieses Kapitel beschreibt die zur Implementierung der im Rahmen dieser Arbeit entwickelten Behälterklassenbibliothek verwendete Programmiersprache TooL [Gawecki, Matthes 95; Gawecki, Matthes 96]. Sie wurde zur besseren Unterstützung objektorientierter Programmierung im Tycoon-System entwickelt.

TooL ist eine objektorientierte Variante der Tycoon Language (TL) und übernimmt einige wesentliche Eigenschaften dieser Sprache. So ist TooL wie auch TL eine strikt typisierte Programmiersprache, die die Einhaltung der Regeln ihres Typsystems durch statische Programmanalyse überprüft. Weiterhin hat man auch in TooL die Möglichkeit zur polymorphen Programmierung durch Verwendung von Typparametern für Methoden und Klassen. Typparameter für Methoden ersetzen dabei in gewisser Weise die in TL vorhandenen Typoperatoren. Im Gegensatz zu vielen anderen objektorientierten Sprachen übernimmt TooL von TL auch die Funktionen höherer Ordnung. Darüber hinaus wird aber in TooL ein streng objektorientierter Ansatz verfolgt. Alle Sprachkomponenten werden als Objekte betrachtet, denen man Botschaften senden kann. Auch Kontrollstrukturen werden auf diese Weise modelliert. So ist while eine Nachricht, die jedes Objekt versteht und die mit Hilfe von Funktionsparametern zur Definition der Abbruchbedingung und des Schleifenrumpfes die erwartete Berechnung auslöst.

```
class Object
...
while(cond :Fun():Bool, statement :Fun():Void) :Void
  {
  cond[] ? {statement[], while(cond, statement)}
  }
...
```

Abbildung 6.1: Kontrollstrukturen als Botschaften

Wie in Abbildung 6.1 zu sehen, wird die Methode while in der Klasse Object definiert. Sie ruft sich rekursiv auf, solange die als Parameter übergebene Funktion cond den Wahrheiswert true liefert. Vor jedem weiteren Aufruf wird die für den Schleifenrumpf stehende Funktion statement ausgeführt.

Die Möglichkeiten zur Datenabstraktion werden in TooL auf die in der Klassendefinition mögliche Aggregation von Methoden und Variablen (Slots) vereinfacht, was der Abstraktion, wie sie durch *records* möglich ist, entspricht. Die weiteren speziellen Eigenschaften von TooL werden nun in den folgenden Abschnitten erläutert.

6.1 Kapselung

Das Prinzip der Kapselung als Hilfsmittel zur Festlegung einer definierten Objektschnittstelle wird in TooL durch die Möglichkeit unterstützt, die Klassendefinition in einen öffentlichen und einen privaten Teil aufzuteilen. Die Teilung wird, wie in Abbildung 6.2 zu sehen, durch die Schlüsselworte **public** und **private** symbolisiert. Zur Abgrenzung von Methoden und Variablen wird zusätzlich noch das Schlüsselwort **methods** benutzt. Die Variablen finden sich jeweils direkt hinter dem den Status beschreibenden Schlüsselwort. Für den Benutzer eines Objektes sind dann nur die Methoden oder Variablen zugreifbar, die in der zugehörigen Klasse als öffentlich ausgezeichnet wurden. Die textuelle Trennung von Methodensignatur und Implementierung der Methode wird dadurch nicht geleistet.

Für den Erben einer Klasse bedeutet die Unterscheidung von öffentlichen und Privaten Definitionen keine Zugriffsbeschränkung. Er kann auch auf die geerbten privaten Methoden zurückgreifen. Allerdings hat der Erbe keine Möglichkeit den Status, der durch die Erbschaft erhaltenen Methoden, zu ändern. Der Erbe muß also die Schnittstelle des Erblassers übernehmen. Die Erlaubnis zur Schnittstellenerweiterung besteht und unter Einhaltung der Subtypbeziehung zum Erblasser sind auch Signaturveränderungen möglich.

```
class Counted(E <*: Equality)
...
public methods
size :Int
  {elementCount}
private
  elementCount :Int
methods
;
```

Abbildung 6.2: Kapselung

6.2 Mehrfachvererbung

TooL bietet dem Programmierer Mehrfachvererbung an. Diese kann vom Subklassenentwerfer genutzt werden, um die durch die Strukturierung aufgeteilten Protokollblöcke je nach Bedarf wieder zusammenzuführen. Mit Hilfe des Schlüsselwortes **super** kann die Liste der Vorfahren einer Klasse definiert werden (Abbildung 6.3).

Probleme bei der Verwendung von Mehrfachvererbung können durch Namenskonflikte entstehen. Diese treten prinzipiell dadurch auf, daß zwei direkte Vorfahren Methoden gleicher Signatur vererben. Es muß dann entschieden werden, welche der geerbten Implementierungen

```
class Indexed(E <*: Equality)
super Bounded(E), Keyed(Int, E), Sortable(E)
...
;
```

Abbildung 6.3: Mehrfachvererbung

angesprochen werden soll. TooL bietet keine direkte Möglichkeit, beide Methodenimplementierungen zu nutzen. In Eiffel ist dies beispielsweise durch Umbenennung von Methoden erreichbar. Die Namenskonflikte werden in TooL durch Erstellung einer Klassenpräzedenzliste gelöst. Dazu werden die Klassen des betroffenen Ausschnitts des Vererbungsgraphen durch topologische Sortierung sequentialisiert. Wird ein Methodenaufruf abgearbeitet, so setzt sich die Klasse durch, welche in der Präzedenzliste am weitesten vorne zu finden ist. Durch die Reihenfolge, in der die Vorgänger in der Liste der Oberklassen angegeben werden, ist das Sortierungsergebnis beeinflussbar und somit die Entscheidung zugunsten der einen oder der anderen Implementierung herbeizuführen.

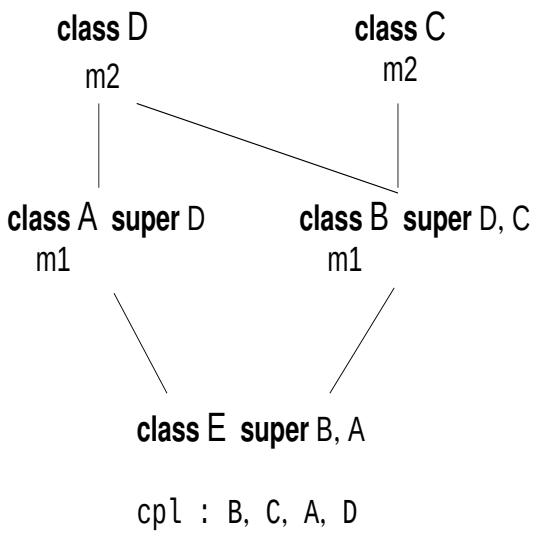

Abbildung 6.4: Klassenpräzedenzliste (cpl)

Durch jede in der Liste der Oberklassen angegebene Klasse wird ein Teilgraph des gesamten Vererbungsgraphen selektiert, der alle über diese Klasse erreichbaren Vorfahren umfaßt. Diese Teilgraphen werden zur Erstellung der Präzedenzliste der Reihe nach betrachtet, so daß die Klassen, des durch die erste Oberklasse selektierten Teilgraphen, im Prinzip auch die höhere Priorität haben. Kommt es jedoch zu Überschneidungen mit anderen Teilgraphen, so wird die betroffene, in beiden vorhandene Klasse erst nach den, in dem anderen Teilgraphen vor ihr liegenden Klassen, berücksichtigt.

In Abbildung 6.4 wird der Vererbungsgraph für eine Klasse E skizziert und die Präzedenzliste für diese Klasse angegeben. Die Klasse B in der Liste der Oberklassen von E selektierte

den ersten Teilgraphen. Die Sequenzialisierung ergibt aufgrund der Angaben in Klasse B die Reihenfolge B, D, C. Der zweite Teilgraph sequentialisiert sich zu A, D. In der Gesamtliste wird aufgrund der Überschneidung in Klasse D nur das zweite Auftreten von D berücksichtigt. Ein Namenskonflikt zwischen den Klassen A und B etwa bezüglich einer Methode m1, wie in Abbildung 6.4 angedeutet, würde zugunsten von Klasse B entschieden. Dies ist auch direkt aus der Reihenfolge in der Liste der Oberklassen von E ersichtlich, da diese Klassen direkte Vorfahren von E sind. Ein weiterer Konflikt zwischen den Klassen D und C, die Methode m2 betreffend führte zur Berücksichtigung der Version von Klasse C obwohl diese Version in Klasse B als der Klasse-D-Version nachrangig eingestuft wurde. Die Reihenfolge der Oberklassen in Klasse E dominiert hier die Angaben in den Oberklassen.

Beim Überschreiben von Methoden kann man mit Hilfe des Schlüsselwortes **super** in Verbindung mit dem Methodennamen die Implementierung der Oberklasse ansprechen. Welche Oberklasse gemeint ist, wird wiederum über die Präzedenzliste entschieden. Dabei sind auch Ketten von **super**-Aufrufen denkbar, wodurch dann auch die überschriebenen Implementierungen, der in der Präzedenzliste weiter hinten liegenden Klassen, erreichbar sind.

6.3 Metaklassen

Die streng objektorientierte Sichtweise von TooL legt die Integration von Metaklassen in das Sprachkonzept nahe [Gawecki, Matthes 95, S.14]. Auch Klassen werden als Objekte betrachtet, denen man Botschaften senden kann. Die Spezifikation der Botschaften die eine Klasse versteht, erfolgt in der Metaklasse. Die Zuordnung einer Metaklasse zu einer Klasse ist nicht automatisiert wie etwa in Smalltalk, sondern wird in TooL explizit über das Schlüsselwort **metaclass** gesteuert. Dadurch ist der Aufbau einer unabhängigen Metaklassenhierarchie möglich. In der Klassenbibliothek von TooL werden die Metaklassen gegenwärtig zur Definition von Methoden zur Objekterzeugung genutzt[1].

```
class SetClass(E <*: Equality, Instance <*: Set(E))
super Class(Instance)
public methods

create(stream :FiniteReadStream(E)) :Instance
  {
  let obj :Instance = new,
  stream.do(fun(e :E)obj.set(e)),
  obj
  }
;
```

Abbildung 6.5: Eine Metaklasse

Das durch eine Klasse spezifizierte Objekt muß erzeugt werden. Dies geschieht in TooL nicht durch fest eingebaute Konstruktoren, sondern durch Senden einer Erzeugungsnachricht an die Klasse des zu erzeugenden Objekts. Die entsprechende Methode wird daher in der

[1] Andere Möglichkeiten für den Einsatz von Metaklassen werden mit weiteren Verweisen bei [Gawecki, Matthes 95, S.14] genannt. Eine Übersicht auf die von verschiedenen Sprachen angebotenen Meta-Konzepte findet sich in [Siberski 95]

Metaklasse spezifiziert. Die Typisierung der Objekterzeugungsmethode erfolgt durch Parametrisierung der Metaklasse mit dem Objekttyp. Durch die Bekanntmachung des Objekttyps in der Metaklasse ist auch der Zugriff auf die Objektschnittstelle der zugeordneten Klasse möglich, was sich bei der Implementierung von Methoden ausnutzen läßt. So ist zum Beispiel die Erzeugung bereits gefüllter Behälter durch Verwendung von Einfügeprotokoll möglich. Eine weitere Anwendung liegt in der Definition von Klassenkonstanten, die für alle Objekte der Klasse nur einmal formuliert zu werden brauchen.

Als Beispiel diene die Metaklasse SetClass in Abbildung 6.5. Sie benötigt den zweiten Typparameter, da die Bindung für den Objekttyp-Parameter Instance selbst ein Objekttyp mit Typparameter ist, welcher den Elementtyp der Liste beschreibt. Sie erbt von der Metaklasse Class welche die Methode new definiert. Class ist auch die voreingestellte Metaklasse, die zugewiesen wird, wenn in der Klassendefinition keine Angabe Metaklasse erfolgt ist.

6.4 Abstrakte Klassen

Das Konzept der abstrakten Klasse ermöglicht die Spezifikation von abstraktem, (noch) nicht implementiertem Protokoll in Form von Methodensignaturen.[2]. Abstrakte Klassen können außerdem genutzt werden, um Protokoll bereits teilweise unter Bezugnahme auf nicht implementierte Basismethoden zu implementieren und um es dann in Subklassen durch Implementierung der Basismethoden individuell an die Bedürfnisse oder die Repräsentation der Subklasse anzupassen.

```
class Finite(E <*: Equality)
super Boxed(E)
metaclass AbstractClass
public methods

size :Int deferred

isEmpty :Bool
  {
  size == 0
  }
;
```

Abbildung 6.6: Abstrakte Klassen

In TooL wird die Zurückstellung der Methodenimplementierung durch das Schlüsselwort **deferred** gekennzeichnet, welches an Stelle des Methodenrumpfes eingefügt wird. Weiterhin ist es üblich, einer solchen abstrakten Klasse die Metaklasse AbstractClass zuzuordnen. Diese spezifiziert im Gegensatz zur Metaklasse Class keine new-Methode. Die so definierte abstrakte Klasse kann also kein Objekt des in ihr beschriebenen Typs erzeugen.

[2]In TooL ist auch die Formulierung von Vor- und Nachbedingungen erlaubt. Siehe dazu auch Abschnitt 6.7

6.5 Typisierung

TooL bleibt, wie schon erwähnt, eine strikt und statisch typisierte Sprache entsprechend ihrer Vorgängerin TL. Die Klassendefinition impliziert die Deklaration des der Klasse zugeordneten Objekttyps. Dieser ergibt sich aus den Signaturen der öffentlichen Methoden. Die Möglichkeiten zu rekursiven Deklarationen beschränken sich auf den Typbezeichner **Self**, welcher den Objekttyp der Klasse selbst beschreibt, und den Wertbezeichner **self**, welcher innerhalb des Methodenrumpfes für das rekursive Ansprechen des Empfängerobjektes zur Verfügung steht. Eine weitere Möglichkeit zur rekursiven Typdefinition bietet aber auch die Verwendung des Klassennamens selbst. Die Unterschiede der beiden Varianten liegen in den Möglichkeiten der Subklassenbildung und der Beziehung zwischen den Typen der Sub- und der Oberklasse. Dies soll im folgenden erläutert werden.

Im Unterschied zu anderen statisch typisierten objektorientierten Sprachen bietet TooL nicht nur eine Subtyprelation ($<:$) auf Objekttypen an, sondern kombiniert diese mit einer weiteren Typrelation ($<*:$). Diese Ähnlichkeitsrelation (type matching relation)[3] ermöglicht die typsichere Vererbung von binären Methoden oder allgemeiner von solchen Methoden, die Parameter vom Typ **Self** haben. Derartigen Methoden muß also ein Objekt übergeben werden, dessen Typ durch die Klasse, in der die Methode definiert wurde, beschrieben ist. Die Vererbung solcher Methoden kann bei Verwendung der reinen Subtypbeziehung als Konsistenzbedingung für die Vererbung nicht erlaubt werden. Die Spezialisierung des Objekttyps beim gleichzeitigen Vorhandensein binärer Methoden führt zur Verletzung der Subtypbeziehung aufgrund des kontravarianten Auftretens des Objekttyps als Parametertyp einer solchen Methode (siehe auch Abschnitt 5.2.2.1 und [Gawecki, Matthes 95, S.8]).

Die Verknüpfung der Ähnlichkeitsrelation mit der Vererbungsbeziehung, das heißt die Verwendung dieser Relation als weitere Konsistenzbedingung für die Beziehung der Objekttypen von Sub- und Oberklasse, soll nun diese zusätzliche Möglichkeit der Spezialisierung offenhalten. Eine informelle Definition der Ähnlichkeitsbeziehung lautet:

> A $<*:$ B *wenn für jeden Typbezeichner* **Self** *in* A
> *ein korrespondierender Bezeichner* **Self** *in* B *existiert*
> *und unter der Annahme, daß korrespondierende Typbezeichner* **Self**
> *den gleichen Typ bezeichnen,* A $\prec$ B *folgt.*

Mit korrespondierenden Bezeichnern sind solche Bezeichner gemeint, die in den beiden beteiligten Typdefinitionen (Klassendefinitionen) an gleicher Stelle auftauchen. Dies kann z.B. als Parametertyp gleichnamiger Methoden oder als Typ gleichnamiger Variablen der Fall sein.

Durch die Verknüpfung der Ähnlichkeitsrelation mit der Vererbungsrelation und aus der Tatsache, daß es sich bei dieser Relation um eine 'bedingte' Subtypbeziehung handelt, folgt unmittelbar, daß es keine erzwungene Subtypbeziehung zwischen Sub- und Oberklasse gibt. Vererbungshierarchie und Subtyphierarchie der Objekttypen der beteiligten Klassen sind damit nicht mehr identisch.

In Abbildung 6.7 wird die enge Verbindung der Ähnlichkeitsrelation mit dem Typbezeichner **Self** sowie die Bedeutung von Subtypbeziehung und Ähnlichkeitsrelation als Konsistenzbedingung für die Vererbung noch einmal verdeutlicht. Die unter a) dargestellte Vererbungssituation, behandelt den Fall der Spezialisierung durch die Subklasse, wobei in der Oberklasse

[3] Eingeführt von [Black, Hutchinson 90] und [Bruce 93]

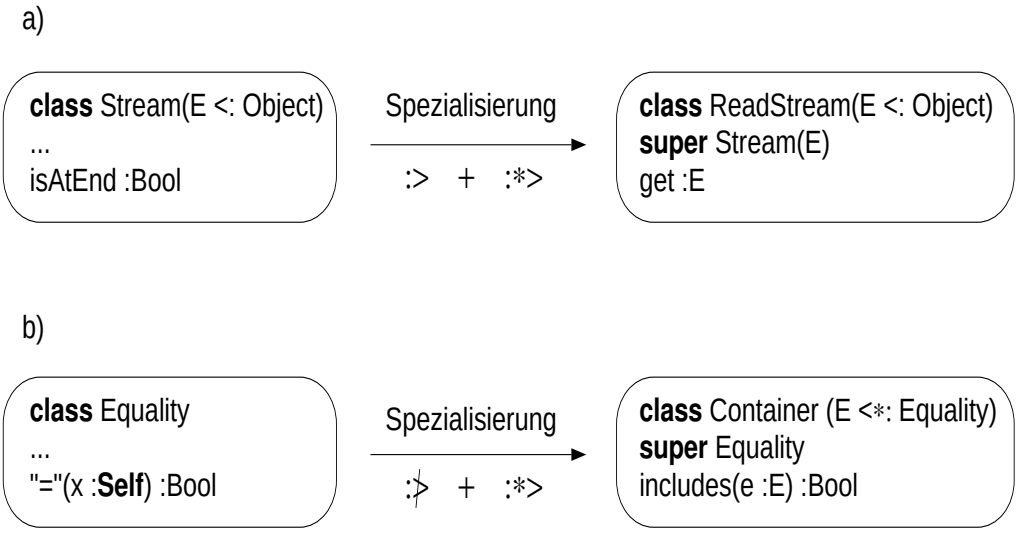

Abbildung 6.7: Verlust der Subtypbeziehung

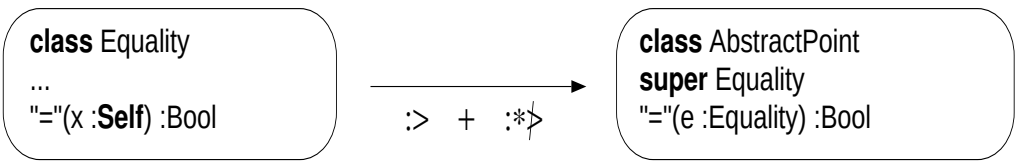

Abbildung 6.8: Verlust der Ähnlichkeitsbeziehung

kein Selbstbezug unter Verwendung des Typbezeichners **Self** zu finden ist. Die Klasse Stream wird von ReadStream durch Hinzufügen einer weiteren Methode spezialisiert. In dieser Situation sind die beteiligten Objekttypen in beiden Relationen enthalten. Für die Vererbung bedeutet dies: solange kein Typbezeichner **Self** benutzt wird, ist die Mitgliedschaft in der Subtyprelation die einzige Konsistenzbedingung für die Vererbung.

In Fall b) handelt es sich ebenfalls um eine Spezialisierungssituation, nur wird in diesem Fall der Typbezeichner **Self** in der Oberklasse Equality als Parametertyp für die Methode "=" benutzt. Die Klasse Container erbt diese Methode und führt gleichzeitig eine weitere Methode ein, sie spezialisiert also. Der Typbezeichner **Self** bezeichnet nun in beiden Klassen unterschiedliche Typen, nämlich gerade den Objekttyp der Klasse Equality bzw. den spezialisierten Objekttyp von Container. Daher wird die Subtypbeziehung zwischen diesen Objekttypen durch die Parameterspezialisierung (Kontravarianzregel) verhindert. Die Ähnlichkeitsbeziehung jedoch ist erfüllt. Denn wenn die korrespondierenden Bezeichner **Self** in den beiden equal-Methoden den gleichen Typ beschreiben würden, so wäre die Subtypbeziehung erhalten geblieben. Laut Definition ist dies aber für die Mitgliedschaft in der Ähnlichkeitsrelation ausreichend. Die Vererbung ist daher korrekt durchgeführt worden.

Es kann aber auch zum Verlust der Ähnlichkeitsbeziehung kommen, wie schon der zweite Satz der Definition vermuten läßt : ... *korrespondierende Typbezeichner* **Self** *müssen existie-*

ren. In Abbildung 6.8 findet sich eine solche Situation. Die Klasse AbstractPoint überschreibt die equal-Methode so, daß sie nur Equality-Objekte vergleichen kann. Trotz eventuell weiterer Spezialisierung durch AbstractPoint besteht hier eine Subtypbeziehung zwischen den Objekttypen der beiden Klassen, da diese Spezialisierungen nicht mehr kontravariant sichtbar werden. Die Ähnlichkeitsbeziehung ist jedoch nicht erfüllt, da es zu dem **Self**-Parameter keinen korrespondierenden Bezeichner in AbstractPoint gibt.

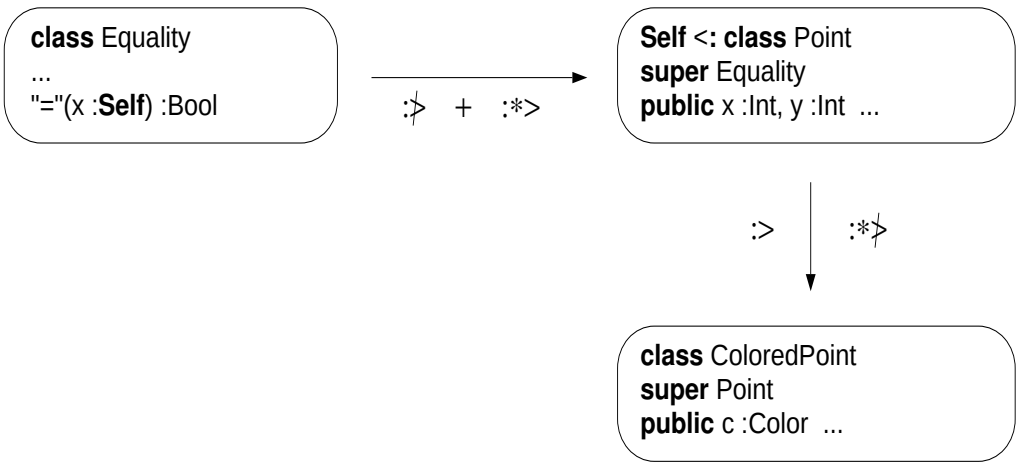

Abbildung 6.9: Rückgewinnung der Subtypbeziehung

Das Umschalten von Ähnlichkeit auf Subtypbeziehung in der Vererbungshierarchie, welches hier noch direkt durch Überschreiben durchgeführt wurde, kann in TooL auch mit Hilfe eines speziellen Konstruktes erfolgen, welches eine Subtypbeschränkung für den Typ **Self** festlegt. Mit der Formulierung

Self <: **class** AbstractPoint ...

wird dem Typüberprüfungsalgorithmus mitgeteilt, daß in allen Nachfolgern der Klasse der Bezeichner **Self** nicht mehr den Objekttyp der aktuellen Klasse, sondern den der Wurzelklasse (hier AbstractPoint) beschreibt. Eine diese Bedingung erfüllende Nutzung der Vererbung wird dann unterhalb einer so formulierten Klasse erzwungen.

Die Einführung des Konstruktes soll die Möglichkeit eröffnen auf einfache Weise den Weg zurück zu einer die Subtypbeziehung einhaltenden Vererbungshierarchie zu gehen, trotz einer weiter oben im Vererbungsgraphen erfolgten kontravarianten Spezialisierung. In Abbildung 6.9 ist diese Situation skizziert. Die nun spezialisierende Klasse Point erfüllt die Ähnlichkeitsbeziehung zu Equality und nutzt die Subtypbeschränkung für **Self**, um unter sich eine Punktehierarchie entstehen zu lassen, für welche die Möglichkeit zur Subsumption unter dem Objekttyp der Wurzelklasse Point besteht. Die Ähnlichkeitsbeziehung zu Point geht aber verloren, da die Verwendung von '**Self** <:' in Klasse Point so interpretiert wird, als wenn in den Subklassen alle Typbezeichner **Self** durch den Typ Point ersetzt würden [Gawecki, Matthes 96, S.13].

6.6 Polymorphismus

TooL bietet wie TL den Subtyppolymorphismus[4] an. Das heißt Variablen und Parameter können statt mit Werten ihres Deklarationstyps auch mit Werten eines Subtyps belegt werden. Weiterhin unterstützt TooL (ebenfalls in Nachfolge von TL) den gebunden-parametrischen Polymorphismus in Bezug auf die Subtyprelation.

```
class AbstractLinkedList(E <: Object, L <*: Link(E))
super AbstractList(E)
metaclass AbstractClass
public methods
item :E {listStart.item}
...
private
listStart :L, listEnd:L  /Idots
;
```

Abbildung 6.10: Parametrischer Polymorphismus

Das bedeutet die Möglichkeit Methoden und Klassen durch Typparameter generisch zu halten, die aber nur Subtypen des einschränkenden Typs als Übergabetyp zulassen. Orthogonal zu dieser Variante des Polymorphismus erlaubt TooL mit der Einführung der Ähnlichkeitsrelation auch einen gebunden-parametrischen Polymorphismus in Bezug auf diese Typrelation. Diese Variante des Polymorphismus in TooL entspricht dem sogenannten 'f-bounded polymorphism' [Abadi, Cardelli 95]. Durch diese Ergänzung besteht die Möglichkeit, generische Anwendungen auch für Objekte zu programmieren, deren Objekttypen nur in Ähnlichkeitsbeziehung stehen, wodurch der Verlust der Subsumptionsfähigkeit etwas relativiert wird. Als Beispiel diene die Klasse AbstractLinkedList in Abbildung 6.10, welche beide Varianten von Typparametern nutzt.

6.7 Semantikspezifikation

TooL unterstützt das Prinzip des 'design by contract' [Meyer 94a, S.23], indem es die Möglichkeit zur Formulierung von einfachen Vor- und Nachbedingungen sowie zur Definition von Klasseninvarianten bietet. Die Bedingungen werden durch Angabe von 'booleschen' Ausdrücken spezifiziert. Dabei kann man die Funktionalität der Klasse Bool nutzen und sich auf die in der Klasse bekannten Methoden beziehen. Eine Unterscheidung des Zustands vor Ausführung einer Methode von dem danach wird derzeit nicht unterstützt[5]. Vor- und Nachbedingungen werden zwischen Methodensignatur und Methodenrumpf mit Hilfe der Schlüsselworte **require** und **ensure** plaziert, während die Klasseninvarianten zwischen der Metaklassenangabe und dem Schlüsselwort **public** angesiedelt sind.

Eine Vererbung der Prädikate erfolgt nicht und ist im allgemeinen auch nicht sinnvoll [Meyer 88]. Ein Anwendungsfall für die Definition einer Vorbedingung ist beispielsweise die

[4] siehe [Cardelli, Wegner 85] für die Klassifizierung der verschiedenen Arten von Polymorphismus

[5] In Eiffel ist dies etwa für Nachbedingungen mit Hilfe des Schlüsselwortes **old** möglich:
ensure index = old index + 1.

```
class Bilinear(E <*: Object)
super Linear(E)
metaclass AbstractClass
invariant (isEmpty => isOff)
public methods ...
gotoIndex(pos :Int) :Void
  require (pos >= 0) & !isEmpty
  ensure isOff || (index == pos)
  {...}
;
```

Abbildung 6.11: Semantikspezifikation

Festlegung eines gültigen Wertebereichs für die Methodenparameter. Durch die Spezialisierung in der Subklasse kann es aber erwünscht sein, den Bereich an gültigen Werten für die Methodenparameter zu erweitern. Wenn die Bereichsfestlegung aus der Oberklasse übernommen würde, wäre dies nicht möglich.

Die Formulierung von Invarianten für Methoden beziehungsweise die Überprüfung weiterer berechnungsabhängiger Bedingungen ist mit Hilfe des Schlüsselwortes **assert** an jeder Stelle des Methodenrumpfes erlaubt. In Abbildung 6.11 sind die verschiedenen Formen der Semantikspezifikation noch einmal dargestellt.

Kapitel 7

Die TooL Behälterklassenbibliothek

Im folgenden wird die Behälterklassenbibliothek von TooL beschrieben, indem zunächst noch einmal die Zielrichtung des Entwurfs formuliert wird. Abschnitt 7.1 beschreibt das Klassifizierungsschema. In weiteren Abschnitten wird dann die Klassifizierung der allgemeinen Behältereigenschaften vorgestellt (Abschnitt 7.2). Anschließend folgt ein Überblick über die spezielleren Behälterkategorien (Abschnitt 7.3). Der Abschnitt 7.4 stellt interessante Implementierungen unter anderem im Hinblick auf die Verwendung der Ähnlichkeitsrelation vor. Die in TooL besondere Problematik der Verwendung von Typparametern zur Beschreibung des Elementtyps der Behälter wird in Abschnitt 7.5 diskutiert. Abschnitt 7.6 geht auf die zur Iterationsabstraktion verwendeten Konzepte ein. Schließlich erfolgt die Formulierung eines Zwischenergebnisses im Hinblick auf die gestellte Zielsetzung (Abschnitt 7.7). Weitere Abschnitte beschäftigen sich mit der Adaption von Entwurfsmustern und den verwendeten Entwurfshilfmitteln (7.8 und 7.9).

Die Implementierung der Behälterklassenbibliothek soll zur Beantwortung der folgenden Fragestellungen beitragen:

1. Ist es möglich, in einer Sprache, die zur streng typisierten Schnittstellenvererbung zwingt, eine Behälterklassenbibliothek zu entwerfen? Oder anders formuliert: lassen sich die Methodensignaturen so sauber klassifizieren, daß diese Methoden auch in den komplizierteren Objekten benutzbar bleiben und damit die einfache Sicht auf komplizierte Objekte garantiert werden kann?

2. Läßt sich gleichzeitig das Ziel hoher Wiederverwendbarkeit und Kombinierbarkeit von implementiertem Teilprotokoll erreichen?

3. Wird durch die Integration der Ähnlichkeitsrelation in das Typsystem von TooL und damit der Möglichkeit zur typsicheren kontravarianten Spezialisierung die Wiederverwendbarkeit erhöht?

Als Vorlage für den Entwurf dient die Behälterklassenbibliothek von Eiffel [Meyer 94a]. Diese fein strukturierte Bibliothek liefert vielfältige Ansätze zur Klassifizierung von Behälterprotokoll.

7.1 Klassifizierungsschema

Das im folgenden vorgestellte Klassifizierungsschema kann als Leitfaden für die Entwicklung, Erweiterung oder Veränderung der Hierarchie angesehen werden oder als Orientierungshilfe innerhalb des Vererbungsgraphen dienen. Es beruht auf der Beobachtung, daß sich die Behälterklassen durch Eigenschaften beschreiben lassen. Diesen Eigenschaften bzw. Kombinationen solcher Eigenschaften läßt sich jeweils ein bestimmtes Teilprotokoll zuordnen. Das Schema teilt nun das Protokoll eines Behälters in ein allgemeines und in ein spezielleres Protokoll auf. Das allgemeine Protokoll läßt sich allgemeinen, die Behälterkonzepte beschreibenden Eigenschaften zuordnen. Es enthält nur solche Methoden, die bereits auf diesem abstrakten Beschreibungsniveau sinnvoll beschrieben werden können. Das speziellere Protokoll kann erst zugeordnet werden, wenn bereits die Entscheidung für eine bestimmte Behälterkategorie durch Kombination einiger allgemeiner Eigenschaften gefallen ist. Ist eine solche Kategorie begründet, so läßt sich durch erneute Aufteilung nach spezielleren Eigenschaften der Behälterkategorie das spezielle Protokoll zuordnen. Konkrete Behälter ergeben sich dann aus der Kombination der speziellen Behältereigenschaften bzw. des mit ihnen verhafteten Protokolls. Die einzelnen Schritte des Klassifizierungsprozesses sind damit:

1. Klassifizierung der allgemeinen Eigenschaften von Behältern in einem weitgehend baumförmig strukturierten Teil des Vererbungsgraphen

2. Zusammenführung allgemeiner Eigenschaften zu allgemeinen Behälterkategorien

3. Erneute Aufteilung zur Beschreibung speziellerer Eigenschaften der Behälterkategorie

4. Kombination spezieller Eigenschaften zu entsprechenden Behältervarianten

Die Klassifizierung der allgemeinen Eigenschaften in einer Baumstruktur, wie unter 1. erwähnt, ergibt sich aus der Möglichkeit, die Eigenschaften dieser Ebene zu gruppieren und für jede dieser Gruppen einen Teilbaum von Eigenschaften zu bilden. Die Erläuterung der Gruppierung folgt in Abschnitt 7.2. Abbildung 7.1 soll das Schema graphisch darstellen.

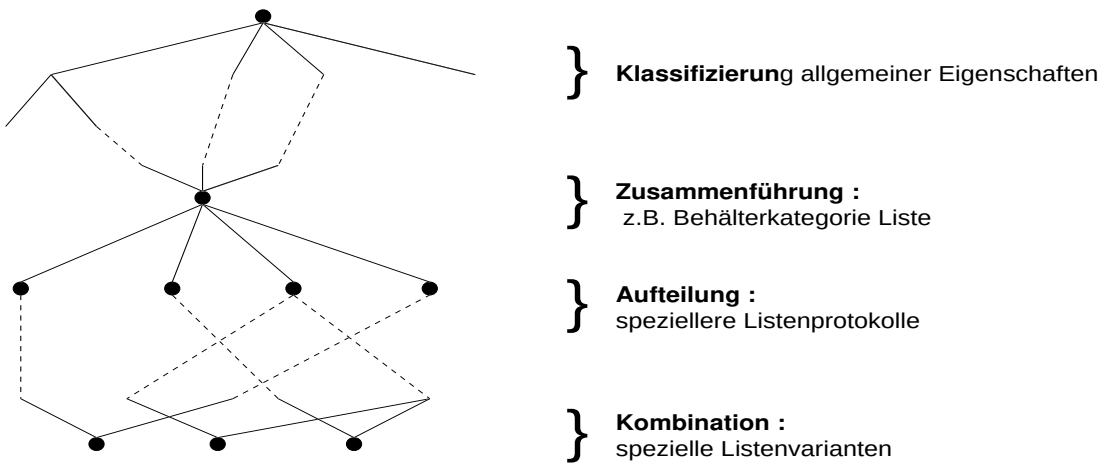

Abbildung 7.1: Klassifizierungsschema

Die Zeichnung auf der linken Seite der Abbildung ist eine Strukturskizze des Vererbungsgraphen. Im oberen Abschnitt ist die Baumstruktur zur Klassifizierung der allgemeinen Behältereigenschaften angedeutet. Als Bezugspunkt wurde die Behälterkategorie *Liste* gewählt. Die Wurzel der Listenhierarchie faßt einige allgemeine Behältereigenschaften zusammen. Durch diese Zusammenfassung erhält die Wurzelklasse bereits ein sehr einfaches Listenprotokoll, welches im Prinzip schon für die Bedienung ausreichen kann. Die Aufteilung der spezielleren Listenprotokolle erfolgt dann wieder eigenschaftsbezogen. Insbesondere in diesem Abschnitt der Klassifizierung bedeutet die Abgrenzung einer Eigenschaft nicht notwendigerweise die Definition weiterer Methoden. Es kann sich auch um eine Implementierungseigenschaft handeln, welche nur die bereits vorhandenen geerbten Methoden auf eine bestimmte Art und Weise implementiert. Diese kann dann mit den anderen, weitere Fähigkeiten vererbenden Eigenschaftsklassen kombiniert werden, sodaß sich immer komplexere implementierte Listen aufbauen lassen, bis schließlich die gewünschte Listenvariante zur Verfügung steht.

Die Abgrenzung des die allgemeinen Behältereigenschaften klassifizierenden Teilgraphens von den Behälterkategorien ist nicht so streng, wie es die Zeichnung zu vermitteln scheint. Eine spezielle Eigenschaft eines Behälters kann durchaus nur die Manifestierung einer allgemeinen Behältereigenschaft sein, die aber nicht auf alle Mitglieder der Behälterkategorie zutrifft. Diese allgemeine Eigenschaft kann dann nicht eine Eigenschaft der Wurzel der Behälterkategorie werden, da sonst das dieser Eigenschaft zugeordnete Protokoll für ein solches Mitglied unbrauchbar wird, was gerade vermieden werden sollte. Es wird ein spezielle Klasse zur Umsetzung dieser Eigenschaft für die aktuelle Behälterkategorie benötigt, in welcher dann aufgrund der Entscheidung für eine bestimmte Behälterart eventuell auch noch weitere Methoden definierbar werden.

Die z.B. von [Wetzel 94, S.36ff] getroffene und in Abschnitt 3.2.4 beschriebene Unterscheidung von Konzept- und Implementierungsvererbung läßt sich auf das Klassifizierungsschema und damit auf die Vererbungshierarchie folgendermaßen übertragen. Die Schritte 1–3 des Klassifizierungsprozesses legen eher die Konzeptvererbung nahe. Hier werden keine Behälterstrukturen festgelegt, sondern nur Protokollvarianten unterschieden und angeordnet. Erst Schritt 3 führt mit den Implementierungseigenschaften oder Implementierungsvarianten einer Behälterkategorie solche Strukturdefinitionen ein, sodaß schließlich in Schritt 4 bei der Kombination der speziellen Eigenschaften der Behälterkategorie und hier insbesondere bei der Kombination mit den Implementierungseigenschaften, die Implementierungsvererbung verstärkt zum Einsatz kommt. In Abschnitt 7.3.1 wird dies exemplarisch anhand der Listen noch einmal aufgegriffen (siehe auch Abbildung 7.12).

7.2 Allgemeine Behältereigenschaften

Für die allgemeinen Behältereigenschaften lassen sich, wie auch in der Containerhierarchie von Eiffel geschehen, drei Hauptkategorien bilden. Sie unterscheiden die wichtigsten Protokollgruppen für die Benutzung eines Behälters. Es sind:

1. Eigenschaften der Aufnahmefähigkeit.

2. Eigenschaften, die Formen des Elementzugriffs beschreiben.

3. Eigenschaften, die Formen des Traversierens beschreiben.

Innerhalb dieser Eigenschaftsgruppen läßt sich eine Strukturierung oft durch Beantwortung einfacher Fragen erreichen, die dann zur Unterscheidung von Eigenschaften führt. So beschäftigt man sich in der unter 1. genannten Gruppe z.B. mit den Fragen: Wieviele Elemente sind enthalten? Gibt es eine Kapazitätsgrenze? Die zweite Gruppe stellt die Frage: Wie wird ein Element identifiziert? Und unter 3. steht die Frage nach der Art des Traversierens im Vordergrund. Über diese Kategorien hinaus gibt es weitere wichtige Eigenschaften wie die Sortierbarkeit des Behälters oder die 'Sortiertheit'. Letztere soll die Eigenschaft beschreiben, daß die Elemente des Behälters immer sortiert gehalten werden.

Die Aufteilung des Protokolls kann in TooL aber nicht wie in Eiffel vorgenommen werden, da TooL keine Möglichkeit bietet, Methoden umzubenennen und somit den Programmierer zur Schnittstellenvererbung zwingt. Dies erfordert innerhalb der Eigenschaftsgruppen und insbesondere in der zweiten Gruppe eine andere Organisation der Eigenschaftshierarchie.

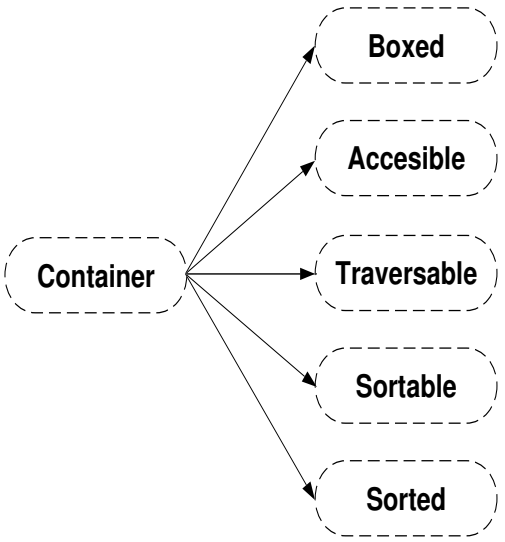

Abbildung 7.2: Allgemeine Eigenschaftsgruppen

Abbildung 7.2 bietet eine Übersicht auf die derzeit unterschiedenen Eigenschaftsgruppen und zeigt gleichzeitig den Wurzelbereich des Vererbungsgraphen der Behälterklassenbibliothek. Die gestrichelte Umrandung der Klassen soll verdeutlichen, daß es sich um abstrakte Klassen handelt. Die Verbindungspfeile symbolisieren die Vererbungsbeziehung[1].

Als allgemeine Eigenschaft der Behälterklassen könnte man auch die Tatsache betrachten, daß alle Behälterklassen einen Typparameter zur Beschreibung des Elementtyps erhalten (Abbildung 7.3). Es handelt sich also um generische Klassen, die sich durch verschiedene Elementtypen instantiieren lassen. Für den Elementtyp wird in der derzeitigen Version der Klassenbibliothek eine Ähnlichkeitsbeziehung zum Objekttyp der Klasse Equality verlangt. Anders formuliert bedeutet dies, daß jedes Elementobjekt eines Behälters die Methode "=" zum Vergleich auf Gleichheit mit einem anderen Elementobjekt verstehen muß. Die spezielle Typisierungsproblematik dieses Parameters wird im Abschnitt 7.5 gesondert betrachtet. Außerdem wird dort eine Alternative für die Ähnlichkeitsbeziehung zum Objekttyp Equality

[1] Die Pfeile folgen den vererbten Eigenschaften. Sie zeigen also von der Oberklasse zur Subklasse.

diskutiert.

Das allgemeinste, insofern eigenschaftsunabhängige Protokoll findet sich in der Wurzelklasse Container der Behälterhierarchie und soll Gegenstand des nächsten Abschnitts sein.

7.2.1 Eigenschaftsunabhängiges Protokoll

Mit eigenschaftsunabhängigem Protokoll sind diejenigen Methoden gemeint, welche jeder Container spezifizieren soll. Konzeptionell ließen sich diese Methoden teilweise auch in den Eigenschaftsgruppen unterbringen, sodaß die Klasse Container entfallen könnte. Sie lassen sich jedoch auch gut in den anderen Gruppen nutzen, etwa zur Formulierung von Vor- und Nachbedingungen oder für die Implementierung anderer Methoden. Wenn diese Methoden also dennoch der Klasse Container zugewiesen werden, so geschieht dies, weil dadurch frühzeitige Querverbindungen im Vererbungsgraphen vermieden werden. Dadurch wird die Übersichtlichkeit erhöht.

Die wichtigsten in Klasse Container definierten Methoden sind in Abbildung 7.3 zu sehen.

```
class Container(E <*: Equality)
super Equality
metaclass AbstractClass
public methods ...
includes(e :E) :Bool deferred
isEmpty :Bool {elements.isAtEnd}
elements :FiniteReadStream(E) deferred
do(statement :Fun(:E):Void) :Void deferred
...
;
```

Abbildung 7.3: Eigenschaftsunabhängiges Protokoll

Die Methode includes soll die Mitgliedschaft eines Elementes im Behälter feststellen. Eine solche Aussage sollte unabhängig von Traversierungseigenschaften, der Frage wieviele Elemente enthalten sind, oder ob das Element etwa doppelt enthalten ist, möglich sein.

Etwas problematisch ist die Vererbung an die Eigenschaftsgruppe der Aufnahmefähigkeit und dort insbesondere an die Eigenschaft Infinite. Genaueres dazu im diese Gruppe behandelnden, folgenden Abschnitt.

Die interessanteste der abgebildeten Methoden ist sicherlich elements. Durch diese Methode wird von jedem Behälter verlangt, daß er seine Elemente aufzählen kann. Sie liefert einen ReadStream zurück, über den dann die Möglichkeit besteht, die Elemente des Behälters eines nach dem anderen abzurufen. Die Streams stellen einen von den Behälterklassen unabhängigen Teil des Vererbungsgraphen von TooL und bieten verschiedene Varianten der Iterationsabstraktion. Eine genauere Beschreibung folgt noch im Abschnitt 7.6. Für die Behälterklassen ist aber schon hier interessant zu erwähnen, daß der Rückgabetyp ReadStream der elements-Methode von Subklassen spezialisiert werden kann. Die Stream-Hierarchie ist so gestaltet, daß für die Nachfolger von ReadStream die Subtypbeziehung erhalten bleibt. Daher kann der Rückgabetyp der Methode elements beim Überschreiben der Methode durch einen spezielleren Stream ersetzt werden.

Die Fähigkeit des Behälters, seine Elemente aufzuzählen, könnte hier bereits zur einfachen Implementierung der anderen Methoden verwendet werden. Dies geschieht aber nicht, wiederum aus Rücksicht gegenüber der Klasse Infinite (näheres im nächsten Abschnitt). Die gesonderte Aufnahme der über die Elemente des Behälters iterierenden Methode do, welche potentiell auch in der Stream-Hierarchie zu finden wäre, wurde vorgenommen, um die Behälter anzuhalten, eine einfachere, die komplexe Implementierung der Streams vermeidende, effizientere Version dieser wichtigen Methode zur Verfügung zu stellen.

7.2.2 Eigenschaften der Aufnahmefähigkeit

Die Eigenschaften der Aufnahmefähigkeit beantworten die Fragen: Wieviele Elemente enthält der Behälter? Gibt es einen Kapazitätsgrenze? Ist diese Grenze verschiebbar? Die Wurzelklasse des Teilgraphen wie er in Abbildung 7.4 zu sehen ist, definiert eine Aussage als unabhängig von den anderen Aufnahmefähigkeitseigenschaften. Dies ist die Aussage, daß alle Behälter potentiell gefüllt seien könnten, und schlägt sich nieder in der Spezifikation der Methode isFull in dieser Klasse. Anhand dieser Methode lassen sich die anderen Eigenschaften der Aufnahmefähigkeit gut unterscheiden, denn sie enthalten jeweils eine andere Implementierung dieser Methode. Die beiden Nachfolger von Boxed beantworten die Frage nach der Anzahl der Elemente.

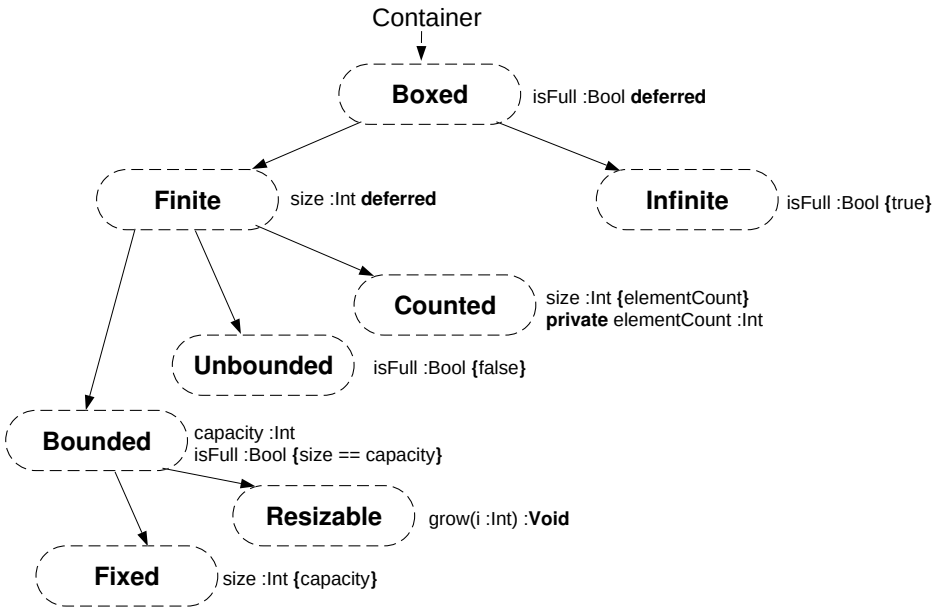

Abbildung 7.4: Eigenschaften der Aufnahmefähigkeit

Da ist zunächst die Klasse Infinite, welche für die Eigenschaft eines Behälters immer unendlich viele Elemente zu enthalten steht. Dies ist für den klassischen Behälterbegriff eine ungewöhnliche Annahme. Es ist aber dennoch möglich, sich entsprechende Behälter vorzustellen. Da wäre zum Beispiel ein Behälter der Zufallszahlen enthält, oder einer, der alle Primzahlen beinhaltet. Es ist also möglich für Behälter die ihre Elemente funktional berechnen können. Mit dieser Betrachtungsweise ist ein unendlicher Behälter immer gefüllt und die

Methode isFull ist daher entsprechend durch true implementiert. Es können keine Elemente eingefügt werden.

Die aus Container ererbten Methoden sind teilweise problematisch. Insbesondere die Frage des Enthaltenseins eines Elementes scheint für eine unendlich Menge nicht vernünftig zu sein, da hier falls das Element nicht vorhanden ist potentiell die Gefahr besteht, daß die Berechnung nicht terminiert. Es lassen sich aber in den angegebenen Beispielfällen Implementierungen angeben: Die Eigenschaft Primzahl kann direkt nachgewiesen werden und ein Behälter von Zufallszahlen enthält alle Zahlen des Generierungsbereichs. Die Definition eines die Elemente aufzählenden Streams (für die Implementierung der Methode elements) macht insoweit keine Probleme, als die hier in Gestalt der Klasse ReadStream verlangte Form der Streams nur eine die Elemente Schritt für Schritt abrufende Funktionalität zur Verfügung stellt. Die die komplexere Funktionalität fordernde Signatur der elements-Methode mit FiniteReadStream als Rückgabetyp, wird daher erst in Accessible der Wurzel der Zugriffsformgruppe spezifiziert. Die Kombination mit dem Protokoll dieser Gruppe (insbesondere Einfüge- und Löschmethoden) wird bei der hier verwendeten Interpretation von Infinite als unpassend betrachtet. Kombinierbar bleibt ein unendlicher Behälter allerdings mit dem Traversierungsprotokoll, was dort allerdings auch zur Komplizierung der Struktur beigetragen hat. Es bleibt anzumerken, daß sich die gefundenen Beispiele für unendliche Behälter auch auf Streams abbilden lassen. Die Beseitigung dieser Eigenschaft scheint daher möglich.

Die gegenteilige Antwort auf die Frage nach den Elementen ist dann, daß der Behälter zu jedem Betrachtungszeitpunkt endlich viele Elemente enthält. Dieser Aussage entspricht die Klasse Finite. Die Eigenschaft, endlich viele Elemente zu enthalten, ermöglicht die Einführung der Methode size, wie dies auch in Abbildung 7.4 angedeutet ist.

Bei der Subklasse Counted handelt es sich um den Repräsentanten einer Implementierungseigenschaft (siehe Abschnitt 7.1). Sie steht für eine häufige Implementierungsform der Methode size, welche die Methode durch Verwaltung der Anzahl der aktuell enthaltenen Elemente mit Hilfe einer Zählvariablen realisiert. Die Klasse führt also diese nicht öffentliche Variable ein und implementiert die geerbte Methode size. Behälter mit der Eigenschaft Finite können dann bei Bedarf diese Klasse dazu erben.

Die beiden weiteren Nachfolger der Klasse Finite beantworten die Frage nach dem Vorhandensein einer Kapazitätsgrenze. Die Klasse Unbounded steht dabei für das Fehlen einer solchen Grenze. In diesem Fall kann immer ein Element eingefügt werden. Ein solcher Behälter ist niemals gefüllt und die Methode isFull antwortet daher immer mit false. Die Klasse Bounded steht für das Vorhandensein einer solchen Kapazitätsgrenze. Dann ist verständlich, daß man auch diese Grenze in Erfahrung bringen möchte, was die Einführung der Methode capacity begründet. Die Aussage über die das Erreichen der Kapazitätsgrenze kann nun durch Vergleich von Kapazitätsgrenze und Elementanzahl getroffen werden. Und so geschieht es auch bei der Implementierung der Methode isFull.

Die letzte zu beantwortende Frage dieser Eigenschaftsgruppe betrifft die Verschiebbarkeit der Kapazitätsgrenze. Die beiden Nachfolger von Bounded repräsentieren die beiden möglichen Varianten. Die Klasse Fixed für die nicht verschiebbare Grenze implementiert nun auch die Methode size, und zwar über die Methode capacity. Dies hat zur Folge, daß die Methode isFull auch für einen Behälter mit der Eigenschaft Fixed immer mit true antwortet. Dabei wird davon ausgegangen, daß ein Behälter fester Größe nicht mehr unterscheidet, ob seine 'Slots' noch den Initialisierungswert enthalten, oder ob bereits weitere Zuweisungen auf sie erfolgt sind. Andere Varianten sind natürlich denkbar und können durch Überschreiben durchgesetzt werden.

Der folgende Abschnitt eröffnet mit der Beschreibung der Zugriffsformeigenschaften die zweite Hauptgruppe der allgemeinen Behältereigenschaften. Hier wird unter anderem das wichtige Einfüge- und Löschprotokoll klassifiziert.

7.2.3 Zugriffsformen

Die Eigenschaftsgruppe der Zugriffsformen beantwortet die Frage nach der Identifizierungsmöglichkeit, die der Behälter für seine Elemente anbietet. Es werden drei Varianten unterschieden:

1. Assoziativ: Das Element identifiziert sich selbst. Zugriff durch Übergabe des Elementes, z.B. bei Mengen

2. Identifikation durch 'Auszeichnung': Der Behälter stellt immer ein 'aktives' Element bereit, z.B. bei Listen.

3. Identifikation durch Schlüssel: Den Elementen ist ein Zugriffsschlüssel zugeordnet, z.B. in Verzeichnissen (Dictionary).

Abbildung 7.5 stellt den die Klassifizierung realisierenden Teilgraphen dar. Die erste und einfachste Form des Zugriffs ist, daß Element selbst zur Identifizierung zu verwenden. Will der Benutzer behälterspezifische Informationen über ein bestimmtes Element erhalten, oder will er eine Aktion bezüglich dieses Elementes einleiten, so muß er der entsprechenden Methode das Element übergeben.

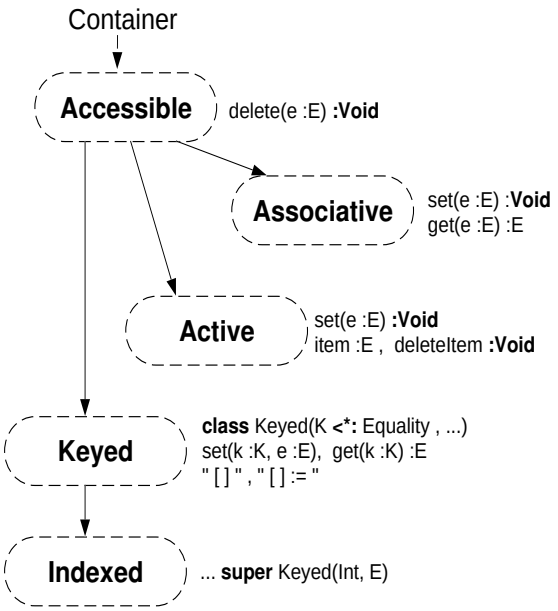

Abbildung 7.5: Klassifizierung der Zugriffsformen

Die der Eigenschaft zugeordnete Klasse heißt Associative. Ein Behälter, der nur diese Zugriffsart kennt, kann kein Element mehrfach enthalten. Denn er könnte unabhängig davon,

ob er nun ein dem übergebenen Objekt gleichartiges oder eine identisches Element enthält, immer nur dieses Element ansprechen. Das Einfügen eines weiteren, zu einem enthaltenen identischen oder gleichartigen Elements könnte dann auch bestenfalls zum Überschreiben des Elementes führen. Mehrfache Vorkommen eines Elementes könnten nicht unterschieden werden. Ein solcher rein assoziativer Behälter benötigt eine Möglichkeit, das in ihm identifizierte Objekt zurückzugeben. Wenn es durch ein nicht identisches Element identifiziert wurde, kann es erwünscht sein, daß im Behälter enthaltene Objekt zurückzuerhalten. Daher findet sich in Associative die Spezifikation einer Zugriffsmethode get, die sowohl als Parametertyp als auch als Rückgabetyp den Elementtyp definiert. Die Methode set in Associative steht für das assoziative Einfügen ohne die Möglichkeit, mehrere identische oder gleiche Elemente zu verwalten. Typische Subklassen dieser Eigenschaft sind z.B. Set und Hashtable.

Durch die zweite Zugriffsform, welche von der Klasse Active repräsentiert wird, bietet der Behälter immer ein Element als 'aktives' Element an. Das heißt, er spezifiziert eine spezielle Methode, hier item genannt, über welche dieses Element abgerufen werden kann. Weiterhin stellt er auf dieses Element bezogene Aktionen zur Verfügung wie z.B. die Methode deleteItem. Durch die Auszeichnung werden nun auch im Behälter enthaltene identische oder gleichartige Elemente unterscheidbar. Die Einfügemethode set hat zwar die Signatur der Methode in Associative, repräsentiert aber für das konzeptionell andere Einfügen mit potentiellem Einfluß auf die aktive Position bzw. das Einfügen mehrerer identischer oder gleichartiger Elemente. Als Subklassen kommen für diese Eigenschaft zum Beispiel Listen, Stapel oder Warteschlangen in Betracht.

Die dritte Variante der Zugriffsformen über Schlüssel findet ihrer Entsprechung in der Klasse Keyed. Diese Klasse erhält einen weiteren Typparameter für den Schlüsseltyp. Auch für diesen wird mindestens eine Ähnlichkeit zur Klasse Equality verlangt, um die Vergleichbarkeit der Schlüssel sicherzustellen. Für diesen Parameter lassen sich die Überlegungen zu Alternativen für die vom Elementtyp verlangte Ähnlichkeitsbeziehung aus Abschnitt 7.5 übernehmen. Durch die Identifikation von Elementen über Schlüssel sind auch in solchen Behältern gleichartige oder identische Elemente unterscheidbar. Die Einfügemethode set benötigt nun einen zusätzlichen Schlüsselparameter um die Zuordnung von Element und Schlüssel herzustellen. Des weiteren wird natürlich die das Element identifizierende Methode "[]" mit Parameter vom Schlüsseltyp und Rückgabetyp vom Elementtyp sowie die syntaktisch verwandte Methode "[] :=" zum Überschreiben einer Schlüsselzuordung angeboten[2]. Typische Subklasse dieser Eigenschaft ist z.B. ein Dictionary.

Die letzte Klasse dieser Eigenschaftsgruppe Indexed stellt eine Spezialisierung der Zugriffsform über Schlüssel dar. Der Schlüsseltyp wird auf ganzzahlige Werte beschränkt. Dies erfolgt einfach durch Instantiierung des Schlüsseltyps der Oberklasse Keyed mit Int. Die Klasse Indexed benötigt erneut nur den Elementtypparameter. Die nun auf den Schlüsseln definierte Ordnungsrelation läßt in dieser Klasse einige Implementierungen zu, so z.B. für die Methode elements aus Container, welche die Verbindung zu den Streams herstellte. Bei dieser Methode läßt sich die Ordnung auf den Schlüsseln auch zur Spezialisierung des Rückgabetyps auf positionierbare Streams nutzen. Typische Nachfolger dieser Klasse sind Arrays.

Zurück zur Zugriffsform mittels Identifikation durch das Element selbst, wie sie durch Associative repräsentiert wird. Sie nimmt insofern eine Sonderrolle ein, als daß Teile des ihr

[2]Diese speziellen Methoden sind syntaktische Varianten der ebenfalls vorhandenen Methoden get und set. Sie erlauben die Angabe des Schlüssels in der gewohnten 'Array-Schreibweise' zwischen den beiden eckigen Klammern

zuzuordnenden Protokolls auch mit den anderen Formen harmonieren. Dieses Protokoll wird daher schon in Vorgängern von Associative eingeführt. So findet sich die Methode includes, eine assoziative Methode, bereits in der Klasse Container, und in der Klasse Accessible wird bereits das assoziative Löschen durch die Methode delete angeboten. Diese Identifikationsform ist deshalb letztlich in allen Behältern anzutreffen, und Associative repräsentiert nur noch diejenigen Behälter, die keine andere, ergänzende Form des Zugriffs anbieten.

7.2.4 Traversierungsformen

Die Eigenschaft der Traversierbarkeit bedeutet für einen Behälter, daß er eine Strukturidee für die Anordnung seiner Elemente explizt macht. Er bietet daher an seiner Schnittstelle Methoden an, die es ermöglichen, sich innerhalb dieser Struktur von einem Element zum anderen zu bewegen. Dies ist im Unterschied zu der Iterationsabstraktion der Streams zu sehen, welche jeder Behälter über die in der Klasse Container definierte Methode elements zur Verfügung stellen muß, auch wenn er sonst keine Reihenfolge auf seinen Elementen sichtbar werden läßt. Die Klasse Traversable definiert zunächst das von der weiteren Differenzierung unabhängige Basisprotokoll, welches sich einfach durch die Tatsache ergibt, daß die Elemente in einer bestimmten Reihenfolge durchlaufen werden sollen, und daß immer eine aktuelle Position während dieses Prozesses existiert.

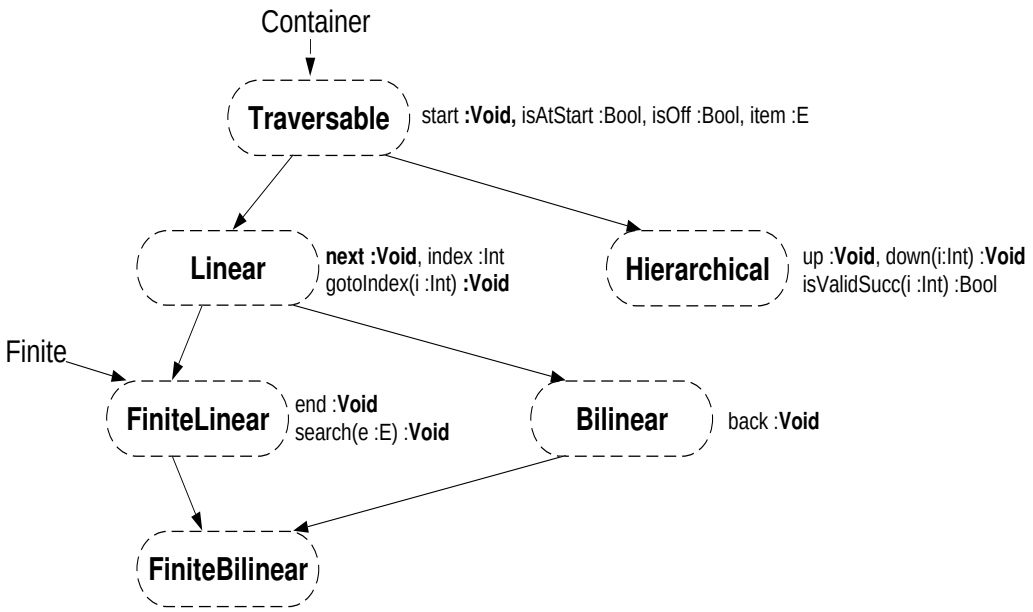

Abbildung 7.6: Eigenschaftsgruppe Traversierungsformen

Daraus ergeben sich die Methoden zum Aufsuchen der Startposition (start) und zur Überprüfung, ob diese Position eingenommen wurde (isAtStart). Weiterhin wird die Identifikation einer gültigen Position und der Zugriff auf ein Element an einer solchen Position über die Methoden isOff und item angeboten. Abbildung 7.6 gibt einen Überblick über die weitere Struktur der Eigenschaftsgruppe.

Die beiden Subklassen Linear und Hierarchical unterscheiden, wie die Namen schon sagen,

die Eigenschaft, ob dem Traversieren eine lineare oder eine hierarchische Strukturvorstellung zugrunde liegen soll. Lineares Traversieren findet sich typischerweise bei Listen, hierarchisches Traversieren ist für bestimmte Bäume vorstellbar.

Die Klasse Linear spezifiziert nun die Standardmethode next zur Einnahme der Folgeposition. Außerdem findet sich dort mit Methode index die Möglichkeit, die Position innerhalb der Reihenfolge der Elemente zu bestimmen und über gotoIndex auch eine solche, soweit sie vorhanden ist, anzusteuern. Damit ist auch für einen Behälter mit dieser Eigenschaft eine freie Positionierbarkeit möglich allerdings durch entsprechend teure Operationen, eine tatsächlich einseitig lineare Speicherungsstruktur der Elemente vorausgesetzt.

Das hierarchische Traversierungsprotokoll der Klasse Hierarchical bietet entsprechend der zugrunde liegenden Strukturvorstellung Methoden zum Auf- und Absteigen in der Hierarchie (up, down) und zur Überprüfung des Vorhandenseins bzw. der Gültigkeit von Nachfolgern (z.B. Methode isValidSucc). Dabei wird hier von vornherein vom Vorhandensein beliebig vieler (auch unterschiedlich vieler) Nachfolger innerhalb der Hierarchie ausgegangen. So definiert dann auch die Methode successorCount die Anzahl der zu einem bestimmten Zeitpunkt prinzipiell möglichen Nachfolger, die dann aber noch unerreichbar seien können.

Die Klasse Bilinear ergänzt das einfach lineare Traversieren zum bidirektional linearen Traversieren durch Einführung der Methode back, durch welche die Vorgängerposition der aktuellen Position betreten wird. Das Vorhandensein der (bei entsprechender Behälterstruktur eventuell direkt implementierten) Methode, ermöglicht schon in dieser Klasse die Optimierung der geerbten Methode gotoIndex zumindest für die Positionen zwischen Startposition und aktueller Position.

In FiniteLinear wird in Abweichung von der Baumstruktur die Eigenschaft des linear traversierbaren Behälters mit der Eigenschaft Finite aus der Gruppe der Aufnahmefähigkeitseigenschaften[3] kombiniert. Die Unterscheidung zwischen linearem und endlich linearem Traversieren wurde vorgenommen, um die Kombinierbarkeit des linearen Traversierens mit den 'unendlichen' Behältern zu ermöglichen. Methoden, bei denen die Gefahr einer Endlosschleife besteht wie z.B. die Methode search, welche die dem übergebenen Element entsprechende Position aufsucht, oder die die Endlichkeit zwingend voraussetzen wie die Methode end, welche die letzte Position der definierten Reihenfolge aufsucht, wurden in die Klasse FiniteLinear ausgelagert.

Die Klasse FiniteBilinear schließlich kombiniert lediglich noch die endlich lineare Eigenschaft mit der des bidirektionalen linearen Traversierens. Hier ist dann noch einmal eine Verbesserung der Methode gotoIndex möglich. Nun unter Berücksichtigung des maximalen Index, der durch die von Finite geerbte Methode size bekannt wird.

7.2.5 'Sortiertheit'

Die Eigenschaft der Sortiertheit soll für einen Behälter bedeuten, daß er seine Elemente immer sortiert zu halten hat. Diese Forderung ist natürlich nur für solche Behälter sinnvoll, die auch eine Reihenfolge für ihre Elemente definieren. Insofern sind die Eigenschaften dieser Gruppe nicht mit jedem Behälter kompatibel.

Für den Elementtyp eines sortierten Behälters wird eine Ähnlichkeitsbeziehung zum durch die Klasse Sorted definierten Objekttyp gefordert. Diese Klasse spezifiziert die Ordnungsrelation definierende Methoden zum Objektvergleich (z.B. "<" und ">"). Die so über den

[3] siehe Abschnitt 7.2.2

```
class Ordered
super Equality
metaclass AbstractClass
public methods
order(x :Self) :Int deferred
"="(x :Self) :Bool {order.(x) == 0}
"<"(x :Self) :Bool {order.(x) <  0}
">"(x :Self) :Bool {order.(x) >  0}
...
;
```

Abbildung 7.7: Ordnungsrelation für Objekte

Elementtyp auch für den Behälter erreichbaren Methoden können dann zur Implementierung der Sortierungserhaltung genutzt werden. Die Struktur der Eigenschaftsgruppe wird anhand der Abbildung 7.8 erkennbar. In der Wurzelklasse Sorted werden Methoden zur Erkundung der Sortierungsrichtung beschrieben(isSortedAscending, isSortedDescending).

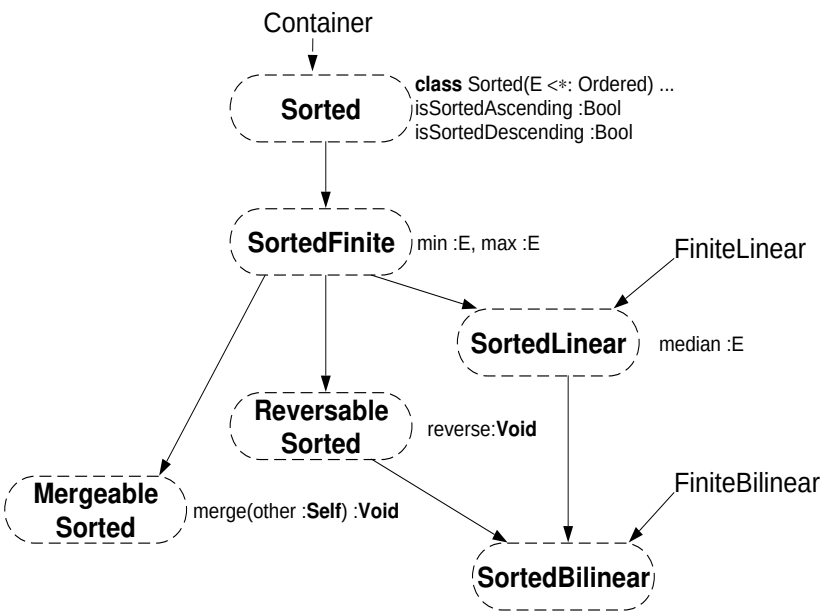

Abbildung 7.8: Sortierte Behälter

Der Nachfolger SortedFinite kombiniert die Eigenschaft des Sortiertseins mit der Endlichkeit. Dadurch werden die Methoden zur Berechnung des Minimums und des Maximums sinnvoll. Die drei Nachfolger dieser Klasse beschreiben verschiedene Varianten des Sortiertseins welche auch kombinierbar sind.

Die Subklasse MergeableSorted führt eine Methode zum Mischen zweier sortierter Behälter ein. Dieses die Sortierung erhaltende Mischen wird hier als die eigentliche Form des Mischens angesehen. Ein Zusammenfügen nach dem Reißverschlußverfahren etwa für Listen müßte an-

ders benannt werden.

Ein weiterer Nachfolger von SortedFinite ist ReversableSorted. Diese Klasse spezifiziert eine Methode zum Umschalten zwischen aufsteigender und absteigender Sortierung. Gleichzeitig implementiert sie die über die Sortierungsrichtung Auskunft gebenden geerbten Methoden mit Hilfe einer privaten Variablen, welche sich die aktuelle Sortierungsrichtung merkt.

Der dritte Nachfolger SortedLinear kombiniert endlich lineares Traversieren mit der Sortiertheitseigenschaft. Durch die Möglichkeit zum Traversieren wird bereits hier eine Reihenfolge der Elemente sichtbar. Dadurch sind die Methoden zur Berechnung von Minimum und Maximum implementierbar. Des weiteren wird eine Methode zur Medianberechnung angeboten.

Die letzte Klasse dieser Gruppe kombiniert die Eigenschaft der Umkehrbarkeit der Sortierrichtung mit dem endlich bilinearen Traversieren und steht somit für die Aussage, daß alle endlich bilinear traversierbaren Behälter auch die Umkehrung der Sortierrichtung unterstützen können.

7.2.6 Sortierbarkeit

Die Eigenschaft der Sortierbarkeit soll in Abgrenzung zu einem immer sortierten Behälter lediglich bedeuten, daß eine Sortiermethode angeboten wird. Dies soll auch dann möglich sein, wenn auf den Elementen des Behälters von vornherein keine Ordnungsrelation definiert ist. Ein Behälter, für dessen Elemente lediglich eine Gleichheitsrelation verlangt wird, kann auch mit Elementen belegt werden, für die eine Ordnungsrelation denkbar wäre. Bei einer solchen Belegung soll es möglich sein, den Behälter zwischenzeitlich zu sortieren. In der Wurzelklasse Sortable wird daher eine Sortiermethode spezifiziert, der man eine die Ordnungsrelation zwischen den Behälterelementen definierende Funktion übergeben kann. Die Angabe einer solchen Funktion ist, insbesondere wenn dem Behälter bei der Erzeugung ein geordneter Elementtyp übergeben wurde, durch einfache Abbildung auf die ordnenden Methoden der Elemente zu leisten.

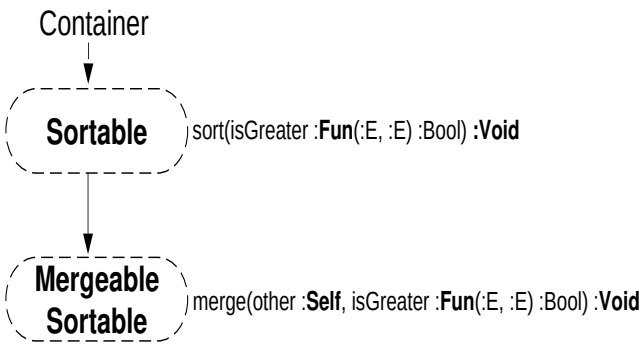

Abbildung 7.9: Sortierbarkeit und sortierungserhaltendes Mischen

Durch die Unabhängigkeit der Sortierbarkeit vom Elementtyp des Behälters kann die Sortiermethode bereits frühzeitig von Behältern geerbt werden, die eine Reihenfolge auf ihren Elementen sichtbar werden lassen. Diese Reihenfolge kann dann von der Sortiermethode verändert werden. Es ist dann nicht mehr nötig, zur Umsetzung dieser Eigenschaft in ei-

ner Behälterkategorie spezielle sortierbare Behälterklassen zu definieren, die nur sortierbare Elemente aufnehmen.

Für die Subklasse MergeableSortable treffen in Bezug auf das sortierungsstabile Mischen die gleichen Überlegungen zu, wie die für ihre Vorgaängerin angestellten Überlegungen bezüglich der Sortiermethode. Derartiges Mischen macht nur Sinn, wenn der Behälter auch sortierbar ist, also eine Sortiermethode anbietet, um zunächst überhaupt einmal eine Sortierung zu erhalten. Dies erklärt die Plazierung als Erbe von Sortable. Die angebotene Mischmethode hat dann auch die gleichen Parameter wie die geerbte Sortiermethode. Prinzipiell lassen sich beide Methoden durch die über den Funktionsparameter mögliche algorithmische Steuerung des Misch- bzw.Sortiervorganges auch zu beliebigen anderen Manipulationen der Reihenfolge nutzen, z.B. zur Erzeugung einer Sequenz von abwechselnd aus den Behältern entnommenen Elementen.

Die Behälterklassenbibliothek verwendet die Eigenschaften der Sortierbarkeit und der 'Sortiertheit' als sich gegenseitig ausschließende Eigenschaften. Dies geschieht aufgrund der Überlegung, daß das (Um)Sortieren der Elemente eines bereits sortierten Behälters in eine der Ordnung dieser Elemente zuwiderlaufende Reihenfolge nicht sinnvoll ist. Bei entsprechender Anpassung der Implementierung besonders der Einfügemethoden ist eine Kombination der beiden Eigenschaften allerdings möglich.

7.3 Behälterkategorien

Die bisherigen Abschnitte beschrieben die allgemeinen Behältereigenschaften und deren Klassifizierung. Im folgenden werden nun bestimmte Behälterkategorien betrachtet, welche in ihren Wurzelklassen allgemeine Behältereigenschaften kombinieren. Im Klassifizierungsprozeß von Abschnitt 7.1 entspricht dies dem Schritt 2. Den größten Raum nimmt in diesem Abschnitt jedoch die Darstellung der Schritte 3 und 4 des Klassifizierungsprozesses ein, welche die Aufteilung nach spezielleren Eigenschaften der Behälterkategorie und die Kombination dieser Eigenschaften zu konkreten spezialisierten Behältern bezeichnen. Diese Schritte sollen exemplarisch an den Kategorien der Listen und der Bäume verdeutlicht werden.

7.3.1 Listen

Abbildung 7.11 gibt einen Überblick über die Listenklassifizierung. Die Wurzelklasse der Listen AbstractList vereinigt die allgemeine Eigenschaften repräsentierenden Klassen Unbounded und Active. Dies bedeutet für die Listen, daß sie immer endlich viele Elemente enthalten, keine Kapazitätsgrenze haben und daß sie immer ein Element als aktuelles, zugreifbares Element zur Verfügung stellen. Die Wurzelklasse erhält auf diese Weise bereits ein einfaches Listenprotokoll das als einheitliche Sicht auf alle Listen verwendet werden kann. Zu den geerbten Methoden gehören zum Beispiel, wie auch in Abbildung 7.11 angedeutet, Methoden zum Löschen von Elementen, Einfügen von Elementen, für den Zugriff auf das 'aktivierte' Element, zum Löschen dieses Elementes und zur Bekanntgabe der aktuellen Listengröße.

Mit Hilfe dieser Methoden ließe sich beispielsweise eine für alle Listen anwendbare Methode zur Abarbeitung der Listenelemente definieren, wie sie in Abbildung 7.10 zu sehen ist. Die Spezifikation der Methode deleteItem in der Klasse Active, definiert als Nachbedingung:

ensure isOff => isEmpty.

Dadurch ist gewährleistet, daß alle Elemente der Liste erreicht werden.

```
class ListHandler(E <*: Equality)
super Object
public methods
workList(T <*: AbstractList(E), l :T,
          action :Fun(:E):Void) :Void
  {while(!l.isEmpty,action(l.item),l.deleteItem)}
...
;
```

Abbildung 7.10: Ausnutzung des Ähnlichkeitspolymorphismus

Für die Einfügemethode, die eigentlich schon in Klasse Active eingeführt wurde, wird nun in AbstractList semantisch ein Einfügen am Ende der Liste vorgeschlagen, um die Reihenfolge der Elemente in der Liste mit der Einfügereihenfolge in Übereinstimmung zu halten. Diese Forderung läßt sich allerdings mit den Mitteln von TooL nicht als Nachbedingung formulieren.

Die der Liste zugrundeliegende Vorstellung, daß die Elemente in einer bestimmten Reihenfolge verwaltet werden, ermöglicht die Einführung von Methoden zum Löschen der Kopien eines mehrfach auftretenden Elementes wie z.B. deletelthOccurence. Die abstrakte Liste ist aber noch nicht positionierbar.

Die Klassifizierung der spezielleren Listeneigenschaften führt zunächst zu fünf Nachfolgern. Sie unterscheiden:

1. positionierbare Listen
2. sortierte Listen
3. unsortierte Listen
4. gekettete Listen
5. rekursive Listen

In den folgenden Absätzen werden die einzelnen Klassen beschrieben und die Methodenzuordnung begründet.

Die Klasse PosList welche die Eigenschaft der Positionierbarkeit repräsentiert, führt das in FiniteLinear definierte endlich lineare Traversieren in die Kategorie der Listen ein. Da nun eine aktuelle Position sichtbar ist, können Methoden wie deleteLeft, deleteRight zum positionsbezogenen Löschen aufgenommen werden. Weiterhin wird zur Verwaltung der aktuellen Position eine private Variable eingeführt. Mit dieser und den geerbten Positionierungsmethoden können viele der spezifizierten Methoden bereits implementiert werden.

Die Klasse UnsortedList erbt die Sortier- und die Mischmethode aus MergeableSortable. Dies soll, wie schon in Abschnitt 7.2.6 allgemein beschrieben, ermöglichen, eine vorübergehende Sortierung der Liste zu erreichen, bzw. das sortierungsstabile Mischen solcher Listen durchzuführen. Durch die Verbindung von Unsortiertheit und der den Listen inhärenten Aufreihung ihrer Elemente können an dieser Stelle Methoden zum Einfügen am Anfang und am Ende der Liste angegeben werden (setStart, setEnd). Die Liste ist hier allerdings noch nicht

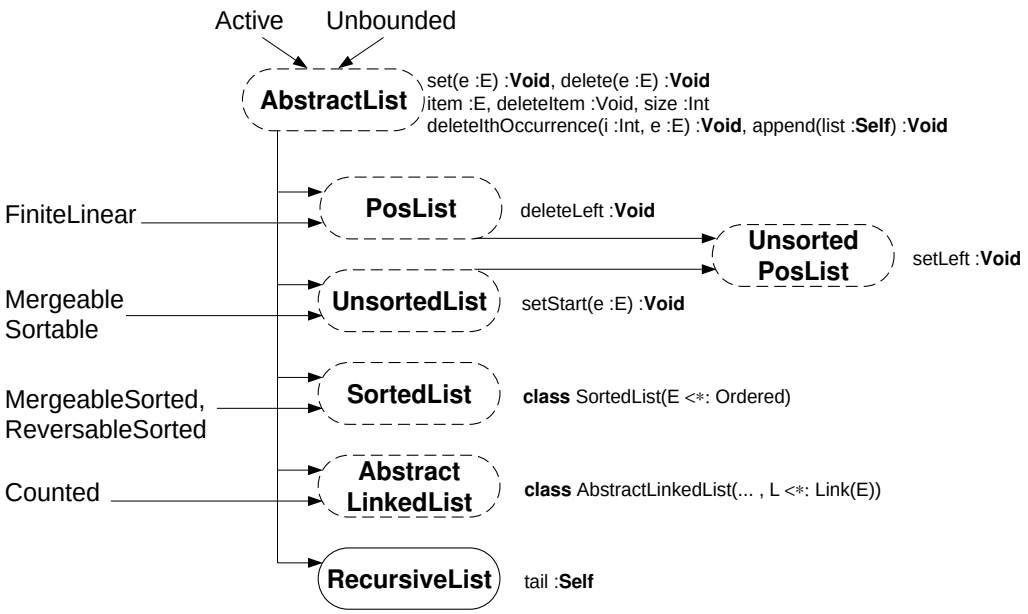

Abbildung 7.11: Die Behälterkategorie Liste

positionierbar, das heißt, die Auswahl des aktuellen Elements bezieht sich immer auf den Listenanfang. Ein anderes Element kann noch nicht direkt angesteuert werden.

Die Kombination der beiden zuvor genannten Eigenschaften erfolgt durch die Klasse UnsortedPosList. Das Sichtbarwerden einer Position innerhalb einer unsortierten Liste ermöglicht die weitere Spezialisierung des Einfügeprotokolls durch Einführung der Methoden zum positionsbezogenen Einfügen (setLeft, setRight). Außerdem kann die Sortiermethode hier mit Hilfe des speziellen Einfüge- und Positionierungsprotokolls implementiert werden.

Die Klasse SortedList ist die Wurzelklasse der sortierten Listen. Sie schränkt den Elementtyp entsprechend der Forderung ihrer Oberklassen MergeableSorted und ReversableSorted auf dem Typ Ordered ähnelnde Typen ein. Für alle sortierten Listen wird verlangt, daß sie sortierungsstabiles Mischen anbieten. Dies ist möglich, da durch die Eigenschaft Unbounded, die ebenfalls für alle Listen gilt, die Aufnahmefähigkeit für die zusätzlichen Elemente gegeben ist. Die Eigenschaft der Umkehrbarkeit der Sortierungsrichtung soll ebenfalls für alle sortierten Listen gelten. Dies kann, wie in den Nachfolgern von SortedList zu sehen, durch Umketten der Listenstruktur, falls eine gekettet Implementierung gewählt wurde, oder im Falle der Positionierbarkeit durch Flexibilisierung der Navigationsmethoden realisiert werden.

Durch die folgenden beiden Klassen werden zwei Implementierungseigenschaften (siehe Abschnitt 7.1) formuliert. AbstractLinkedList beschreibt die Implementierung als verkettete Zellen, welche in ihrer einfachsten Form durch die Klasse Link beschrieben werden, auf generische Art und Weise für alle Link-ähnlichen Zellen. Dazu wird ein weiterer Typparameter für die Klasse eingefügt, auf dem der Typ des aktuell verwendeten Links übergeben werden kann. Genaueres zu dieser Technik findet sich im Abschnitt 7.4.1.

Die zweite mit der Klasse RecursiveList angebotene Implementierungseigenschaft beschreibt eine rekursive Realisierung, in der die Liste aus einem Listenkopf mit einem Element und einer Restliste besteht. Der Vorteil von rekursiv definierten Listen liegt in der einfa-

chen Formulierung mächtiger Listenoperationen, sowie darin, daß diese Listen gemeinsame Teilbereiche haben können.

Die Definition einer solchen Liste im objektorientierten Umfeld der Behälterklassenbibliothek von TooL führt allerdings zu Problemen. So harmonieren die für solche Listen meist funktional spezifizierten Methoden nicht mit der Auffassung der meisten objektorientierten Methoden, welche ihre Manipulationen als Seiteneffekte auf das Objekt selbst durchführen. Die Signatur der geerbten Methode set zum Einfügen eines Elementes beispielsweise erlaubt nicht die Rückgabe einer entsprechend konstruierten neuen Liste, sondern intendiert den Einbau des Elements in den Empfänger der Nachricht. Dies verlangt bereits die Offenlegung bzw. Modifikationsfähigkeit für die Elementvariable und den Restlistzeiger der rekursiven Liste. Die Implementierung der Liste ist dadurch nicht mehr sauber gekapselt. Es besteht zum Beispiel die Möglichkeit, zyklische und damit unendliche Listen zu konstruieren. Die Schnittstelle bietet beide Arten von Methoden an: funktionale und imperative. Ein Methode zur Zyklusdetektion wird beigesteuert.

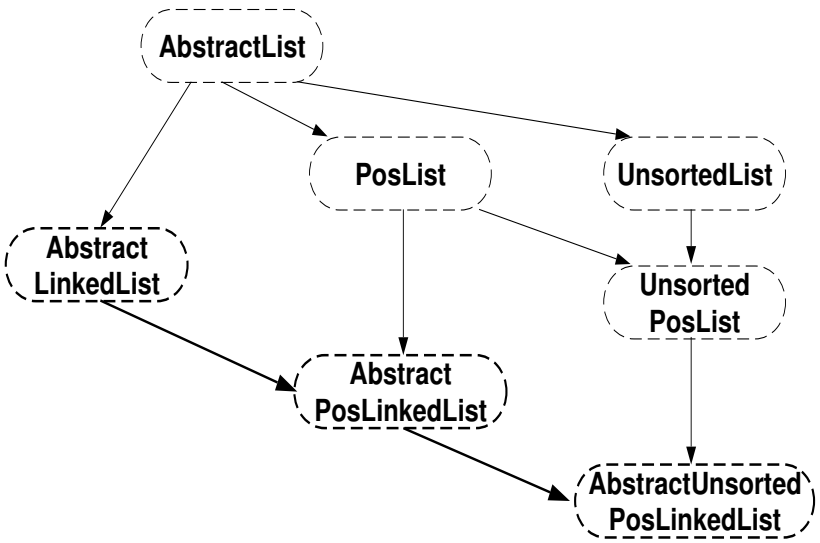

Abbildung 7.12: Kombination spezieller Listeneigenschaften

Der letzte Schritt des Klassifizierungsprozesses, wie er in Abschnitt 7.1 definiert wurde, beschreibt die Kombination der speziellen Eigenschaften einer Behälterkategorie zu konkreten implementierten Behältern. Während die Kombination solcher Eigenschaften, die keine Implementierungseigenschaften sind, noch zur Klassifizierung des speziellen Protokolls der Behälterkategorie gerechnet werden muß[4], entstehen bei Einbeziehung von Implementierungseigenschaften implementierte Klassen, die jeweils um das neu hinzukommende Protokoll erweitert bzw. an dieses angepaßt werden müssen.

Sofern die neu hinzukommende Eigenschaft nur ergänzendes Protokoll zur Verfügung stellt, müssen nur die nicht implementierten Methoden der Klasse realisiert werden, um die Methoden zu integrieren. Redefinitionen Effizienzgründen sind natürlich denkbar. Wird jedoch

[4]Die Klasse UnsortedPosList steht zum Beispiel für so einen Fall. Die Kombination der Eigenschaften in dieser Klasse führte zur Einführung von Methoden zum positionellen Einfügen.

ein neues Konzept geerbt, welches auch Einfluß auf die bereits vorhandenen implementierten Methoden hat, so müssen diese geändert werden. In vielen Fällen kann aber die Implementierung der Oberklassen genutzt werden indem, auf sie mittels des Aufrufs super.MethodenName Bezug genommen wird.

In Abbildung 7.12 wird eine entsprechende Situation für die Klasse AbstractLinkedList skizziert. Die Klasse ist zwar abstrakt, repräsentiert aber eine Implementierungseigenschaft, so daß an ihr und ihren Nachfolgern das Prinzip verdeutlicht werden kann[5]. Die Klasse realisiert also das aus AbstractList geerbte Basisprotokoll unter Verwendung verketteter Zellen. Die Erweiterung der Fähigkeiten der so implementierten Liste um die Eigenschaft der Positionierbarkeit findet man in AbstractPosLinkedList. Die Implementierung der Positionierbarkeit verlangt die Einführung eines weiteren Zeigers in die Kettungsstruktur, welcher die aktuelle Position markiert. Diese Ergänzung der Implementierung zwingt zur Korrektur der von AbstractLinkedList geerbten Methoden. So muß z.B. die Methode zum Einfügen eines Elementes set im Falle einer leeren Liste den neu eingeführten Zeiger auf das aktuelle Element sowie die Variable für den Index aktualisieren. In den anderen Fällen ist die geerbte Implementierung ausreichend und kann mit **super**.set(e) referenziert werden,wie in Abbildung 7.13 zu sehen.

```
class AbstractPosLinkedList(E <*: Equality, L <*: Link(E))
super AbstractLinkedList(E, L), PosList(E)
public methods
set(e :E) :Void
  ensure includes(e)
  {isEmpty ? {super.set(e), current := listStart, currentIndex := 0} : {super.set(e)}, nil}
...
;
```

Abbildung 7.13: Nutzung der geerbten Implementierung

Die Festlegung auf unsortiert positionierbare Listen hingegen, welche in AbstractUnsortedPosLinkedList vollzogen wird, bringt nur eine Protokollergänzung in Gestalt der Methoden zum positionsbezogenen Einfügen mit sich. Daher müssen hier auch nur die nicht realisierten Methoden dieses Protokollblocks der Implementierungsvariante entsprechend definiert werden.

Die Vererbungskette von AbstractLinkedList über AbstractPosLinkedList hin zu AbstractUnsortedPosLinkedList nutzt, wie in Abschnitt 7.1 bereits angedeutet, Implementierungsvererbung. Eine Änderung der Implementierungsstruktur in der Wurzelklasse muß zur Verifizierung der Nachfolger führen. Dies geschieht aber durch die Trennung der Implementierungseigenschaften von den konzeptuellen Protokollgruppen in diesem abgegrenzten Teilbereich des Vererbungsgraphen und ist somit beherrschbar.

7.3.2 Bäume

Die Behälterkategorie der Bäume beschreibt derzeit nur Suchbäume. Dies sind Bäume auf deren Elementen eine totale Ordnung definiert ist. Selbiges wird in der Wurzelklasse Ab-

[5]Die Klasse bleibt abstrakt durch den zusätzlichen Typparameter, welcher die generische Implementierung für unterschiedliche 'Links' ermöglicht bzw. durch die dadurch notwendig gewordene nicht realisierte private Methode für die Erzeugung eines Links. Siehe dazu auch Abschnitt 7.4.1

stractTree durch die Beschränkung des den Elementtyp beschreibenden Typparameters auf dem Typ Ordered ähnelnde Typen modelliert. Außerdem erbt die Klasse die zwei allgemeinen Behältereigenschaften Unbounded und SortedFinite. Die erste Eigenschaft entspricht der Feststellung, daß ein Baum im allgemeinen immer weitere Elemente aufnehmen kann, seine Aufnahmefähigkeit also nicht durch eine Kapazitätsgrenze eingeschränkt wird. Durch die zweite Eigenschaft werden die Methoden zur Feststellung von aufsteigender oder absteigender Sortierung sowie zur Berechnung von Minimum und Maximum der enthaltenen Elemente eingebracht. Darüber hinaus wird die Methode arity eingeführt, welche die maximal auftretende Anzahl von Nachfolgern eines Baumknotens zurück gibt.

Im folgenden werden drei spezielle Behältereigenschaften unterschieden von denen zwei eine Implementierungseigenschaft repräsentieren. Abbildung 7.14 verschafft einen Überblick, der auch die vorhandenen Kombinationen der speziellen Eigenschaften erfaßt.

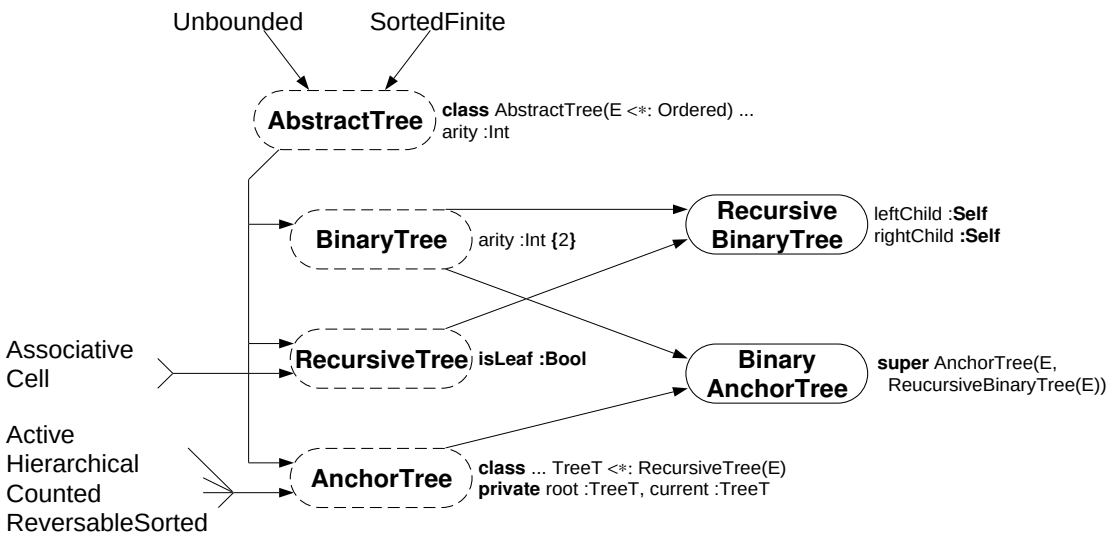

Abbildung 7.14: Bäume für geordnete Elemente

Da ist zunächst die Klasse BinaryTree. Sie steht für binäre Bäume, also solche mit höchstens zwei Nachfolgern eines Knotens, und implementiert die Methode arity entsprechend.

In RecursiveTree wird eine rekursive Implementierung des Baumes angenommen. Das bedeutet die Einheit von Baum und Baumknoten. Ein Baum ist ein Knoten mit zugeordneten Unterbäumen und ein Knoten ist die Wurzel eines Baumes. Die Klasse erbt daher von Cell eine Variable zur Aufnahme eines Elementes und aus Associative das Zugriffsprotokoll für den nur durch das Element selbst gesteuerten Zugriff. Es wird weder ein durch Schlüssel gesteuerter Zugriff angeboten, noch ein besonderes Element ausgezeichnet. Das Wurzelelement kann nicht als solches bezeichnet werden, da nicht direkt beeinflußbar ist welches Element dort zu finden ist. Der Knotenrolle entspricht die Einführung der Methode isLeaf. Die Nachfolgerzeiger werden noch nicht spezifiziert, da die Anzahl noch nicht feststeht. Denkbar wäre die Einführung eines Arrays noch nicht festgelegter Größe zur Aufnahme der Nachfolger. Dies würde einige weitere Implementierungen in dieser Klasse ermöglichen.

Durch AnchorTree wird eine spezielle Implementierung eingeführt, welche hierarchisches Navigieren auf einer rekursiven Baumstruktur ermöglicht. Dazu definiert sich die Klasse als

Anker für diese Struktur, welcher die Verwaltung der Navigation übernimmt. Die Klasse ist daher mit einem weiteren Typparameter ausgestattet, durch den die rekursive Struktur beschrieben wird. Für den übergebenen Typ wird daher eine Ähnlichkeit zum Typ RecursiveTree verlangt. Zugriffsprotokoll erbt die Klasse von Active, Traversierungsprotokoll wie vorgesehen aus Hierarchical. Das Vorhandensein von Navigationsmethoden ermöglicht die einfache Umkehrung der Sortierungsrichtung durch Umkehrung der Methodenfunktionalität, was die Erbbeziehung zu ReversableSorted erklärt. Darüber hinaus definiert die Klasse den Wurzelzeiger auf die Baumstruktur, sowie den Zeiger für den aktuell selektierten Knoten.

Die beiden abgebildeten implementierten Baumvarianten realisieren jeweils die Binärbaumversion für die zwei angebotenen Implementierungseigenschaften. In der Klasse RecursiveBinaryTree werden nun auch die Nachfolgerzeiger definiert. Dies muß, wie schon im die Listen beschreibenden Abschnitt ausgeführt, aufgrund der rekursiven Definition öffentlich oder mit Methoden gestütztem manipulierendem Zugriff geschehen. Der Baumaufbau ist ansonsten nicht möglich.

Die binäre Variante des Ankerbaumes belegt den die Baumstruktur beschreibenden Typparameter ihrer Oberklasse mit dem durch RecursiveBinaryTree beschriebenen Objekttyp und tritt somit als Nutzer dieser Klasse auf. Zur Verwaltung der Navigation ist bei nicht rückwärtig zum Elternknoten hin verknüpften Knoten ein Knotenstapel notwendig. Dieser wurde noch nicht in AnchorTree eingeführt um auch solche speicheraufwendigeren Knotentypen zuzulassen, für die dann diese Variable überflüssig wäre.

7.3.3 Weitere Behälterklassen

Verschiedene andere Behälterabstraktionen werden in der Klassenbibliothek angeboten. Bei den meisten wurde aber bisher keine so feine Klassifizierung wie bei den Listen durchgeführt.

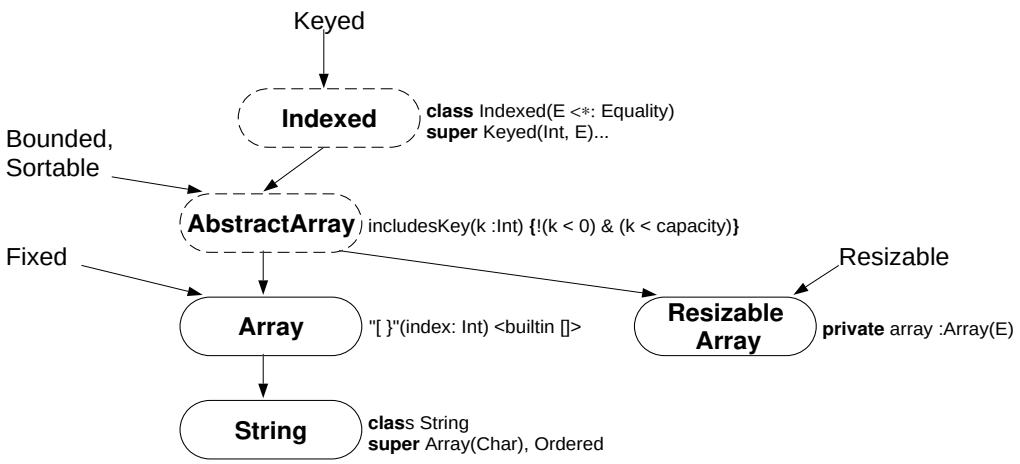

Abbildung 7.15: Arrays

Die Wurzelklasse der Behälterkategorie Array AbstractArray setzt für alle Felder die Eigenschaft fest, eine aktuelle Kapazitätsgrenze zu haben und sortierbar zu seien. Sie implementiert die Sortiermethode und legt den gültigen Bereich für die Schlüssel fest. Angeboten werden zwei Arrayvarianten. Es handelt sich um ein einfaches Array fester aber wählbarer Größe mit

entsprechender Erbbeziehung zur Klasse Fixed und um ein Array mit veränderbarer Kapazität entsprechend der Eigenschaft Resizable. Letzteres benutzt das einfache Array und schafft so die Möglichkeit verbunden mit entsprechendem Kopieraufwand zur Größenanpassung.

Bedarf für die Unterscheidung weiterer spezieller Behältereigenschaften bestand bisher nicht. Er könnte allerdings entstehen, wenn man die in Abschnitt 7.5 beschriebene flexiblere Elementvergleichsmethode einführen wollte. Diese würde prinzipiell durch eine private Variable implementiert, was für das Array nicht von Nachteil wäre. Der Array–Nachfolger String jedoch sollte möglichst ohne Variable auskommen, sodaß hier für die Arrays eine Differenzierung durchgeführt werden müßte. Abbildung 7.15 Zeigt die Arrayvarianten und den Nachfolger String.

Die Bibliothek bietet außerdem ein aus der Tycoon Modulbibliothek übernommenes und entsprechend angepaßtes, auf mehrdimensionale Bereichsanfragen spezialisiertes Wörterbuch[6]. Dieses stützt sich auf eine Verfeinerung des rekursiven, binären Suchbaumes zu einer mehrdimensionalen Struktur, genannt RangeTree[7].

Der besondere Elementtyp dieses Wörterbuches, nämlich der eines mehrdimensional geordneten Objektes, führt zur Einführung einer Nachfolgerin der Klasse Ordered, welche diese spezielle Sortierung beschreibt:

```
class MultiDimOrdered
super Ordered
metaclass AbstractClass
public methods
dimOrder(x :Self, dim :Int) :Int
  require dim <= maxDimension
  deferred
maxDimension :Int deferred
order(x :Self) :Int {privateOrder(x, maxDimension)}
...
private methods
privateOrder(x :Self, dim :Int) :Int
  {
  let result = dimOrder(x, dim),
  ((result != 0) | (dim = 1)) ? {result} : {privateOrder(x, dim−1)}
  }
```

Die Klasse MultiDimOrdered definiert entsprechend zur eindimensionalen Ordnung eine Methode dimOrder, welche nun die Ordnungsbeziehung zwischen zwei Objekten der Klasse dimensionsbezogen berechnet. Mit Hilfe dieser Methode, kann dann die geerbte Methode order, welche die nicht dimensionsbezogene Ordnung auf den Elementen berechnet, implementiert werden[8].

Das mehrdimensionale Wörterbuch verwendet nun den so spezifizierten Objekttyp zur Beschreibung seines Schlüsseltyps:

```
class DDimDictionary(K <*: MultiDimOrdered, E <*: Equality)
super Keyed(K, E), Unbounded(E), Counted(E)
...
```

[6] Eine ausführlich Beschreibung der nahezu schnittstellenidentischen TL Version findet sich in [Lotter, Römer 94]

[7] Eine Definition findet sich in [Mehlhorn 84]

[8] Dies geschieht durch Priorisierung der Dimensionen. Der durch maxDimension selektierten Dimension wird dabei die höchste Priorität zuerkannt.

Mit der Klasse HashTable wird ein Behälter angeboten, dessen Rolle als Behälter durchaus umstritten ist. So schreibt [Traub 95]: „Eine Hashtabelle ist kein Behälter. Sie ist eine Datenstruktur, die wir zur Implementierung von verschiedenen Behältern verwenden können." Dem hier verwendeten Klassifizierungsprozeß entsprechend kombiniert die Klasse zunächst die allgemeinen Eigenschaften Associative und Resizable. Die erste steht für die Zugriffsform, bei der nur das Element selbst zur Identifikation innerhalb des Behälters dient, die zweite für die einer Tabelle entsprechende begrenzte, aber erweiterbare Aufnahmefähigkeit. Die Kombination begründet einen neue Behälterkategorie. Es wird jedoch keine über das ererbte Protokoll hinausgehende Methode spezifiziert. Einzige spezielle Eigenschaft ist ein besonderes Implementierungsverfahren. Dieses wird daher direkt in die Wurzelklasse der Kategorie aufgenommen.

Der so geschaffene Behälter kann für sich oder zur Implementierung anderer Behälter genutzt werden. Es gibt daher keinen Grund, ihm seine Eigenständigkeit abzusprechen. Die Optimierung einer bestimmten Zugriffsmethode (hier des Tests auf Mitgliedschaft eines Elementes) stellt genauso eine fachliche, zur Auswahl des Behälters beitragende Eigenschaft dar wie beispielsweise die Sortierbarkeit.

7.4 Implementierung

Für die Wiederverwendbarkeit ist nicht nur die Gestaltung des Vererbungsgraphen entscheidend, sie wird auch durch geschickte Implementierungen verbessert. Die für Behälterklassen typische, generische, vom Elementtyp weitgehend abstrahierende Form ermöglicht die Wiederverwendung durch Benutzung. Es läßt sich aber durch Einführung von Typparametern in Klassen auch ein Nutzen für die Vererbung erreichen. In 7.4.1 werden zwei Klassen, die sich dieser Möglichkeit bedienen vorgestellt. Zwei weitere, die Wiederverwendbarkeit erhöhende Aspekte der Methodenimplementierung sind unter 7.4.2 und 7.4.3 beschrieben.

7.4.1 Implementierungsvarianten

Die Möglichkeit mehrere Implementierungsvarianten in einer Klasse zusammenzufassen, ergibt sich in der Behälterkategorie der Listen unter Ausnutzung der Ähnlichkeitsrelation. Die Realisierung einer Liste durch verkettete Zellen ist eine mögliche Konkretisierung der Listenidee. Es sind unterschiedlich komplexe Zellen denkbar. Die einfachste Variante sind einfach und doppelt verkettete Zellen. Implementierungen der beiden Varianten sind sich algorithmisch jedoch sehr ähnlich. Eine Zusammenlegung gelingt durch zwei Abstraktionsschritte:

- Generalisierung des Kettungsvorganges durch Bereitstellung entsprechender Methoden durch die Zelle selbst.
- Einführung eines Typparameters für den Zellentyp in der die abstrakte Listenimplementierung leistenden Klasse

In Abbildung 7.16 wird der erste Schritt angedeutet.

Die Klassen Link und DoubleLink repräsentieren die beiden Zellenvarianten. DoubleLink geht als Spezialisierung aus Link hervor. Letztere Klasse definiert bereits die Methoden zum Ein- und Ausketten sequentiell angeordneter Links. Die Methoden benötigen Parameter vom Objekttyp, wodurch im Zuge der Spezialisierung die Subtypbeziehung zwischen den beiden

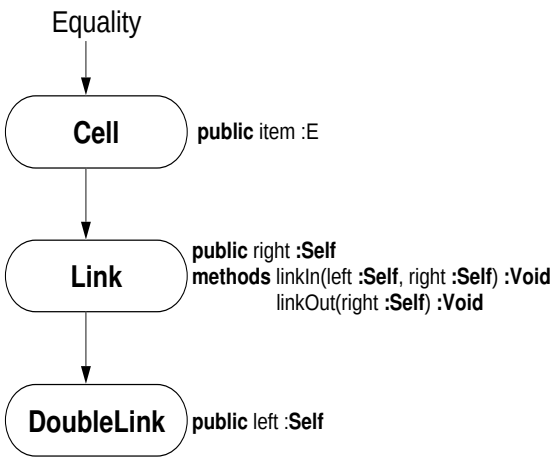

Abbildung 7.16: Kettungsabstraktion

durch die Klassen spezifizierten Objekttypen verloren geht. Dies geschieht aber auch schon durch die in der Klasse Link definierte öffentliche Variable vom Objekttyp die auf den Nachfolger zeigen soll. Die automatisch bereitgestellten Zugriffs- und Zuweisungsmethoden benötigen ebenfalls entsprechende Parameter. Die Öffentlichkeit der Variablen ist notwendig für die Kettenbildung und Navigation innerhalb einer Klasse, die Link-Objekte nutzt.

Schritt zwei der Abstraktion ist aufgrund des Verlustes der Subtypbeziehung in der gewohnten Weise nicht durchführbar. Erst die Ähnlichkeitsrelation bzw. die Einführung eines Typparameters,welcher die Ähnlichkeit zum ObjekttypLink fordert, ermöglicht die Formulierung polymorpher Algorithmen für die Link-Objekte und ihrer Spezialisierungen, wie es in der Klasse AbstractLinkedList geschieht:

```
class AbstractLinkedList(E <*: Equality, L <*: Link(E))
super AbstractList(E)
metaclass AbstractClass
public methods
item :E {listStart.item}
set(e :E) {... linkIn(listEnd, newLink)...}
...
private listStart :L, listEnd:L  ...
methods
initialLink :L deferred
;
```

Der übergebene Typ wird zur Spezifikation der privaten Zeiger auf den Anfang und das Ende der Liste genutzt. Methoden des aus AbstractList geerbten Basisprotokolls für Listen, wie z.B. set, können nun unter Verwendung der Kettungsfunktionalität der Links realisiert werden. In Ermangelung einer Möglichkeit, den über den Typparameter erhaltenen Typ zur Objekterzeugung zu nutzen, wird dafür noch eine private Methode benötigt, die die Erzeugung übernimmt. Sie ist schließlich die einzige nicht implementierte Methode der Klasse. Die Benutzung der Klasse wäre durch die notwendige Instantiierung mit einem Link nicht so

komfortabel. Die beiden einfachen Listenvarianten LinkedList und DoubleLinkedList lassen sich jedoch einfach durch Subklassenbildung beschreiben:

```
class DoubleLinkedList(E <*: Equality)
super AbstractLinkedList(E, DoubleLink(E))
metaclass ListClass(E, DoubleLinkedList(E))
private methods
initialLink :L {DoubleLink(:E).new}
;
```

Der Typparameter der Oberklasse AbstractLinkedList wird einfach mit dem gewünschten 'Linktyp' belegt, und die Erzeugungsmethode kann dann unter Rückgriff auf die new-Methode der entsprechenden 'Linkklasse' angegeben werden.

Dieses Beispiel leitet auch die zweite Variante einer durch die Ähnlichkeitsrelation ermöglichten polymorphen Klassenimplementierung ein. Die Metaklasse ListClass ist universell für alle Listen einsetzbar. Sie stellt listenspezifische Methoden zur Objekterzeugung zur Verfügung. Aufgrund der binären Methoden die bereits in Container mit der Methode "=", aber auch in AbstractList selbst mit der Methode append eingeführt wurden, besteht auch für die Objekttypen der Subklassen von AbstractList keine Subtypbeziehung zum Objekttyp der Listenwurzel. Daher wird auch hier für die Typparameter der Metaklasse die Ähnlichkeitsrelation benötigt:

```
class ListClass(E <*: Equality, Instance <*: AbstractList(E))
super Class(Instance)
public methods
singleton(e :E) :Instance
  {list1(e)}
list1(e1 :E) :Instance
   {let result = new, result.set(e1), result}
list2(e1 :E, e2 :E) :Instance
  {let result = list1(e1), result.set(e2), result}
...
;
```

Der über Instance beschriebene Listentyp wird als Rückgabetyp der speziellen Objekterzeugungsmethoden genutzt. Durch die Belegung des Typparameters der Oberklasse Class mit eben diesem Objekttyp kann auch ein Objekt dieses Typs mittels der geerbten Methode new erzeugt werden. Durch die Ähnlichkeitsbeziehung zum Objekttyp AbstractList(E) ist gewährleistet, daß die verwendete Operation set von diesem Objekt auch unterstützt wird.

7.4.2 Vermeidung von Redefinitionen

Dieser Abschnitt beschäftigt sich mit der Frage, inwieweit die Methodenredefinition im Zuge von Implementierungsvererbung vermieden werden kann. Wie schon im Abschnitt 7.3.1 geschildert, führt die eigenschaftsbezogene Aufteilung des Protokolls bereits zu einer Verringerung der notwendigen Überschreibungen oder doch mindestens zu einer Vereinfachung des bei der Redefinition zu formulierenden Programmsegments. Die Funktionalität der geerbten Methode kann in vielen Fällen genutzt werden und muß nur noch an die neue Eigenschaft angepaßt werden. Dies kann meist auch ohne intensives Studium der geerbten Implementierungen erfolgen.

Es kann jedoch auch durch voreilige algorithmische Entscheidungen eine sonst nicht notwendige Redefinition ausgelöst werden. Das heißt, wenn die Oberklasse bei der Implementierung einer Methode bestimmte Annahmen über die Struktur des Behälters macht, die dann in der Subklasse nicht mehr zutreffen, zwingt dies die Subklasse zu einer Neudefinition. Derartige Probleme mit den geerbten Implementierungen sind vom Subklassenkonstrukteur schwer zu entdecken. Entweder müssen die getroffenen Annahmen daher sichtbar gemacht werden, dies ist in TooL z.B. durch Klasseninvarianten möglich, oder die Realisierung von Methoden erfolgt bereits unter Vermeidung solcher Annahmen.

Als Beispiel diene die Klasse AbstractPosLinkedList, welche das Protokoll einer positionierbaren Liste über verkettete Zellen realisiert. Sie implementiert unter anderem die Methode isAtEnd zur Identifikation der Endposition der Liste:

```
class AbstractPosLinkedList(E <*: Equality, L <*: Link(E))
super PosList(E), AbstractLinkedList(E, L)
public methods
isAtEnd :Bool
  {(current !== nil) && }current.right == nil}}
...
;
```

Dies geschieht durch Überprüfung des rechten Nachbarn der aktuellen Zelle. Ist er nicht vorhanden, so geht die Methode davon aus, daß die Endposition erreicht ist[9]. Die Annahme, daß es keinen rechten Nachbarn für die letzte Zelle der Liste gibt, könnte aber durch eine Subklasse relativiert werden, etwa wenn diese Listen als Teillisten anderer, größerer Listen betrachtet. Eine solche Annahme müßte also eigentlich als Klasseninvariante formuliert werden. Besser ist allerdings die direkte Implementierung durch Vergleich mit dem ebenfalls vorhandenen Zeiger auf das Listenende:

```
class AbstractPosLinkedList(...)
...
isAtEnd :Bool
  {(current == listEnd) & (current !== nil)}
...
;
```

Dadurch kann eine spätere Überschreibung der Methode verhindert werden. Der Grundsatz sollte also sein, sich beim Realisieren der Methoden nur auf die innerhalb der Klasse sichtbaren Informationen zu stützen und möglichst keine darüber hinausgehenden Annahmen zu verwenden.

7.4.3 Erweiterbare Methodenketten

Dieser Abschnitt beschreibt ein weiteres Verfahren, welches dem Subklassenkonstrukteur für bestimmte Methoden den Rückgriff auf bereits realisierte Funktionalität ermöglicht. Den Ansatzpunkt bieten insbesondere Methoden, die bereits weit oben im Vererbungsgraphen spezifiziert und auch implementiert wurden, deren Aufgabenbereich sich aber immer wieder ändert.

[9]Das doppelte '&' Symbol benennt die 'lazy evaluation' Version der Und-Verknüpfungsmethode, deren zweiter Parameter (eine Funktion ohne Parameter, gekennzeichnet durch die geschweiften Klammern) nur ausgewertet wird, wenn der erste zu true evaluiert.

Ein einfaches Beispiel für diese Situation ist die Initialisierungsmethode init. Sie wird bereits in der Klasse Object realisiert.

```
class Object
metaclass AbstractClass
public methods
init :Self {self}
...
;
```

Die für alle Klassen voreingestellte Metaklasse Class stößt dann die Initialisierung innerhalb der von ihr definierten new-Methode an, sodaß die Methode letztlich bei jeder Objekterzeugung aufgerufen wird. Sie soll dazu dienen, die von den Klassen definierten Variablen individuell mit Anfangswerten zu belegen. In Object sind noch keine Variablen definiert, sodaß die Methode hier nur das unveränderte Objekt zurück gibt. Sobald eine Klasse jedoch Variablen definiert, wird sie die Methode init überschreiben und die Variablen mit den vorgesehenen Werten belegen. Damit nun nicht jede Klasse erneut alle ererbten Variablen initialisieren muß wird eine solche Redefinition auf die Implementierungen ihrer Vorfahren zurückgreifen, indem sie diese mit **super**.init anspricht. Auf diese Weise entsteht eine Kette von **super**-Aufrufen, welche die Initialisierungsfunktionalität auf den Vererbungsgraphen verteilt.

Eine Verlängerung der Kette ist jederzeit möglich und auch Änderungen innerhalb des Vererbungsgraphen, etwa durch Hinzufügen von weiteren Variablen, bleiben zumindest für die init-Methoden der Nachfolger ohne Auswirkungen. Der generelle Bedarf nach Revision der Nachfolger bei solchen Modifikationen wird dadurch natürlich nicht hinfällig.

Das Prinzip der Funktionalitätsverteilung läßt sich auch für andere Methoden umsetzen, wenn diese von vornherein darauf angelegt werden. Für die Methode zur Beurteilung der Gleichheit zweier Behälter, wie sie in der Klasse Container durch die Erbbeziehung zur Klasse Equality für alle Behälter gefordert wird, wurde dieser Ansatz umgesetzt.

Als Grundlage der Implementierung dient die Definition, daß zwei Behälter als 'gleich' anzusehen sind, wenn sie einerseits die gleichen Elemente enthalten, aber andererseits auch gleiches Verhalten bei einer beliebigen Befehlsreihenfolge zeigen. Um dies sicherzustellen, muß beim Vergleich der Elemente sobald vorhanden auch die Behälterstruktur verglichen werden. Außerdem sind besondere Eigenschaften zu vergleichen, wie z.B. bei positionierbaren Behältern die aktuelle Position. Auch der Vergleich der Größen sollte, sobald dieser ohne alle Elemente zu durchlaufen möglich ist, durchgeführt werden, um die Entscheidung zu beschleunigen.

Die Bildung einer Kette von **super**-Aufrufen ist aber für diese Methode nicht so einfach zu erreichen. Vergleichsmethoden die alle Elemente der beiden Behälter vergleichen, um dann eine geerbte Implementierung anzusprechen, die das gleiche tut, sind natürlich indiskutabel. Die Lösung liegt in der Aufteilung der Methode in zwei Methoden. Die erste und weiterhin mit '=' benannte Methode führt den Elementvergleich durch. Sie ruft dann die zweite, private Methode mit Namen privateEqual auf, welche die weiteren Eigenschaften der Behälter vergleicht. Diese zweite Methode kann dann lokal definiert werden, daß heißt sich nur auf die neu eingeführten Eigenschaften der Klasse zu beziehen, und dann die anderen lokalen Tests durch den Aufruf von **super**.privateEqual anzustoßen. Für die den Element- bzw. Strukturvergleich durchführende Methode gilt, daß sie überall dort neu definiert wird, wo eine neue Struktur eingeführt wird und effizient verglichen werden kann.

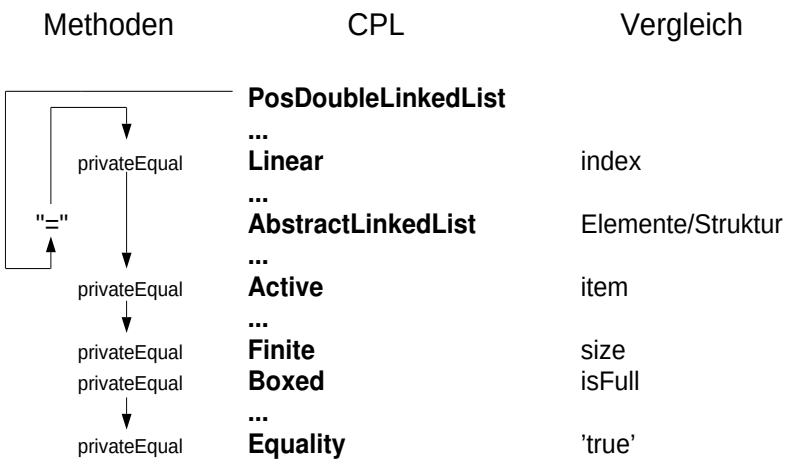

Abbildung 7.17: Funktionalitätsverteilung für die Methode '='

Für den Subklassenentwurf hat die eingeführte Methodik den Vorteil, daß er sich in den meisten Fällen auf die geerbte Funktionalität stützen kann und den Vergleich nur inkrementell um die neu eingebrachten Eigenschaften ergänzen muß. Dies kann der Entwickler einfach durch Überschreiben der Methode privateEqual tuen.

In Abbildung 7.17 wird die Verteilung der Funktionalität auf die verschiedenen Vorgänger der Klasse PosDoubleLinkedList skizziert. In der Mitte ist die Klassenpräzedenzliste für diese Klasse abgebildet, wobei nur Klassen berücksichtigt wurden, die auch durch den Aufruf der Vergleichsmethode angesprochen werden. Die linke Seite stellt dar, welche Methoden in den Klassen beteiligt sind und auf der rechten Seite wird angedeutet, welchen Vergleich sie ausführen.

Der Aufruf der Methode '=' wird durch die Implementierung in der Klasse AbstractLinkedList entgegengenommen. Dort wird zunächst der Identitätstest durchgeführt. Anschließend erfolgt der Aufruf von privateEqual. Der teurere Vergleich der Elemente bzw. der Struktur sollte zuletzt vollzogen werden. Zu beachten ist noch, daß die zuletzt aufgerufene Methode der privateEqual-Kette in Equality immer den Wahrheitswert 'true' evaluiert. Dies entspricht der Interpretation, daß, wenn keine Eigenschaften zu vergleichen sind, die Objekte gleich seien müssen. Würde an dieser Stelle eine Implementation verwendet, die auf Identität testete, so würde dies der Behauptung entsprechen, daß nur identische Objekte auch gleich sind.

7.5 Generische Behälterklassen

Wie schon an mehreren Stellen deutlich geworden, sind alle Behälterklassen in TooL durch den Elementtypparameter generische Klassen. Im Umfeld des besonderen Typsystems von TooL hat Art und Weise der Definition dieses Parameters (z.B. Subtyp- oder Ähnlichkeitsbindung) Auswirkungen auf die Benutzbarkeit der Behälterklassen oder beeinflußt die Form, in der Methoden in den Klassen definiert werden müssen (siehe auch [Gawecki, Matthes 96]). In diesem Abschnitt werden einige Varianten von Klassentypparametern zur Beschreibung des Elementtyps einer Klasse auf ihre Vor- und Nachteile hin untersucht.

In der derzeitige Version der Behälterklassenbibliothek verlangt der Elementtypparameter

vom übergebenen Typ eine Ähnlichkeitsbeziehung zur Klasse Equality. Dadurch wird sichergestellt, daß jedes Element eines Behälters eine Methode zur Feststellung der Gleichheit zu einem anderen Element versteht. Diese Methode wird in den Behälterklassen als Vergleichsmethode für alle Elementvergleiche genutzt. Die Frage ob nur die Identität oder andere Bedingungen die Identifikation von Elementen entscheidet, wird auf die Elemente selbst, bzw. die Implementierung ihrer Vergleichsmethode abgewälzt.

Eine weitere Konsequenz wird sichtbar, wenn man versucht in einen mit dem Typ Point aus Abbildung 7.18 instantiierten Behälter einen in der gleichen Abbildung beschriebenen ColoredPoint einzufügen. Dieser zweite Objekttyp ist ein Subtyp des Instantiierungstyps und man würde unter Anwendung des Subtyppolymorphismus erwarten, daß die Zuweisung erlaubt ist. Für den Elementtyp des Behälters wird jedoch eine Ähnlichkeitsbeziehung zu Equality verlangt. Diese existiert aber nicht für den Typ ColoredPoint, da in ihm, wie auch in Abschnitt 6.5 erläutert, alle auftretende **Selfs** durch den Typ Point ersetzt wurden. Es existieren daher keine korrespondierenden **Selfs** zu den Definitionen in Equality mehr.

```
Self <: class Point
super Equality
public x :Int, y :Int methods
"="(e :Self) :Bool {(x == e.x)&(y == e.y)}
;
class ColoredPoint
super Point
public c :String methods
;
```

Abbildung 7.18: In Subtypbeziehung stehende Elementtypen

Die Existenz derselben war jedoch Bedingung für die Erfüllung der Ähnlichkeitsrelation. Es scheint also so, als ob diese Variante von Elementtypparameter heterogene Behälter verhindert.

Dies trifft aber nur für die direkte Verwendung eines Behälters zu. Durch Definition einer einfachen neuen Subklasse für den gewünschten Behälter, läßt sich die Heterogenität des Behälters in Bezug auf die Subtypen des Instantiierungstyps wieder herstellen:

```
class PointArray
super Array(Point)
metaclass ArrayClass(Point, Array(Point))
public methods
;
```

Die Klasse PointArray belegt bei der Angabe ihrer Oberklasse den Elementtypparameter des in der Behälterklassenbibliothek definierten Arrays einfach mit dem Wurzeltyp der gewünschten Subtyphierarchie. Der so erhaltene Behälter hat die gleiche Funktionalität wie die Bibliotheksversion und ermöglicht die erwarteten Zuweisungen:

```
| > define arr :PointArray;
| > arr := PointArray.new(10);
| > arr[0] := Point.new;
| > arr[1] := ColoredPoint.new;
```

Eine zweite Variante zur Beschreibung des Elementtyps eines Behälters versucht den Wunsch nach Heterogenität direkt durch die Parameterwahl zu modellieren [Gawecki, Matthes 96, S.16]. Sie definiert dazu ebenso wie die zuvor geschilderte Methode einen Typparameter, der eine Ähnlichkeit zum Typ Equality verlangt. Dieser wird aber ergänzt durch einen weiteren, den eigentlichen Elementtyp beschreibenden Parameter, für welchen nun eine Subtypbeziehung zum auf dem ersten Parameter übergebenen Typ gefordert wird:

```
class Container(T <*: Equality, E <: T)
...
;
```

Die direkte Anpassung des Behälters an die Nutzungssituation könnte dann erreicht werden, indem der erste Parameter beispielsweise mit Typ Point der zweite mit Typ ColoredPoint belegt wird.

Problematisch ist allerdings die für den Elementtyp E verlangte Subtypbeziehung zu T. Sie führt bei Methoden, deren Rückgabetyp mit **Self** spezifiziert wurde, zur Ersetzung diese Typs durch den mit T bezeichneten Typ. Dies liegt an der derzeitigen Implementierung der Subtypbeziehung für den Typ **Self** im Typüberprüfungsalgorithmus von TooL, welcher diese Ersetzung vor Verifizierung der Subtypbeziehung vornimmt.

Denkbar wäre hier auch, daß ein Typ **Self** in kovarianter Position der Spezialisierung in der Subklasse folgen darf, wie dies ja auch in der allgemeinen Subtypregel für Funktionstypen formuliert ist. Die Ersetzung durch den Supertyp bewirkt jedoch, daß z.B. die Methode copy, welche ein Element vom Typ E aus der Klasse Object geerbt hat, statt des Typs E nunmehr den Rückgabetyp T hat. Wird diese copy-Methode nun innerhalb eines Behälters genutzt, etwa um eine vollständige Kopie des Behälters anzufertigen, so kann das Ergebnis der Elementkopie nicht mehr auf eine Variable vom Elementtyp des Behälters zugewiesen werden:

```
class Array(T <*: Equality, E <: T)
...
copy :Self
  {..., copiedArray[i] := self[i].copy,...}  !! → Typfehler !!
;
```

Für die dritte Variante lassen sich die Überlegungen zur zweiten Variante übernehmen. Sie führt jedoch die Flexibilisierung des Elementvergleichs ein und soll daher hier auch skizziert werden.

In dieser Version wird für den Elementtyp nur noch die Subtypbeziehung zum Typ Object verlangt. Die Methoden zum Elementvergleich, zur Umdefinierung der Gleichheitsrelation und zum Auslesen der aktuellen Relation werden in der Klasse Container spezifiziert. Die Implementierung könnte über eine private Funktionsvariable zur Aufnahme der die Gleichheit entscheidende Funktion erfolgen. Die Zugriffsmethoden für diese Variable lassen eine individuelle Anpassung des Zustandes zu, in dem eine Änderung der Gleichheitsrelation vorgenommen werden darf:

```
class Container(E <: Object)
...
public methods
equal(e :E, e1 :E) :Bool
  {equalityRelation[e, e1]}
```

```
setEqualityRelation(rel :Fun(:E, :E):Bool) :Void
  require isEmpty
  {equalityRelation := rel}
getEqualityRelation :Fun(:E, :E):Bool
  {equalityRelation}
...
private
equalityRelation :Fun(:E, :E):Bool
;
```

In Eiffel wird eine ähnliche Form vorgeschlagen [Meyer 94a, S.325]. Da Eiffel keine Funktionen höherer Ordnung bzw. Funktionsvariablen kennt, wird die Steuerung über eine 'boolesche' Variable durchgeführt, welche festlegt, ob auf Gleichheit oder auf Identität verglichen werden soll. Jede Methode die einen Objektvergleich ausführen will, muß zwei Varianten des Vergleichs in Abhängigkeit des Wertes der Steuerungsvariablen implementieren.

Die hier vorgestellte Technik ist darüber hinaus völlig unabhängig von der Gleichheitsmethode des Elementes. Beliebige Modifikationen der Gleichheitsrelation sind möglich.

7.6 Iterationsabstraktion

Unter Iterationsabstraktion soll hier die Möglichkeit verstanden werden, die Elemente eines Behälters in einer von der Behälterart unabhängigen Art und Weise zu durchlaufen. Es werden innerhalb der Bibliothek drei Varianten angeboten. Zwei davon sind bereits in der Wurzelklasse des Vererbungsgraphen Container sichtbar. Die dritte ergibt sich aus der Aufteilung der allgemeinen Behältereigenschaften in Eigenschaftsgruppen. Eine Iteration kann folgendermaßen durchgeführt werden:

- ▷ Nutzung des von jedem Behälter zur Verfügung gestellten Elementstromes
- ▷ Verwendung der behältereigenen Traversierungsfunktionalität, sofern diese vorhanden ist
- ▷ Einfache Iteration über alle Elemente mittels der von allen Behältern angebotenen Methode do

Die eigentliche Iterationsabstraktion und die mächtigste Funktionalität bieten die unter der Wurzelklasse Stream zusammengefaßten verschiedenen Varianten von Objektströmen. Abbildung 7.19 bietet eine Übersicht auf die verschiedenen Stromkonzepte.

Die Schnittstelle zwischen Behältern und Objektströmen stellt die in Container eingeführte Methode elements dar (siehe Abbildung 7.3). Die Applikation dieser Methode verlangt von jedem Behälter die Erzeugung eines Stroms von Objekten seines Elementtyps. In der Klasse ReadStream, durch welche der Rückgabetyp von elements beschrieben ist, werden zunächst nur wenige Methoden angeboten. Der Rückgabetyp kann jedoch durch die Subklassen von ReadStream spezialisiert werden. Dies ist möglich, da die Spezialisierung innerhalb der Stream-Hierarchie subtyperhaltend durchgeführt wurde.

Die Implementierung erfolgt über die nicht abstrakten Nachfolger von Stream, welche durch den Präfix Internal gekennzeichnet sind. Sie spezifizieren ein einfaches, dem Spezialisierungsgrad angemessenes Basisprotokoll, dessen Realisierung durch Initialisierung der bereitgestellten Funktionsvariablen mit der richtigen Implementierung erreicht wird.

Das in der Eigenschaftsgruppe der Traversierungsformen zusammengefaßte Protokoll wird von unterschiedlichen Behältern geerbt. Dadurch wird die Möglichkeit zum Traversieren zu einer Eigenschaft des Behälters selbst. Eine Trennung zwischen Behälterzustand und Navigationszustand besteht nicht. Daher stellt diese Form des Traversierens eigentlich keine Iterationsabstraktion dar, weil sie den Zustand des Behälters direkt beeinflußt. Es besteht deshalb im allgemeinen auch nicht die Möglichkeit, mehrere Iterationen dieser Art gleichzeitig durchzuführen.

Die Einführung einer Methode do in der Klasse Container scheint redundant zu seien, da entsprechende Funktionalität bereits über die Streams angeboten wird. Durch diese Methode soll jedoch eine einfache, optimierbare Möglichkeit der Iteration über alle Elemente angeboten werden. Diese kann dann unter Vermeidung der eventuell komplexen Stream-Implementierung durch direkten Zugriff auf die Repräsentation realisiert werden.

7.6.1 'Stream'-Varianten

Die Klasse Stream spezifiziert die Methode isAtEnd, durch welche ein Versiechen des Stroms erkannt werden kann. Die Einführung einer Methode, die die Größe des aktuellen Streams bekannt gibt, wurde vorgenommen, obwohl diese Information eventuell nicht beschaffbar ist (z.B. für einen unendlichen Stream). Es kann aber auch sein, daß die Beschaffung des Wertes zu aufwendig ist, etwa wenn der Strom für die Berechnung verbraucht werden müßte. Es wird daher von vorneherein auch der Rückgabewert **nil** für eine unbekannte Größe des verbleibenden Stroms zugelassen.

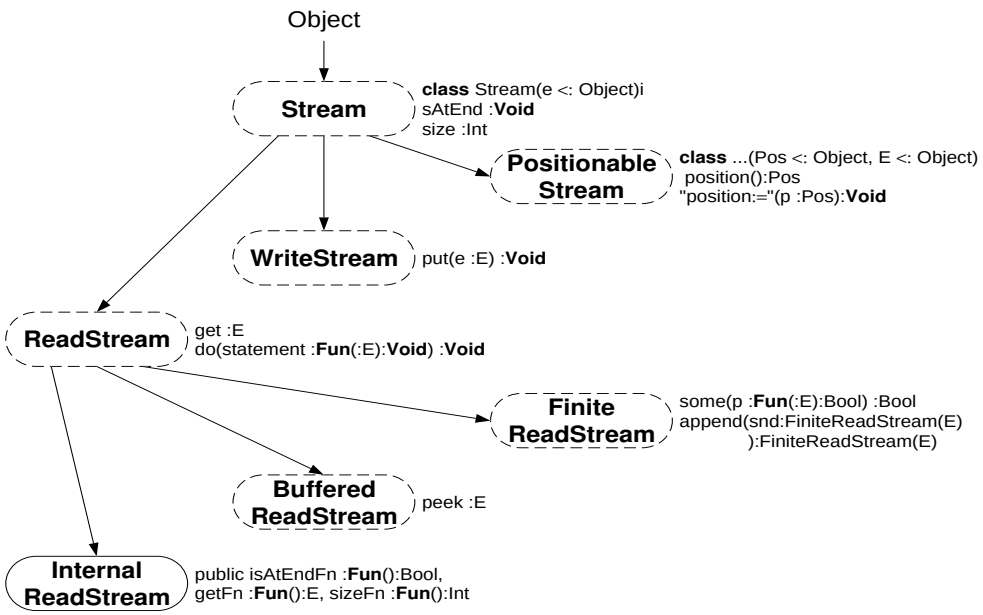

Abbildung 7.19: Stream-Varianten

Im folgenden werden drei Stromeigenschaften unterschieden. Der lesbare Strom ReadStream definiert die Basismethode für den Elementzugriff und das Ansteuern des nächsten Elementes get, sowie weitere Methoden zur Iteration und zur Abbildung auf einen anderen

Elementtyp (do und map). Der beschreibbare Strom WriteStream bietet mit der Methode put das Gegenstück zu get. Sie modifiziert ein Element und geht weiter zum Nächsten. Der positionierbare Strom führt einen zusätzlichen Typparameter in die Klasse ein, der den Positionstyp beschreibt. Über die Methoden position und position:= kann dann die Position abgefragt bzw. eine bestimmte Position innerhalb des Stroms angesteuert werden.

Für lesbare Ströme werden drei weitere Varianten unterschieden. Der endliche lesbare Strom FiniteReadStream bietet weitere Methoden zur Bearbeitung von Strömen an, die erst durch die Annahme der Endlichkeit sinnvoll werden. So sind dort beispielsweise prädikative Aussagen über einen Strom zu finden wie die Methode some, welche feststellt, ob eine Element innerhalb des Stroms existiert, daß das übergebene Prädikat erfüllt. Ein anderer Bereich beschäftigt sich mit der Reduktion von Elementen auf einen Wert oder mit der Verknüpfung von Strömen, wie die Methode (append) der in Abbildung 7.19 abgebildeten Klasse.

Anhand dieser letzten Methode append in FiniteReadStream sind die Konsequenzen der subtyperhaltenden Spezialisierung innerhalb der Stream-Hierarchie zu erkennen. Die Parameter und der Rückgabetyp der Methode sind auf den Objekttyp der Klasse festgelegt. Dies bedeutet, daß bei der Nutzung einer solchen Methode für speziellere, z.B. durch Kombination mit den anderen Eigenschaften entstandene Ströme das verfeinerte Protokoll über die Parameter nicht sichtbar ist und daß als Ergebnis der Methode auch nur ein FiniteReadStream zur Verfügung steht. Die Implementierung einer Methode kann daher in einem spezialisierten Strom nicht unter Ausnutzung der neuen Möglichkeiten optimiert werden und ein Nutzer, der einen spezialisierten Strom vorliegen hat, verliert durch Anwendung einer solchen Methode die Spezialisierungseigenschaften. Vorteil der subtyperhaltenden Spezialisierung ist jedoch die Möglichkeit zur Definition einer für alle Behälter einheitlichen Methode zur Erzeugung von Elementströmen.

Der gepufferte Strom BufferedReadStream bietet mit der Methode peek die Möglichkeit das aktuelle Element (mehrfach) abzurufen, ohne den Schritt zum nächsten Element zu vollziehen.

Die Klasse InternalReadStream schließlich stellt die schon angekündigten Funktionsvariablen für das Basisprotokoll eines ReadStreams zur Verfügung. Sie werden bei Kombination mit anderen Eigenschaften durch weitere Variablen ergänzt. So müssen zur Realisierung der Positionierbarkeit die Methoden zum Auslesen und Setzen der Position vom Stromerzeuger (oder von einer Subklasse) auf diese Weise implementiert werden.

7.6.2 Abgrenzung

Die Ströme der TooL Klassenbibliothek sind durch Anpassung und Erweiterung aus den in der Tycoon Modulbibliothek angebotenen Iteratoren entstanden. Sie sind, im Gegensatz zu der dort erfolgten rein funktionalen Implementierung, prinzipiell eigenständige Objekte, mit internem Zustand und definiertem Protokoll. Offensichtlich ist dies bei Nachfolgern wie File und PrimesStream, welche die Basismethoden direkt implementieren.

Die Realisierung des Basisprotokolls mittels Funktionsvariablen, denen die Implementierung erst zugewiesen werden muß, verlagert einen Teil des internen Zustandes in den Funktionsabschluß (die 'Closure') der zugewiesenen Funktionen und damit auf die Seite des Nutzers.

Aus Sicht eines Behälters bedeutet dies, daß er die Anpassung des Stream-Protokolls individuell selbst durchführen kann. Der Bedarf für eine parallele, den verschiedenen Behältertypen gerecht werdende Stream-Hierarchie entsteht dadurch nicht. Für die Trennung von Behälterzustand und Zustand des Streams ist der Nutzer (die Behälterklasse) allerdings selbst verantwortlich, was bei der Verwendung von auf den Behältertypus spezialisierten Streams vermieden

werden könnte. Andererseits läßt sich in der vorliegenden Variante die Koppelung zwischen Behälter und angebotenem Elementstrom flexibel gestalten, ohne das die Definition eines veränderten Basisprotokolls notwendig wäre. Die Implementierung der elements-Methode in einem Behälter sollte aber auf jeden Fall ermöglichen, daß mehrere Iterationen gleichzeitig, bzw. geschachtelt auf dem selben Behälter durchfühbar sind ohne daß diese sich gegenseitig stören. Dieser Grundsatz wurde bei den vorhandenen Implementierungen eingehalten.

Bei der Nutzung der Streams als Iterationsabstraktion für die Behälter stellt sich die Frage nach der Stabilität des Iterierungsprozesses, d.h. seiner Beeinflußbarkeit oder Störungsanfälligkeit gegenüber Einfüge- Lösch- und Reorganisationsaktionen auf dem in Bearbeitung begriffenen Behälter [Traub 95, S.86].

Für Iteratoren kann in diesem Zusammenhang eine Unterteilung in interne und externe Iteratoren getroffen werden [Kofler 93]. Als extern wird das schrittweise Durchlaufen mittels weniger Grundoperationen, als intern die Bearbeitung der Gesamtmenge durch *einen* Methodenaufruf bezeichnet.

Bei einem rein funktionalen Ansatz ist das interne Iterieren gegenüber Störungen unanfällig. Die Methode ist auf eine gewünschte Aussage spezialisiert und läßt keine Beeinflussung zu:

```
without(e :E) :ReadStream(E) reject(fun(e1 :E)e1 == e)
```

Eine andere Form des internen Iterierens wird durch die Übergabe einer Funktion gesteuert, wie die zur Implementierung von without verwendete Methode reject. Sind Funktionen seiteneffektfrei, so ist auch in diesem Fall der Iterationsprozess stabil.

Beim schrittweisen Vorgehen sind prinzipiell jederzeit Operationen möglich, die den zugrundeliegenden Behälter beeinflussen und dadurch die Iteration destabilisieren können.

Alle drei Varianten kommen nebeneinander in der Schnittstelle eines TooL-Streams vor. Eine Aufteilung in interne und externe Iteratoren wird nicht vorgenommen. Funktionen in TooL können im allgemeinen seiteneffektbehaftet sein. Die Funktionsrümpfe können jede Art von Behältermanipulationen enthalten, so daß die Unterscheidung der internen und externen Iteration hier nicht greift.

Der Grad der Stabilität des Iterierungsprozesses über Behälterelemente kann durch eine dreistufige Klassifizierung festgelegt werden. Dabei wird auf drei Methodengruppen Bezug genommen, welche die Inkonsistenz verursachen können. Die drei Gruppen sind Einfüge-, Lösch- und reorganisierende Methoden. Die Klassifizierung lautet dann nach [Traub 95, S.86]:

- Als **stabil** gilt eine Iterierung, wenn sie weder durch Einfüge- und Löschoperationen in den Behälter noch durch die Reorganisation desselben in einen inkonsistenten Zustand gerät.
- **Robust** heißt eine lediglich gegenüber Behälterreorganisationen konsistenzgefährdete Iterierung.
- Mit **instabil** bezeichnet man schließlich eine, durch alle drei Modifikationsereignisse gefährdete Iterierung.

In dem hier betrachteten Fall ist der Grad der Stabilität von der Implementierung der Stream-Basisoperationen durch die einzelnen Behältertypen abhängig und wird daher von jedem Behälter selbst bestimmt. In der derzeitigen Version der Behälterbibliothek liefern die meisten Behälter nach dieser Definition instabile Streams.

Die Nutzung eines Streams zur direkten Manipulation der Behälterstruktur ist allerdings meistens nicht notwendig. Dies kann ebenso direkt über die Schnittstelle des Behälters geschehen, ohne daß die Gefahr einer Inkonsistenz besteht. Die Streams können dagegen zur kompakten Bearbeitung, Filterung, Konvertierung und Transportierung der Elementmenge eingesetzt werden, wie dies die meisten Methoden auch nahelegen. Die Einhaltung dieser Trennung erspart den Aufwand für eine robuste oder gar stabile Implementierung. Die Umsetzung dieser Trennung von Anfrage und Modifikation führt zu algorithmischen Problemlösung auf einem hohen Abstraktionsniveau.

Die Verwendung eines Konzeptes, welches explizite frei bewegliche Zeiger in den Behälter anbietet, verleitet dagegen eher zu semantisch komplizierteren Lösungen [Murer et al. 94, S.8]. So stellt ein solcher Zeiger eben keine Iterationsabstraktion zur Verfügung sondern verlangt die Positionierung. Er lädt von vornherein eher zur parallelen Modifikation des Behälters ein und verlangt somit auch mehr nach einer Lösung der Stabilitätsproblematik.

Die Frage nach der Konsistenz der Iterierung kann im Prinzip beliebig ausgeweitet werden. Konsistenz sollte in jedem Fall bedeuten, daß der Prozeß nicht 'gewaltsam unterbrochen' wird oder einen anderen fehlerhaften Verlauf nimmt, zum Beispiel die Fortsetzung der Iterierung in einem anderen Behälter. Sie kann sich aber auch auf den Zustand der Elementmenge des Behälters beziehen. Auf jeden Fall sollten alle Elemente des Behälters erfaßt werden. Sind dies aber die Elemente zu Beginn des Prozesses oder werden auch Änderungen während des Durchlaufens sichtbar? Werden dann alle Änderungen oder nur solche in dem noch nicht erfaßten Bereich des Behälters berücksichtigt? Die Trennung von Anfrage und Modifikation mittels der Stream-Funktionalität einerseits und Veränderung der Behälterstruktur durch Einfüge- Lösch- und Reorganisationsaktionen über die Behälterschnittstelle andererseits läßt diese Probleme jedoch in den Hintergrund treten, sofern man diese Trennung einhalten will.

7.7 Zwischenergebnis

Der folgende Abschnitt soll ein kurzes Zwischenergebnis formulieren im Hinblick auf die am Anfang von 7 beschriebenen Zielsetzungen.

Zunächst wurde am Beginn des Kapitels die Frage aufgeworfen, ob sich eine Behälterklassenbibliothek unter den besonderen Randbedingungen der Sprache TooL entwerfen läßt. Besonders hervorgehoben wurde in diesem Zusammenhang die strikte Typisierung, der Zwang zur Schnittstellenvererbung, der keine Umbenennung und kein Verstecken von Methoden erlaubt, sowie die Einführung der Ähnlichkeitsrelation und des diesbezüglichen parametrischen Polymorphismus. Weiterhin sollte das Ziel hoher Wiederverwendbarkeit und Kombinierbarkeit von Funktionalität nicht aus den Augen verloren werden.

Der Ansatz, die Methoden der Behälterklassen so aufzuteilen, daß eine Umbenennung, Ausklammerung oder 'Entfunktionalisierung'[10], wie es in vielen Bibliotheken praktiziert wird[11], vermieden werden kann, führt zwangsläufig zu einer feineren Aufteilung. Es muß darauf geachtet werden, daß die Methoden mit den Eigenschaften eines konkreten Behälters harmonieren, damit Veränderung der Schnittstellen gar nicht erst nötig werden.

Das in Abbildung 7.1 skizzierte Entwurfsschema führt eine eigenschaftsbezogene Anordnung der Methoden durch. Dabei werden zunächst allgemeine Behältereigenschaften formu-

[10] Damit soll die Überschreibung einer Methode durch einen Implementierung gemeint sein, die nur noch eine Fehlermeldung ausgibt

[11] Siehe Abschnitte 4.1, 4.1.2, 4.3 und 4.4

liert, deren korrespondierende Klassen entsprechend allgemeines Protokoll aufnehmen. Diese Eigenschaften werden zur besseren Übersichtlichkeit in Eigenschaftsgruppen zusammengefaßt.

Durch Kombination solcher Eigenschaften werden Behälterkategorien gebildet. Die Zusammenfassung von allgemeinen Eigenschaften in einer Wurzelklasse einer Behälterkategorie unterstützt die einfache Sicht auf kompliziertere Behältervarianten einer solchen Sparte.

Diese Behältervarianten ergeben sich aus der erneuten Unterscheidung von speziellen Eigenschaften innerhalb einer Behälterkategorie, welche auch nur eine Konkretisierung von allgemeinen, in der Wurzelklasse der Kategorie noch nicht integrierten Eigenschaften darstellen können. Diesen speziellen Eigenschaften kann nun weiteres behälterspezifisches Protokoll zugeordnet werden, welches dann zur Kombination bereit steht.

Durch das beschriebene Verfahren kann die gewünschte Aufteilung und Kombinierbarkeit von Teilfunktionalität erreicht werden. Es entsteht allerdings eine komplexere Klassifizierungsstruktur. Die Komplexität wird durch die Gewährleistung der Schnittstellenvererbung für den Benutzer gemildert, da er sich auch mit dem einfacheren Protokoll einer höheren Abstraktionsstufe zufrieden geben kann. Bei der Entwicklung neuer Subklassen kann das Klassifizierungsschema als Orientierungshilfe dienen. Es sind aber weitere Hilfsmittel zur Darstellung des Vererbungsgraphen, Anzeige von Objektschnittstellen und zur Entwurfsunterstützung notwendig. Es wurden im Rahmen der Arbeit auch bereits erste Möglichkeiten zur Visualisierung der Klassendefinition und der Vererbungsbeziehungen geschaffen, die im Abschnitt 7.9 vorgestellt werden.

Die letzte der zu Anfang des Kapitels aufgeworfenen Fragen betraf die Auswirkungen des speziellen Typsystems von TooL auf die Behälterklassenbibliothek. Die Ähnlichkeitsrelation ermöglicht die Integration der Vererbung von Methoden mit Parametern vom Objekttyp der Klasse selbst, der dann in der Subklasse in typsicherer Art und Weise spezialisiert wird. Dies erleichtert einen intuitiven Umgang mit diesen Methoden. Andere Sprachen verlangen hier generell Subtypbeziehung[12] und erlauben trotzdem die Spezialisierung, wie dies z.B. in Eiffel der Fall ist. Dadurch geht die Substituierbarkeit der Implementierung in den Kontext der Oberklassen prinzipiell verloren. Es wird eine Analyse dieses Kontexts benötigt, wodurch die Modularität der Typsicherheitsüberprüfung verloren geht [Gawecki, Matthes 96, S.4]. Dies kann in TooL vermieden werden.

Die durch das Design der Behälterklassenbibliothek sichergestellte einfache Sicht auf kompliziertere Objekte kann trotz Verlustes der Subtypbeziehung zwischen den Behälterklassen durch Anwendungsprogramme genutzt werden, indem sie den Ähnlichkeitspolymorphismus zur Formulierung von generischen Methoden einsetzen (Siehe Abbildung 7.10). Dieser kann auch bereits innerhalb der Behälterklassenbibliothek nutzbar gemacht werden. So zum Beispiel durch den Einsatz von Typparametern zur Spezifikation einer generischen Klasse für die Implementierung einer Liste durch verkettete Zellen. Die einfache rekursive Definition der Zellen unter Ausnutzung der Vererbung, die zum Verlust der Subtypbeziehung führt, erlaubt dennoch die polymorphe Formulierung von Algorithmen (Siehe auch Abschnitt 7.4.1 und Abbildung 7.16).

Die Verwendung der Ähnlichkeitsrelation zur Beschreibung des Elementtyps eines Behälters als mit einer Gleichheitsrelation ausgestattet, ist naheliegend. Sie gerät jedoch in Konflikt mit der Forderung nach Subtypheterogenität[13] der Behälter. Diese spezifischen, mit der Implementierung der Ähnlichkeitsrelation in TooL zusammenhängenden Probleme lassen

[12] In [Meyer 89, S.6] wird die kontravariante Subtypregel für Methodenparameter als wenig intuitiv kritisiert

[13] Dies soll die Fähigkeit eines Behälters bezeichnen, auch Subtypen seines Elementtyps aufzunehmen.

sich aber auf einfache Weise umgehen (Abschnitt 7.5), sodaß die Funktionalität der Behälter trotzdem genutzt werden kann.

7.8 Verwendung von Entwurfsmustern

In verschiedenen Veröffentlichungen wurde in letzter Zeit der Einsatz von Entwurfsmustern (*design patterns*) in der objektorientierten Softwareentwicklung diskutiert und entsprechende Mustersammlungen vorgestellt [Gamma et al. 95; Pree 95].

Die Entwurfsmuster dieser Sammlungen beruhen auf den bei der Entwicklung von großen objektorientierten Systemen gemachten Erfahrungen. Mit der Beschreibung von Entwurfsmustern wird versucht, Lösungen für bestimmte immer wieder auftretende Problemstellungen als Schema zu erfassen, daß heißt den Kern des Lösungsansatzes darzustellen und zu benennen. Gelingt dies, so können sie von anderen Entwicklern genutzt werden, um in ähnlichen Situationen die gleiche Technik einzusetzen. Der Entwickler kann das Muster in seinen Kontext übernehmen und seine Entwurfsentscheidungen auf einem abstrakteren Niveau überdenken. Die allgemeine Formulierung solcher Lösungsansätze macht sie austauschbar.

Die Beschreibungen von Entwurfsmustern beruhen meist auf der Darstellung von Beziehungen zwischen Objekten und Klassen. Die Problemsituation geschildert, die Lösung skizziert und mit Beispielen verdeutlicht. Entwurfsmuster werden sowohl zur Verdeutlichung von systemspezifischen Entwurfsentscheidungen, um dem Benutzer den Einstieg zu erleichtern, als auch zur allgemeinen Darstellung von Konstruktionstechniken eingesetzt.

Im folgenden werden einige Entwurfsmuster vorgestellt und anschließend ihre Anwendung in der Behälterklassenbibliothek von TooL diskutiert.

7.8.1 Brückenklassen

Das Entwurfsmuster *Bridge* oder Brückenklasse [Gamma et al. 95, S.151ff] beschreibt eine Technik zur Entflechtung von Implementierung und Abstraktion. Werden verschiedene Implementierungen für eine Abstraktion als Subklassen realisiert, so führt dies bei einer Verfeinerung der Abstraktion zu entsprechend vielen Nachkommen der Verfeinerung, welche die Implementierungsvarianten umsetzen. Umgekehrt bewirkte die Einführung einer neuen Implementierungsform, daß diese für die Abstraktion selbst und für jede Verfeinerung der Abstraktion umgesetzt werden muß. Die Brückentechnik kann diese Situation vereinfachen und eine separate Verfeinerung für beide Seiten ermöglichen. Weiterhin kann unter anderem der Implementierungswechsel zur Laufzeit bzw. eine automatische Implementierungswahl erreicht werden.

Grundidee für die Brückenbildung ist die Definition eines abstrakten Implementierungsprotokolls durch eine separate Implementierungsklasse. Die Nachfolger dieser Klasse realisieren dann dieses Protokoll für unterschiedliche Varianten der Repräsentation. Mit Hilfe dieses Protokolls kann nun eine Brücke zu der eigentlichen Abstraktion geschlagen werden. Diese nutzt es, um ihre Methoden (teilweise) zu implementieren. Um eine konkrete Implementierung zuordnen zu können, wird eine Variable für ein solches Implementierungsobjekt eingeführt. Über diese wird das abstrakte Implementierungsprotokoll angesprochen, sodaß bei einer anderen Anfangsbelegung oder einem Austausch des Objekts die jeweilige konkrete Implementierung verwendet wird. In Abbildung 7.20 wird das Verfahren skizziert.

Die Nutzung des abstrakten Implementierungsprotokolls ist entscheidend für die einfachere Ergänzung weiterer Verfeinerungen oder Implementierungsvarianten. Werden statt dessen direkte Brückenklassen[14] zur Verbindung von Abstraktion und Implementierungsvariante ge-

[14] Eine spezielle Klasse, welche die durch die Abstraktion spezifizierten Methoden mit Bezug auf eine konkrete Implementierungsklasse realisiert.

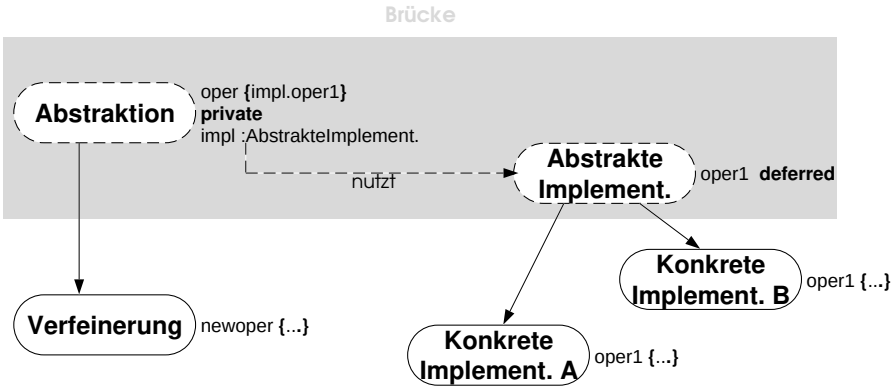

Abbildung 7.20: Entflechtung von Implementierung und Abstraktion: Brückenbildung (aus [Gamma et al.,S.153])

nutzt [Traub 95, S.64], so muß bei einer Verfeinerung erneut für jede Variante eine Brücke hergestellt werden. Diese degenerierte Brückentechnik [Gamma et al. 95, S.160] unterstützt also gerade nicht die Verfeinerung auf beiden Seiten der Brücke.

7.8.2 Fabrikklassen

Das Entwurfsmuster *abstract factory* [Gamma et al. 95, S.87ff] generalisiert die Erzeugung von Objekten.

Eine abstrakte Fabrikklasse definiert eine Schnittstelle mit Erzeugungsmethoden für zusammengehörige oder abhängige Familien von Objekten. Konkrete Nachfolger dieser Klasse repräsentieren familienübergreifende gleichartige Erzeugungsverfahren und implementieren die Methoden in diesem Kontext. Diese Nachfolger können dann genutzt werden, um die Objekte der einzelnen Familien kontextabhängig zu erzeugen.

Für einen Klienten bedeutet dies, daß er sich durch einmalige Wahl der 'richtigen' Fabrik versichern kann, immer die korrespondierende Objektvariante aus den verschiedenen Familien zu erhalten. Er kann von der kontextspezifischen Erzeugung abstrahieren, da diese von der Fabrik geleistet wird. Ein Kontextwechsel kann außerdem einfach durch Austausch der Fabrik erfolgen.

In Abbildung 7.21 ist die Technik skizziert. Die beiden abstrakten Klassen ProduktA und ProduktB beschreiben zwei Objektfamilien, für deren Nachfahren VarianteA1 und VarianteB1 sowie VarianteA2 und VarianteB2 eine Verbindung durch ihr Auftreten in einem gemeinsamen Kontext besteht. Ein solcher Kontext könnte zum Beispiel die Plattformabhängigkeit von bestimmten Objekten sein. In der Klasse AbstrakteFabrik werden Methoden zur Erzeugung von Objekten beider Familien eingeführt. Die Klasse KonkreteFabrik1 beispielsweise definiert diese Methoden dann so, daß jeweils die Variante eins der Objektfamilien A und B generiert wird.

Nachteilig für die Erweiterbarkeit ist die Spezifizierung einzelner Erzeugungsmethoden für jede Objektfamilie. Bei Einführung weiterer Objektfamilien müssen entsprechende Erzeugungsmethoden ergänzt werden, was dann zur Änderung aller kontextbezogenen Nachfolger der abstrakten Fabrik veranlaßt. Vermeidbar ist dieses Problem durch Verwendung eines Pa-

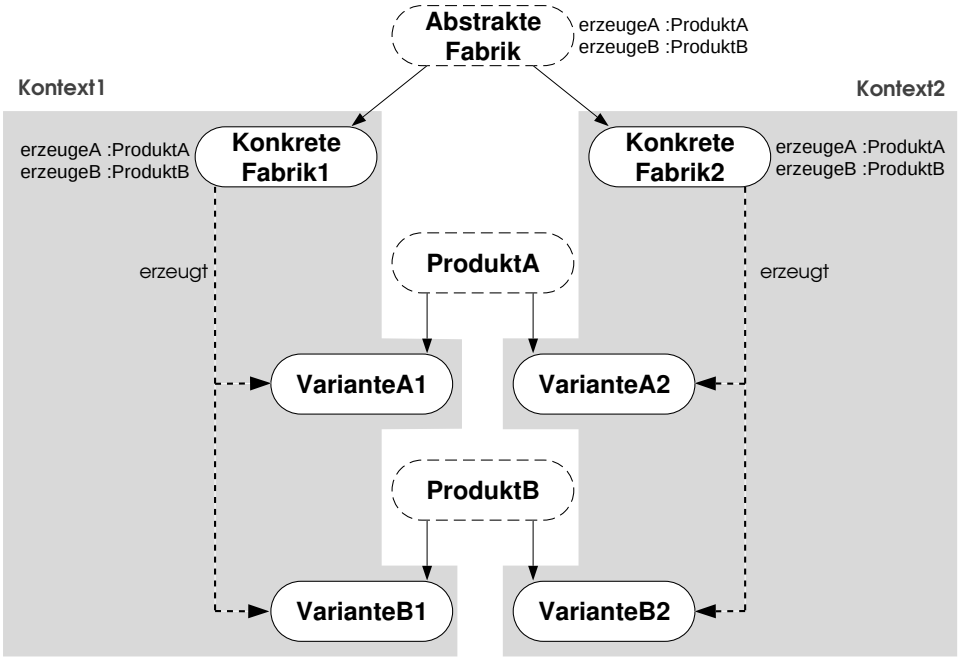

Abbildung 7.21: Kontextbezogene Objekterzeugung: Abstrakte Fabrik (nach [Gamma et al. 95, S.88])

rameters, mit dem die zu erzeugende Objektart festgelegt wird. Es wird dann nur noch eine Erzeugungsmethode benötigt. Für statisch typisierte Sprachen ist diese Variante generell nur einsetzbar, wenn die beteiligten Objektfamilien eine gemeinsame Oberklasse haben. Dann besteht die Gefahr, daß nur Objekte mit dem gemeinsamen abstrakten Protokoll dieser Oberklasse erzeugt werden und Spezialisierungen für Klienten nicht mehr sichtbar sind [Gamma et al. 95, S.91]. Durch den Einsatz von Typparametern, wie dies in TooL möglich ist, bleibt die Fabriktechnik aber auch dort einsetzbar. Die Fabriken können bei ihrer Erzeugung so instantiiert werden, daß die gewünschte Spezialisierung mit vollzogen wird. Dies wird auch in den Beispielen des nächsten Abschnittes deutlich werden.

7.8.3 Anwendung

Im folgenden soll die Umsetzung der beiden vorgestellten Entwurfstechniken *Brücke* und *AbstrakteFabrik* für die Behälterklassenbibliothek von TooL anhand eines Beispiels demonstriert und diskutiert werden.

Der Zielsetzung der Brückentechnik entsprechend liegt es für die Behälterklassen nahe, unterschiedliche Implementierungsvarianten einer Behälterkategorie auf diese Weise zu realisieren. Solche Varianten wurden bisher in der Behälterbibliothek von TooL als spezielle Eigenschaften einer solchen Kategorie interpretiert, wie dies am Beispiel der Listen in Abbildung 7.11 deutlich wird. Die Definition einer weiteren Implementierungseigenschaft ArrayedList, die für alle Varianten realisiert werden soll, würde zur Definition eines Teilgraphen von Klassen für die Kombination mit den anderen speziellen Eigenschaften führen. Durch die Anwendung der Brückentechnik kann der Aufbau dieses Teilgraphen eingespart werden, wenn man für

das abstrakte Implementierungsprotokoll eine weitere Subklasse definiert, die dieses Protokoll über ein Array realisiert.

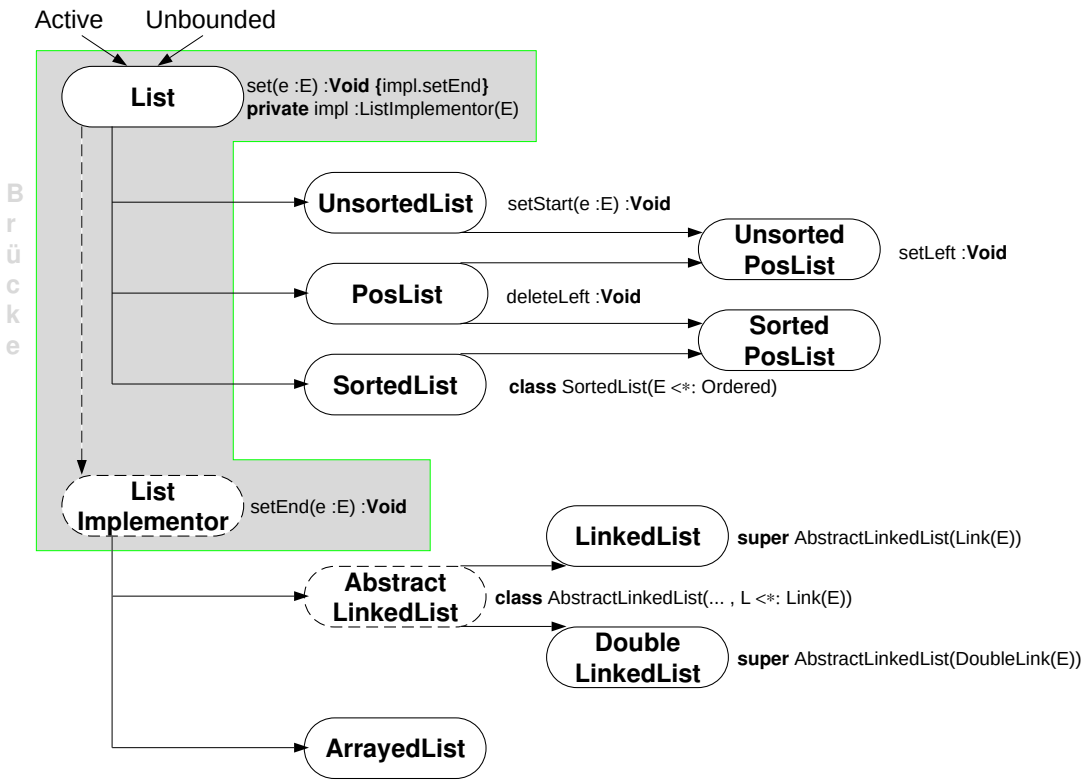

Abbildung 7.22: Listen mit separater Repräsentationsimplementation

Die Behälterkategorie der Listen kann dann in der in Abbildung 7.22 gezeigten Form definiert werden. Waren in der alten Version noch die Kombinationen mit den Implementierungseigenschaften zur besseren Übersichtlichkeit nicht abgebildet, so ist hier bereits die gesamte Listenkategorie zu sehen. Der Graph wird flacher. In den Listenklassen sind nun alle Methoden mit Hilfe des abstrakten Protokolls aus ListImplementor realisiert. Auch die Wurzelklasse ist bereits instantiierbar. Sie definiert jetzt eine Variable impl zur Verwaltung der aktuellen Implementierungsform. Der Typ dieser Variablen wird durch die Wurzelklasse der Listenimplementierungen spezifiziert. Dies verlangt die Einhaltung der Subtypbeziehung innerhalb der Hierarchie von Implementierungsvarianten. Soll auch hier die spezialisierende Vererbung von Methoden mit Objekttypparametern eingesetzt werden, so können solche Imlementierungen durch die Einführung eines weiteren Typparameters für die Klasse List, der den Implementierungstyp beschreibt, genutzt werden.

Dadurch, daß in der Listenhierarchie von der Art der Repräsentation abstrahiert wird, tritt eine diesbezügliche Implementierungsvererbung dort nicht mehr auf. Die Implementierungshierarchie ist prinzipiell flach angelegt, da die einzelnen Varianten nebeneinander stehen. Generell ist aber hier auch eine Verfeinerung möglich, sodaß diese Art der Vererbung dort wieder auftreten kann. Auf der Listenseite bleibt weiterhin auch die Vererbung von nicht repräsentationsbezogenen Implementierungen bestehen.

Das von der Klasse ListImplementor angebotene Protokoll muß so mächtig sein, daß alle Listenmethoden der verschiedenen Listenvarianten damit implementierbar sind und die jeweiligen Vorteile der Repräsentationsstrukturen auch zum Tragen kommen. Im Prinzip wird also durch diese Klasse bereits eine alle Eigenschaften vereinigende Liste spezifiziert, deren Mächtigkeit dann durch die eigenschaftsbezogenen Varianten der Listenhierarchie wieder gekapselt wird. Diese Beobachtung könnte dazu verleiten, bereits die Wurzelklasse mit der gesamten Funktionalität auszustatten bzw. die Aufteilung des Listenprotokolls erst gar nicht vorzunehmen. Damit enfallen aber die Möglichkeiten, die eine solche Aufteilung mit sich bringt: das eigenschaftsbezogene Auswählen, die vereinfachte Sicht auf komplizierte Objekte und die Kombination mit neuen Eigenschaften unter Vermeidung von inkompatiblen Protokollgruppen. Die skizzierte Lösung verknüpft diese Vorteile mit den oben dargelegten, der Brückentechnik zuzuordnenden Verbesserungen.

Etwas problematisiert wird die Erzeugung von Listenobjekten. Die auf die Implementierung verweisende private Variable muß belegt und damit die Entscheidung für eine Implementierung gefällt werden. Verschiedene Lösungsansätze sind denkbar. Die Entscheidung kann intern getroffen werden, sollte aber für die Listenbehälter vom Benutzter beeinflußbar sein, um die Wahl der effizientesten Elementstruktur dem der Liste zugedachten Einsatzbereich anzupassen. Die Belegung der privaten Variable kann in TooL nur durch Subklassen oder mit Hilfe von Methodenaufrufen erfolgen. Subklassenbildung entspricht der Bildung von direkten Brückenklassen zu den Implementierungsvarianten und ist daher aus den bereits unter 7.8.1 aufgeführten Gründen abzulehnen.

Günstig wäre es, die Entscheidung für eine Implementierungsvariante gleich bei der Erzeugung treffen zu können. Methoden zur Erzeugung von Objekten einer Klasse werden in TooL in der Metaklasse dieser Klasse definiert. Diese braucht dann aber den Zugriff auf die Implementierungsvariable. Es wird deshalb eine Methode zur Zuweisung der Implementierung der durch List spezifizierten Objektschnittstelle hinzugefügt. In der Metaklasse erhält die Methode new nun einen Parameter, auf dem die gewünschte Implementierung übergeben werden kann. Auch alle anderen Erzeugungsmethoden benötigen diesen Parameter.

Die Erzeugung der verschiedenen Implementierungsvarianten der Listen kann allerdings auch mit Hilfe der unter 7.8.2 geschilderten Fabriktechnik durchgeführt werden. Die Adaption des Schemas mit Objektfamilien und kontextabhängiger Erzeugung kann folgendermaßen geschehen. Als Objektfamilie können die mit verschiedenen konkreten Implementierungen ausgestatteten Listenobjekte einer bestimmten Listenvariante betrachtet werden. Also z.B. die Array- oder Kettungsimplementierungen einer positionierbaren Liste. Jeder Variante sind entsprechende Objekte zuzuordnen[15] und jede Variante kann daher als eigenständige Objektfamilie betrachtet werden. Als gemeinsamer Kontext wirken die verschiedenen Implementierungen als Array, oder unter Verwendung einfach bzw. doppelt verketteter Zellen. Es ist also die Erzeugung verschiedener Listenvarianten in dem gemeinsamen Kontext, daß zum Beispiel eine Arrayimplementierung benötigt wird, zu modellieren.

Abbildung 7.23 zeigt die entsprechende Listenfabrik. Sie führt Typparameter zur Anpassung des Rückgabetyps der Erzeugungsmethoden ein, wie schon unter 7.8.2 angedeutet.

Bei der Erzeugung einer Listenfabrik muß der Typ der zu erzeugenden Listenvariante übergeben werden. Die Fabrik ist dann auf diese Variante zugeschnitten. Die Implementierungsart kann nun explizit durch die Wahl der entsprechenden Fabrikklasse für die gewünschte Form

[15] Diese Objekte können auch als Objekte der nicht definierten direkten Brückenklassen jeder Listenvariante interpretiert werden.

Abbildung 7.23: Erzeugung von Listenvarianten: Listenfabriken

festgelegt werden.

Jeder Erzeugungsmethode müßte eigentlich eine konkrete Klasse des richtigen Typs übergeben werden, damit diese zur Erzeugung eines Listenobjekts benutzt werden kann. Um diesen Parameter zu vermeiden, wurde eine weitere Methode in AbstractListFactory eingeführt welche den Produktionsvertrag der Fabrik mit einer konkreten Listenvariante schließt. Diese Klasse wird auf einer privaten Variable gespeichert und dann in den Subklassen von der Methode new zur Erzeugung eines Objekts dieser Klasse genutzt. Die Initialisierung der Implementierungsvariablen des Listenobjekts erfolgt entsprechend dem gewünschten Implementierungskontext.

Erzeugung und Einsatz einer Fabrik können dann, wie in Abbildung 7.24 zu sehen, durchgeführt werden.

```
| > define listFactory :AbstractListFactory(Int, UnsortedList(Int));
| > listFactory := ArrayedListFactory.new;

a ArrayedListFactory
| > listFactory.contractWith(UnsortedList);

nil
| > define l :UnsortedList(Int);
| > l := listFactory.list4(1,2,3,4)

a UnsortedList{1 2 3 4}
| >
```

Abbildung 7.24: Fabrikation

Die hier realisierte Variante des allgemeinen Entwurfsmusters mit Typparametern erlaubt es auch weiterhin, innerhalb des Anwendungsprogramms vom Erzeugungsvorgang zu abstra-

hieren. Die Initialisierung mit der Implementierungsvariante braucht nicht mehr explizit zu erfolgen. Außerdem werden Informationen darüber, welche Klasse eine solche Implementierungsvariante spezifiziert, für die Formulierung der Anwendung nicht mehr benötigt. Der Name der benutzten Listenvariante taucht hier allerdings im Anwendungsprogramm auf. Allerdings nur im Zusammenhang mit der Fabrikerzeugung, da er dort für die Instantiierung der Typparameter benötigt wird. Die Erzeugung konkreter Listen kann dann immer über die Listenfabrik erfolgen.

Durch die Verwendung der vorgestellten Techniken kann also eine flachere Struktur und damit eine Vereinfachung des Vererbungsgraphen erreicht werden. Das Hinzufügen von weiteren Klassen wird durch die Trennung von Implementierung und Abstraktion erleichtert. Der Wechsel der Implementierungsform zur Laufzeit ist auch in der Biliotheksversion mit gleichem Aufwand möglich, allerdings nicht direkt durch den Aufruf einer Listenmethode. Es bedarf der Erzeugung einer Liste mit der gewünschten Implementierung, in welche dann die Elemente mit Hilfe der Streams eingefügt werden.

Ein geeignetes abstraktes Implementierungsprotokoll zu finden, kann problematisch sein. Fällt es zu umfangreich aus, so leidet die Übersichtlichkeit. Bei der Definition einer einfachen neuen Listeneigenschaft bedeutet die Beschäftigung mit diesem Protokoll eine unnötige Komplizierung für den Entwickler. Bietet das Protokoll dagegen nur wenige Methoden an, dann besteht die Gefahr, daß die Vorteile konkreter Implementierungen nicht genutzt werden können. Die Bindung aller Listenklassen an das abstrakte Implementierungsprotokoll verlangt darüber hinaus eine frühe Stabilisierung dieser Schnittstelle. Änderungen wirken sich auf alle Listenvarianten aus.

Die Anwendung der Fabriktechnik scheint dagegen für die Behälterklassen keine so großen Vorteile zu bringen. Sie erspart zwar den sonst für jede Erzeugungsmethode der Metaklasse notwendigen Parameter für die Implementierungsform, dies ist aber nur für Anwendungen von Belang, die häufig gleichartige Objekte diesen Typs erzeugen wollen wie z.B. Fenster einer Benutzeroberfläche. Bei Behältern ist dies nicht unbedingt zu erwarten. Es steht eher die Arbeit mit dem Behälter im Fordergrund nicht so sehr seine Erzeugung. Unter diesem Aspekt ist die zusätzliche Erzeugung einer entsprechenden Fabrikklasse eine Komplizierung des Verfahrens. Weiterhin positiv zu beurteilen ist, daß die Implementierungsklassen im Verborgenen bleiben, was aber auch durch entsprechende Methodenvielfalt in der Metaklasse erreicht werden kann.

7.9 Entwurfshilfsmittel

Die strenge Typisierung der Sprache TooL, der Zwang zur Schnittstellenvererbung in Verbindung mit dem Wunsch nach Kombinierbarkeit von Funktionalität und Vereinheitlichung haben einen tiefen Vererbungsgraphen entstehen lassen. Die Arbeit mit einer solchen Struktur bedarf der Unterstützung durch Werkzeuge. Diese sollten mindestens einen Überblick über die Vererbungsbeziehungen und eine Darstellung der durch eine Klasse spezifizierten Objektschnittstelle bieten. Denkbar wären aber auch Werkzeuge, die den Entwurfsprozeß direkt unterstützen, in dem sie beispielsweise nur Teilbereiche des Vererbungsgraphen anzeigen. Spezielle Werkzeugfunktionen dieser Art könnten die Darstellung aller in einer Behälterkategorie unterschiedenen Eigenschaften, die Anzeige einzelner Eigenschaftsgruppen mit ihren Nachfahren oder die Auswahl einer Klasse mit spezifischer Eigenschaftskombination unterstützen. Dieser Abschnitt soll die im Rahmen dieser Arbeit entstandenen ersten Ansätze zur Visualisierung der Klassendefinitionen und der Vererbungsbeziehungen vorstellen.

7.9.1 Methodenakkumulation

Durch die Tiefe des Vererbungsgraphen verteilen sich die Methodenspezifikationen und Implementierungen von Protokollgruppen auf die Oberklassen einer konkreten Klasse.

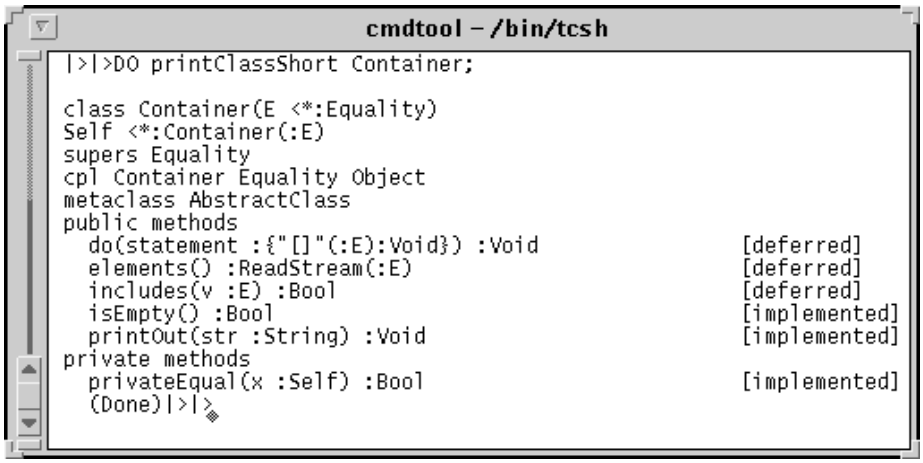

Abbildung 7.25: Anzeige der Klassendefinition : Kurzform

Für die Nutzung, Implementierung und Spezialisierung einer Klasse ist es aber notwendig, die komplette (bisher) vorhandene Objektschnittstelle aber auch die innerhalb der Klasse verfügbaren privaten Methoden zu kennen. Um diesen Überblick zu gewinnen, müßten alle Oberklassen betrachtet werden. Es ist also eine Möglichkeit benötigt, sich die akkumulierte Methodenmenge anzeigen zu lassen. Derartige Funktionen wurden in den Kommandointerpreter von TooL integriert. Dieser Interpreter dient unter anderem dem Anstoßen von Lade-, Übersetzungs- und Typüberprüfungsvorgängen. Es können aber auch direkt TooL Ausdrücke eingegeben werden.

Nach dem Laden einer Klasse sind die Methodensignaturen und die Klassenpräzedenzliste im Objektspeicher des Tycoon-Systems verfügbar und persistent. Ihre Speicherung ist für die

modulare Typüberprüfung notwendig. Dadurch ist es möglich diese Informationen jederzeit sichtbar zu machen und über die Präzedenzliste auch eine Akkumulierung der Methodensignaturen durchzuführen.

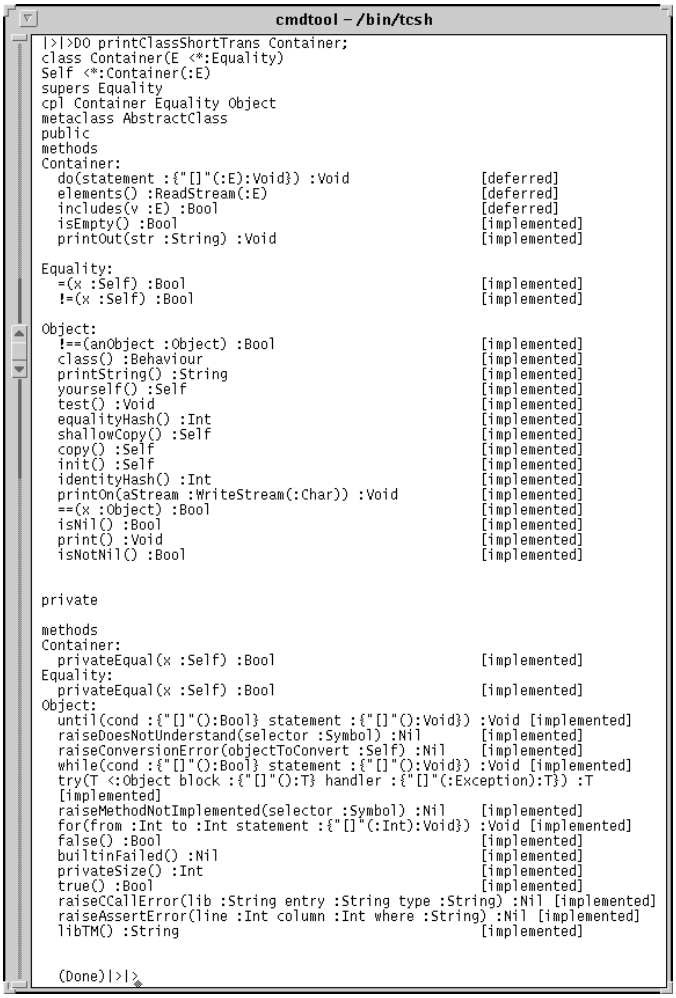

Abbildung 7.26: Anzeige der Klassendefinition: akkumulierende Form

Es werden unterschiedliche Varianten von Interpreterbefehlen angeboten. Die einfachste Form zeigt, wie in Abbildung 7.25 zu sehen, nur die in einer Klasse spezifizierten Methodensignaturen an. Die über die textuelle Klassendefinition hinausgehende Information besteht in der Anzeige der Klassenpräzedenzliste, welche hinter der Zeile zur Angabe der Oberklassen eingefügt wird. Anschließend werden die Signaturen in der gewohnten Weise, aufgeteilt in einen öffentlichen und privaten Bereich, aufgelistet. Für jede Methode wird bekundet, ob es sich um eine implementierte oder nicht implementierte (mit dem Schlüsselwort **deferred** gekennzeichnete) Methode handelt.

Weitere Befehle stellen auch die aus den Oberklassen geerbten Methoden dar. Das Kommando '**DO** printClassShortTrans' listet die Vorfahren der aktuellen Klasse in der Reihenfolge der Präzedenzliste mit den ihnen zugeordneten Methoden und Variablen auf (Abbildung 7.26).

Es werden auch die Überschriebenen und die nicht implementierten Methoden angezeigt, um gegebenenfalls eine zur Beeinflussung der Präzedenzliste nötige Umstellung in der Liste der Oberklassen erkennen zu können. Ergänzende Kommandos schränken diese allgemeinste Anzeige wieder ein, etwa indem nur die Variablen und die implementierten Methoden ausgegeben oder nur die öffentlichen Methoden und Variablen angezeigt werden. Dies entspricht der Darstellung der kompletten, durch eine Klasse beschriebene Objektschnittstelle. Das Kommando lautet daher '**DO** printInterface'.

7.9.2 Visualisierung der Vererbungsbeziehungen

Auch zur Darstellung des Vererbungsgraphen sind bereits Hilfsmittel entstanden. Die Grundidee ist die Nutzung eines WWW-Browsers zum Navigieren im Vererbungsgraphen mit Zugriff auf die Klassendefinitionen. Dazu sind mehrere Schritte notwendig.

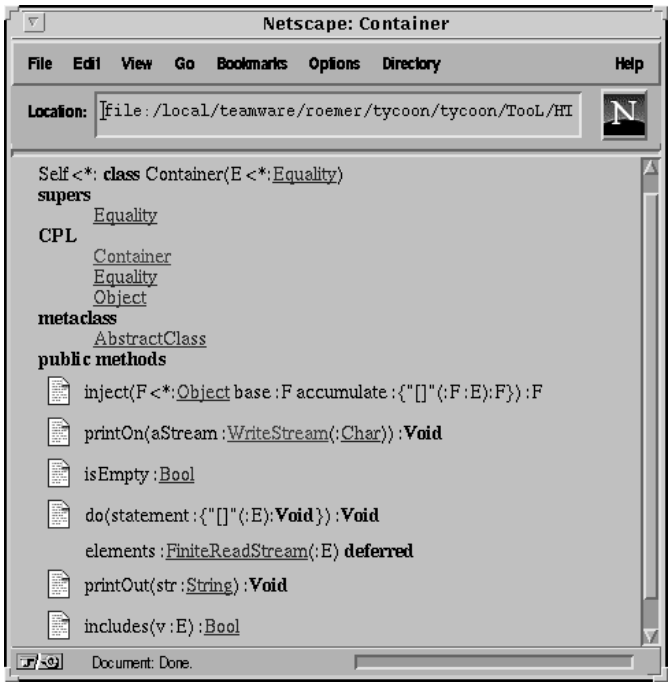

Abbildung 7.27: Eine Klasse als WWW-Seite

Um die Klassendefinitionen als HTML-Seiten verfügbar zu machen, wurde ein weiteres Interpreterkommando eingefügt. Dieses wandelt die, wie bereits oben erwähnt, im Objektspeicher des Tycoon-Systems vorhandenen Signaturen in HTML-Texte um. In diesen HTML-Texten sind die implementierten Methoden mit einem speziellen Text-Icon gekennzeichnet, über welche der Quelltext der Klassendefinition erreichbar ist. Abbildung 7.27 zeigt die Klasse Container mit ihrer Klassenpräzedenzliste. Schlüsselworte sind fett hervorgehoben. Andere auftretende Klassennamen sind als Verweise auf die entsprechenden Klassendefinitionen zu nutzen. Das Text-Icon ist z.B. vor der Methode includes zu erkennen.

Da der Quelltext im Gegensatz zu den Signaturen nicht im Objektspeicher vorhanden ist, muß dieser aus dem Quelltextverzeichnis für die TooL-Klassen beschafft werden. Dies wird

durch ein C-Programm geleistet, welches bei Auswahl des Icons aktiviert wird. Es generiert nun auch für den Quelltext der ausgewählten Klasse eine HTML-Seite und positioniert den Cursor auf die gewünschte Methode.

Es Bleibt noch die Darstellung des Vererbungsgraphen als Einstieg in die nun als HTML-Struktur bereitstehenden Klassenspezifikationen. Hier wurde zunächst versucht, den Graphen mit Hilfe von Grapheneditoren wie *graphed* [Himsolt 89; Himsolt 93] und *daVinci* [Fröhlich, Werner 94] zu entwerfen, um die von diesen angebotenen Graphalgorithmen zur Formatierung nutzen zu können. Insbesondere der Editor *daVinci* wäre durch die Möglichkeit, die Graphen in einer abstrakten Syntax einzulesen, gut geeignet gewesen, eine Aktualisierung des Graphen zu automatisieren.

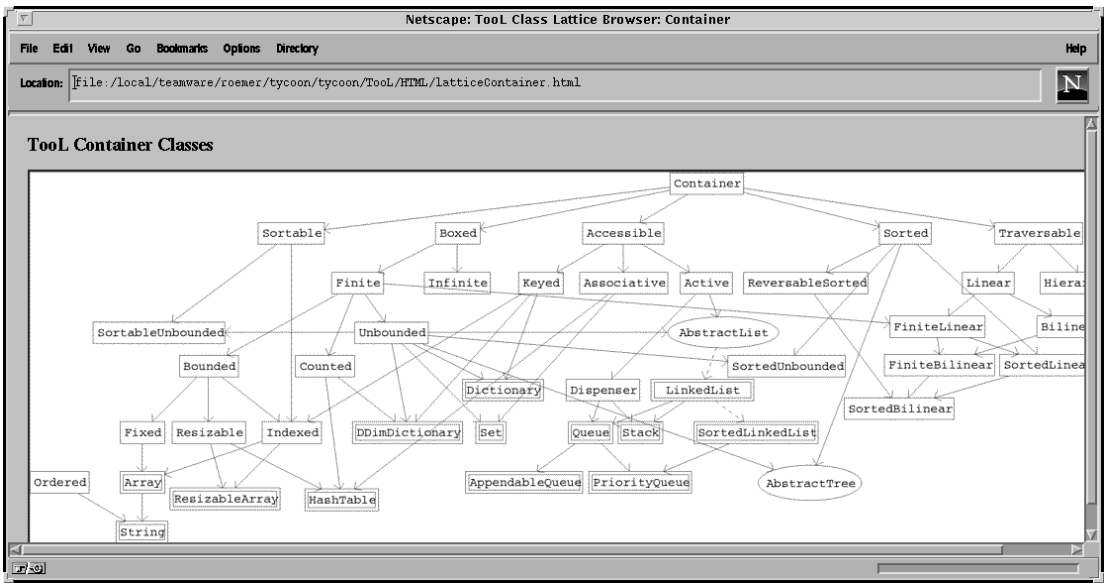

Abbildung 7.28: Die Behälterklassenbibliothek im WWW

Das automatische Layout des Graphen ergab allerdings kein befriedigendes Ergebnis, sodaß die Darstellung schließlich von Hand optimiert wurde. Der so erzeugte Graph wurde als Bild im 'GIF'-Format gespeichert und dann für den Netscape-Browser als 'clickable map' hergerichtet. In Abbildung 7.28 ist diese zu sehen. Die Pfeile symbolisieren die Vererbungsbeziehung. Gestrichelte Pfeile bedeuten eine indirekte Vererbung über nicht abgebildete Klassen. Die implementierten Klassen sind mit doppelter Umrandung abgebildet, ein Kreis verweist auf einen weiteren Teilgraphen. So wurden für die Listen und Bäume separate Graphen gezeichnet, die auf diese Weise erreichbar werden.

Um die Auswahl mit dem Mauszeiger den einzelnen Klassen zuzuordnen, müssen die Positionen der Klassen innerhalb des Bildes als Rechtecke in einer entsprechenden Datei vermerkt werden. Diese Angaben werden natürlich bei einer Umorganisation des Graphen ungültig. Insgesamt ist die angewandte Technik jedoch eine gute Möglichkeit, schnell einen einfachen Klassen-Browser zu schaffen.

Neuere Arbeiten versuchen mit Hilfe der Programmiersprache Java [Gosling, McGilton 95] den Vererbungsgraphen direkt aus den Klassendefinitionen für einen diese Sprache un-

terstützenden WWW-Browser zu erzeugen. Schwierigkeiten bereitet weiterhin die Wahl eines geeigneten Algorithmus für die Darstellung des Graphen. Die Entwicklung eines auf die Formatierung des speziellen Vererbungsgraphen von TooL zugeschnittenen Verfahrens, ist auf diesem Wege jedoch eher möglich als beim Rückgriff auf das automatische Layout der Grapheneditoren.

Kapitel 8

Zusammenfassung und Ausblick

Ausgangspunkt für diese Arbeit war der Wunsch nach Neugestaltung der Bibliothek für Massendatentypen des Tycoon-Systems. Die Aufgabenstellung führte zunächst zur Untersuchung von Qualitätsanforderungen für Softwarebibliotheken. Sodann wurde analysiert, ob die erarbeiteten Anforderungen mit dem bis dahin im Tycoon-System verfolgten und durch die hochsprachliche Schnittstelle TL unterstützten modulorientierten Entwurfsansatz überhaupt zu erreichen sind. Als Alternative wurde ein objektorientierter Ansatz betrachtet. Ergebnis der Analyse war die Entscheidung für den objektorientierten Ansatz, da dieser in Bezug auf Wiederverwendbarkeit und Vereinheitlichung der Schnittstellen deutliche Vorzüge besitzt. Die Umsetzung objektorientierter Konzepte in der Programmiersprache TL führte aber zu komplizierteren Softwarearchitekturen, was auch in anderen Projekten zu beobachten war. Um diese Komplexität in der Bibliotheksentwicklung zu vermeiden und aus dem Bedürfnis nach direkter Unterstützung der objektorientierten Programmierung im Tycoon-System, entstand die experimentelle Programmiersprache TooL als objektorientierte Variante von TL. Nach Analyse einiger existierender objektorientierter Behälterklassenbibliotheken erfolgte dann die Neuentwicklung der Bibliothek in dieser Programmiersprache.

Die Arbeit läßt sich somit grob in zwei Phasen aufteilen, denen die enstandenen Ergebnisse zuzuordnen sind:

1. Analysephase
   (a) Darstellung der Vorteile des objektorientierten gegenüber einem modulorientierten Entwurfsansatz für die Bibliotheksentwicklung.
   (b) Untersuchung bestehender Bibliotheken und Herausstellung ihrer Klassifizierungsprobleme.
2. Entwurfs- und Implementierungsphase
   (a) Entwicklung eines eigenschaftsbezogenen Klassifizierungsschemas für Behälterprotokoll.
   (b) Entwicklung und Implementierung der Behälterklassenbibliothek für TooL.
   (c) Diskussion von Auswirkungen des neuartigen Typsystems von TooL auf den Entwurf und die Benutzbarkeit der Behälterklassenbibliothek
   (d) Skizzierung der Umsetzung von Design-Mustern zur Vereinfachung der Struktur der Behälterklassenbibliothek.

Die einzelnen Ergebnisse werden im folgenden noch einmal aufgeschlüsselt.

Das Ergebnis der Bewertung von modulorientiertem und objektorientiertem Entwurfsansatz für die Entwicklung der Behälterbibliothek war die Bevorzugung des objektorientierten Ansatzes (1a). Die von Modulen durchgeführte Kapselung der Repräsentationsdaten und die vollständige Spezifizierung des Protokolls an ihren Schnittstellen unterstützt eher eine Schichtenarchitektur, welche an den internen Schichtgrenzen oft einen Wechsel der Repräsentation vollzieht. Die Erweiterbarkeit oder Kombination von Funktionalität wird nicht unterstützt. Für die Behälterbibliothek bedeutet dies, daß bei Definition eines neuen Behälters nur auf die Funktionalität vorhandener Module zurückgegriffen werden kann, wenn gleichzeitig die dort gewählte Repräsentation der Daten akzeptabel ist. Ansonsten muß auf die Nutzung verzichtet werden bzw. eine andere Repräsentationsform implementiert werden.

Mit einem objektorientierten Ansatz hingegen läßt sich eine Ergänzung der Bibliothek insbesondere beim Einsatz von Mehrfachvererbung, durch Kombination von vorhandener Funktionalität oder Anpassung von unvollständig implementierter Teilfunktionalität aus abstrakten Klassen realisieren. Weiterhin kann die Struktur des Vererbungsgraphen einen Nutzer der Bibliothek bei der Auswahl eines Behälters unterstützen. Die Spezifikation von Methoden in weit oben im Vererbungsgraphen anzutreffenden Klassen kann als vereinfachte Sicht auf kompliziertere Subklassen genutzt werden. Darüber hinaus wird eine Vereinheitlichung der Behälterschnittstellen erreicht.

Die Analyse bestehender Behälterklassenbibliotheken (1b) führte zu der Feststellung, daß unsaubere Klassifizierungen bei den meisten Bibliotheken in den Subklassen korregiert werden müssen, z.B. durch Ausgabe von Laufzeitfehlermeldungen (Smalltalk: Abschnitt 4.3) oder indem vorher öffentliche Methoden als privat deklariert werden (NIHCL: Abschnitt 4.1.2). Andere Bibliotheken nutzen Sprachkonzepte zur Umbenennung von Methoden, um die Objektschnittstellen anzupassen (Eiffel: Abschnitt 4.4). Beide Varianten verhindern eine vereinfachte Sichtweise auf die Behälter, da sich der Nutzer nicht mehr auf das Vorhandensein bzw. die Anwendbarkeit einer im Zuge der Klassifizierung eingeführten Methode verlassen kann. Dem wurde in dieser Arbeit eine Klassenhierarchie gegenübergestellt, die ohne Umbenennung oder 'Entfunktionalisierung' von Methoden auskommt.

Das entwickelte Klassifizierungsschema (2a) unterstützt die gewünschte Vereinheitlichung der Schnittstellen und die vereinfachende Sicht auf komplexe Behälterklassen über das Protokoll ihrer Vorgänger durch eine eigenschaftsbezogene Klassifizierung der Methoden. Es unterteilt die Behältereigenschaften in allgemeine, von vielen Behältern übernommene und spezielle, einer bestimmten Behälterkategorie zuzuordnende Eigenschaften. Die Zusammenfassung allgemeiner Eigenschaften zu Behälterkategorien und die anschließende erneute Aufteilung des spezielleren Behälterprotokolls führt zu einem allen Behältern einer Kategorie zugeordneten Basisprotokoll, wodurch die vereinfachte Sicht auf die Behälter dieser Kategorie unterstützt wird. Die eigenschaftsbezogene Aufteilung erlaubt eine Selektion geeigneter Behälter für bestimmte Aufgaben bzw. die Auswahl und Kombination geeigneter Methodengruppen zur inkrementellen Entwicklung neuer Behälter.

Die Implementierung der Behälterklassenbibliothek (2b) umfaßt 130 Klassen. Davon sind 48 Klassen abstrakt. In den Klassen sind ca. 580 Methodenimplementierungen (davon ca. 170 in abstrakten Klassen) und ca.60 abstrakte Methodenspezifikationen zu finden. Als grobes Maß für die Wiederverwendung von Klassen kann in Anlehnung an [Dumke 92, S.142] die durchschnittliche Anzahl der Subklassen einer Klasse angesehen werden. Die Behälterklassen werden in ca. 190 Fällen als direkte Oberklassen angegeben. Berücksichtigt man auch die indirekte Vererbung, so sind ca. 780 Zuordnungen zu verzeichnen (Die Klasse Container wurde

nicht berücksichtigt). Dies ergibt für die direkte Wiederverwendung einen Faktor von 1,38 und für die indirekte Wiederverwendung von 6,0 pro Klasse.

Es wurden zum Teil komplexe Algorithmen realisiert, wie z.B. ein mehrdimensionales Wörterbuch (Dictionary). Die Wiederverwendbarkeit von Methoden kann durch die Art und Weise der Implementierung erhöht werden. In der Bibliothek wird daher die Verwendung von Annahmen, die durch Subklassen relativiert werden könnten, vermieden (Abschnitt 7.4.2). Außerdem wird für einige Methoden durch Verteilung der Funktionalität über die Klassifizierungsstruktur die inkrementelle Erweiterung der Algorithmen möglich (Abschnitt 7.4.3). – Die teilweise schwierige Arbeit mit dem Sprachprototyp wurde durch die schon frühzeitig zur Verfügung stehende Typüberprüfung erleichtert. Die Fehlersuche konnte so erheblich beschleunigt werden. Ebenso günstig wirkte sich die frühzeitige Realisierung der einfachen *Klassenbrowser* auf den Entwicklungsprozeß aus.

Die Auswirkungen des neuartigen Typsystems von TooL (2c) lassen sich folgendermaßen zusammenfassen: Mit der Einführung der Ähnlichkeitsrelation als ergänzende Konsistenzbedingung für die Vererbung steigt die Modellierungskomplexität. Der Klassendesigner muß sich entscheiden, in welcher Situation er die jeweilige Relation nutzen will. Die Verwendung der Ähnlichkeitsrelation ist in TooL eng mit der Verwendung des Typbezeichners **Self** verknüpft. Die typsichere Vererbung von Methoden mit Parametern dieses Typs (z.B. binären Methoden) wird ermöglicht. Die Spezialisierung des Objekttyps in derartig beerbten Subklassen verhindert jedoch die Subsumption unter dem Objekttyp der Oberklasse. Dieser eventuelle Nachteil für den Benutzer der Bibliothek kann jedoch durch den Einsatz von Typparametern mit Ähnlichkeitsbindung kompensiert werden. Diese Technik wird bereits in der Behälterklassenbibliothek für die Entwicklung generischer Algorithmen genutzt (Abschnitt 7.4.1). Probleme mit der Subtypheterogenität von Behältern entstehen bei Verwendung einer Ähnlichkeitsbindung für den Elementtypparameter einer Klasse. Sie lassen sich aber für einen konkreten Behälter durch Definition einer (zusätzlichen) Subklasse umgehen, welche ohne weitere Implementierungen die Anpassung an die gewünschte Subtyphierarchie leistet (Abschnitt 7.5).

Der Einsatz von Design-Mustern (2d) und hier insbesondere der Brückentechnik, kann eine Vereinfachung des Vererbungsgraphen bewirken (Abschnitt 7.8). Die Anordnung von Implementierungsvarianten einer Behälterkategorie in einer separaten Hierarchie und die Nutzung des mächtigen abstrakten Implementierungsprotokolls dieser Hierarchie zur Implementierung der Behältervarianten führt zu einer flacheren, die Kombination mit den Implementierungseigenschaften des Klassifizierungsschemas aussparenden Struktur. Die auf die Elementrepräsentation bezogene Implementierungsvererbung wird ebenfalls vermieden. Eine eigenschaftsbezogene Klassifizierung des Behälterprotokolls muß jedoch nicht aufgegeben werden.

Das in TooL entwickelte Klassifizierungsschema, die Strukturierung des Vererbungsgraphen mit der eigenschaftsbezogenen Aufteilung der Methoden und die geschilderten Implementierungstechniken sind durchaus auf andere Sprachen übertragbar. Der Einfluß des speziellen Typsystems von TooL war nicht so groß, daß eine Anpassung etwa an eine andere streng typisierte objektorientierte Sprache unmöglich erscheint. Insofern können die Ergebnisse der Arbeit auch für andere Bibliotheksentwickler interessant sein. Unabhängig davon kann die Klassenbibliothek weiter verbessert werden. In einigen Behälterkategorien ist eine feinere Strukturierung denkbar. Auch die Umsetzung anderer Design-Muster könnte untersucht werden, um die Struktur weiter zu vereinfachen oder flexibler zu gestalten.

Literaturverzeichnis

Abadi, Cardelli 95: Abadi, M. und Cardelli, L. „On Subtyping and Matching“. In: *ecoop95*, 1995.

Ammersaal 95: Ammersaal, L. *C++ for Programmers.* Wiley, 1995.

Antoniou, Sperschneider 88: Antoniou, G. und Sperschneider, V. „Modulare Programme und ihre Verifikation“. Technical report, Universität Ossnabrück, Fachbereich Mathematik/Informatik, Juli 1988.

Atkinson et al. 90: Atkinson, M.P., Richard, P., und Trinder, P.W. „Bulk Types for Large Scale Programming“. In: *Information Systems*, 1990. (also as Rapport Technique Altair 60-90 nov. 1990, GIP Altair, France).

Balzert 82: Balzert, H. *Die Entwicklung von Software-Systemen.* B.I.-Wissenschaftsverlag, Mannheim, 1982.

Beaudouin-Lafon 94: Beaudouin-Lafon, Michel (Hrsg.). *Object-oriented Languages. Basic principles and programmin techniques.* Chapman & Hall, 1994.

Black et al. 87: Black, A., Hutchinson, N., Jul, N., Levy, H., und Carter, L. „Distribution and Abstract Types in Emerald“. *IEEE Transactions on Software Engineering*, Jg. 13, Januar 1987, Nr. 1, S. 65–76.

Black, Hutchinson 90: Black, Andrew P. und Hutchinson, Norman C. „Typechecking polymorphism in Emerald“. Technical Report TR 90-34, Dept. of Computer Science, University of Arizona, Dezember 1990.

Boehm et al. 78: Boehm, Barry, Brown, John R., Kaspar, Hans, Lipow, Myron, MacLeod, Gordon J., und Merrit, Michael. *Characteristics of Software Quality.* North-Holland Publishing Company, 1978.

Booch 94: Booch, Grady (Hrsg.). *Objektorientierte Analyse und Design.* Addison Wesley, 1994.

Borning 82: Borning, Alan. „Multiple Inheritance in Smalltalk-80“. In: *Proc. AAAI 1982*, 1982, S. 234–237.

Bruce 93: Bruce, K. B. „Safe type checking in a statically typed object-oriented language“. In: *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, 1993.

Budgen 89: Budgen, David. *Software Development with Modula-2.* Addison-Wesley, 1989.

Card et al. 86: al., D. Card et. „An Empirical Study of Software Design Practices". *IEEE Transactions on Software Engineering*, Februar 1986, S. 264–271.

Cardelli, Mitchell 89: Cardelli, L. und Mitchell, J.C. „Operations on Records". Digital Systems Research Center Reports 48, DEC SRC Palo Alto, August 1989.

Cardelli, Wegner 85: Cardelli, L. und Wegner, P. „On Understanding Types, Data Abstraction, and Polymorphism". *ACM Computing Surveys*, Jg. 17, Dezember 1985, Nr. 4, S. 471–522.

Cardelli 84: Cardelli, L. „A Semantics of Multiple Inheritance". In: Kahn, G., MacQueen, D.B., und Plotkin, G. (Hrsg.), *Semantics of Data Types*, Bd. 173. Springer Verlag, 1984, S. 51–67.

Cavano, McCall 78: Cavano, J. P. und McCall, J. A. „A Framework for the Measurement of Software Quality". In: *The Proceedings of the ACM Software Quality Assurance Workshop*, November 1978, S. 133–139.

Cho 80: Cho, Chin-Kuei. *An Introduction to Sofware Quality Control.* John Wiley & Sons, 1980.

Cin et al. 89: Cin, Dal, Lutz, und Risse. *Programmierung in Modula-2.* 4. Auflage, B. G. Teubner, Stuttgart, 1989.

Coad, Nicola 94: Coad, P. und Nicola, J. (Hrsg.). *Objektorientierte Programmierung.* Prentice Hall, 1994.

Coad, Yourdon 91: Coad, Peter und Yourdon, Edward (Hrsg.). *Object-Oriented Design.* Yourdon Press, 1991.

Coad, Yourdon 94: Coad, Peter und Yourdon, Edward (Hrsg.). *OOA. Objektorientierte Analyse.* Prentice Hall, 1994.

Cohen 72: Cohen, Alan. „Modular Programs: Defining the Module". *Datamation*, Januar 1972, S. 34–37.

Coleman et al. 94: Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., und Jeremaes, P. *Object-Oriented Development. The Fusion Method.* Prentice Hall Object-Oriented Series. Prentice Hall, 1994.

Cook 91: Cook, S. „Object-oriented techniques: scope, principles, languages, methods and strategies". In: *Object-oriented Programming Systems. Tools and applications*, 1991, S. 23–45.

Cook 92: Cook, William R. „Interfaces and Specifications for the Smalltalk-80 Collection Classes". In: *Conference on Object-oriented Programming Systems, Languages, and Applications*, 1992, S. 1 – 15.

Coplien 92: Coplien, James O. *Advanced C++. Programming Styles and Idioms.* Addison-Wesley, 1992.

Crosby 72: Crosby, P. B. *Quality Is Free.* McGraw-Hill, New York, 1972.

Decker, Hirshfield 95: Decker, R. und Hirshfield, S. *The Object Concept. An Introduction to Computer Programming Using C++.* PWS Publishing Company, 1995.

Denert 79: Denert, E. „Software-Modularisierung". *Informatik-Spektrum*, 1979, S. 204–218.

DeRemer, Kron 76: DeRemer, Frank und Kron, Hans H. „Programming-in-the-Large Versus Programming-in-the-Small". *IEEE Transactions on Software Engineering*, Jg. 2, Juni 1976, Nr. 2, S. 80–86.

Dumke 92: Dumke, Reiner. *Softwareentwicklung nach Maß.* Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1992.

Dunn 93: Dunn, Robert H. *Software-Qualität.* Carl Hanser Verlag, München, 1993.

Fröhlich, Werner 94: Fröhlich, M. und Werner, M. „The Graph Visualization System daVinci - A User Interface for Applications". Technical Report 5/94, Department of Computer Science, University of Bremen, September 1994.

Gamma et al. 95: Gamma, Erich, Helm, Richard, Johnson, Ralph, und Vlissides, John. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing. Addison-Wesley Publishing Company, 1995.

Gawecki, Matthes 95: Gawecki, A. und Matthes, F. „TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification". FIDE Technical Report FIDE/95/135, FIDE Coordinator, Department of Computing Sciences, University of Glasgow, 1995.

Gawecki, Matthes 96: Gawecki, Andreas und Matthes, Florian. „Integrating Subtyping, Matching, and Type Quantification: A Practical Perspective". In: *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96*, Linz, Österreich, Juli 1996. Springer Verlag.

Gladney 82: Gladney, H. M. „Towards Modular Programs – A CONCISE Overview". Technical Report RJ3477, IBM Research Laborarory, San Jose, California 95193, Oktober 1982.

Goldberg, Robson 83: Goldberg, A. und Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.

Goldberg 84: Goldberg, A. (Hrsg.). *Smalltalk-80 The interactive Programming Environment.* Addison-Wesley, Reading, Ma, 1984.

Goos 74: Goos, Gerhard. „Zum Begriff des Programmoduls". Technical report, Universität Karlsruhe, Fakultät für Informatik, 1974.

Gorlen 90: Gorlen, Keith E. „NIH Class Library Reference Manual". Technical report, National Institute of Health, Juli 1990.

Gosling, McGilton 95: Gosling, James und McGilton, Henry. „The Java Language Environment: A White Paper". Technical report, Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-110 U.S.A., 1995.

Hennicker 92: Hennicker, Rolf. „Behavioural Specification and Implementation of Modular Software Systems". Technical Report MIP-9203, Universität Passau, Fachbereich für Mathematik und Informatik, April 1992.

Himsolt 89: Himsolt, M. „GraphEd: An Interactive Graph Editor". In: *Proceedings STACS 89, Lecture Notes in Computer Science*, Bd. 349, 1989, S. 532–533.

Himsolt 93: Himsolt, M. „A View to Graph Drawing Algorithms through GraphEd". In: *Proceedings of the International Workshop on GraphDrawing 93, Sevres (France)*, 1993, S. 117–118.

Höcker et al. 84: Höcker, Hanns, Itzfeld, Wolf D., Schmidt, Monika, und Timm, Michael. „Comparative Descriptions of Software Quality Measures". Technical Report 81, GMD-Studien, Maerz 1984.

Hoffmann 87: Hoffmann, H. *Smalltalk verstehen und anwenden.* Hanser, München, 1987.

Josuttis 94: Josuttis, N. *Objektorientiertes Programmieren in C++.* Addison-Wesley, 1994.

Jung 95: Jung, Tanja. „Softwarebibliotheken: Entwurf und Dokumentation am Beispiel von Tycoon". Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Datenbanken und Informationssysteme, 1995.

Kilberth et al. 94: Kilberth, K., Gyczan, G., und Züllighoven, H. *Objektorientierte Anwendungsentwicklung.* Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1994.

Klint 86: Klint, P. „Modularization and reusability in current programming languages". Technical Report CS-R8635, CWI Centrum voor Wiskunde en Informatica, Computer Science/Department of Software Technology, 1986.

Knudsen et al. 93: Knudsen, J. L., Löfgren, M., Lehrmann-Madsen, O., und Magnusson, B. *Object-Oriented Environments: The Mjølner Approach.* Prentice Hall, 1993.

Koch 96: Koch, G. *HTML. Einführung und Referenz.* Addison-Wesley, 1996.

Kofler 93: Kofler, Thomas. „Robust Iterators in ET++". *Structured Programming*, Jg. 14, 1993, Nr. 2, S. 62–85.

Lientz, Swanson 79: Lientz, B. P. und Swanson, E. B. „Software Maintenance: A User/Manager Tug of War". *Data Management*, April 1979, S. 26–30.

Liggesmeyer 90: Liggesmeyer, Peter. *Modultest und Modulverifikation.* B.I.-Wissenschaftsverlag, Mannheim, 1990.

Lippman 89: Lippman, Stanley B. *C++ Primer.* Addison-Wesley, 1989.

Lotter, Römer 94: Lotter, Björn und Römer, Thorsten. „Ein Wörterbuch beliebiger Dimension in Tycoon". Studienarbeit, Universität Hamburg Arbeitsbereich Datenbanken und Informationssysteme, 1994.

Mathiske et al. 93: Mathiske, Bernd, Matthes, Florian, und Müßig, Sven. „The Tycoon System and Library Manual". Technical report, Universität Hamburg, Arbeitsbereich DBIS, Dezember 1993.

Matthes et al. 94: Matthes, F., Müßig, S., und Schmidt, J.W. „Persistent Polymorphic Programming in Tycoon: An Introduction". FIDE Technical Report Series FIDE/94/106, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, August 1994.

Matthes, Schmidt 91: Matthes, F. und Schmidt, J.W. „Bulk Types: Built-In or Add-On?". In: *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece.* Morgan Kaufmann Publishers, 1991, S. 33–54. (also appeared as FIDE Technical Report 91/20, Department of Computing Science, University of Glasgow).

Matthes, Schmidt 92: Matthes, F. und Schmidt, J.W. „Definition of the Tycoon Language TL – A Preliminary Report". DBIS Tycoon Report 062-92, Fachbereich Informatik, Universität Hamburg, Oktober 1992.

Matthes 92: Matthes, F. *Generische Datenbankprogrammierung: Sprachliche und Architektonische Grundlagen.* Dissertation, Fachbereich Informatik, Universität Hamburg, September 1992.

Matthes 93: Matthes, Florian. *Pesistente Objektsysteme.* Springer Verlag, Berlin, 1993.

Maynard 72: Maynard, Jeff. *Modular Programming.* Auerbach Publishers, London, 1972.

McCall et al. 81: McCall, J., Markham, D., Stosick, M., und McGindly, R. „The Automated Measurement of Software Quality". In: *Software Quality Assurance*, 1981, S. 388–394.

Mehlhorn 84: Mehlhorn, Kurt. *Multi-dimensional Searching and Computational Geometry.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1984.

Meyer 88: Meyer, Bertrand. *Object Oriented Software Construction.* Prentice Hall, 1988.

Meyer 89: Meyer, Bertrand. „From Structured Programming to Object-Oriented Design: The Road to Eiffel". *Structured Programming*, Jg. 10, 1989, Nr. 1, S. 19–39.

Meyer 90: Meyer, Bertrand. „Lessons from the Design of Eiffel". *Communications of the ACM*, Jg. 33, 1990, Nr. 9.

Meyer 92: Meyer, Bertrand (Hrsg.). *Eiffel. The Language.* Prentice Hall, 1992.

Meyer 93: Meyer, Bertrand. „An overview over the thechnology". In: Meyer, Bertrand und Nerson, Jean-Marc (Hrsg.), *Object-Oriented Applications.* Prentice Hall, 1993, S. 1–30.

Meyer 94a: Meyer, Bertrand (Hrsg.). *Eiffel. The Libraries.* Prentice Hall, 1994.

Meyer 94b: Meyer, Bertrand (Hrsg.). *Reusable software. The base OO component libraries.* Prentice Hall, 1994.

Murer et al. 94: Murer, Stephan, Omohundro, Stephen, Stoutamire, David, und Szyperski, Clemens. „Iteration Abstraction in Sather". Technical report, International Computer Science Institute, Berkeley, 1994.

Niederée 92: Niederée, Claudia. „Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung.". Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Datenbanken und Informationssysteme, 1992.

Nord 92: Nord, Robert Louis. *Deriving and Manipulating Module Interfaces.* Dissertation, Carnegie Mellon University, School of Computer Science, Pittsburgh, Mai 1992.

Otto 91: Otto, Werner. „Objektorientiertes Programmieren". In: Held, Gerhard (Hrsg.), *Objektorientierte Systementwicklung.* Siemens Nixdorf Informationssysteme AG, 1991, S. 53–95.

Parnas 71: Parnas, D.L. „Information Distribution Aspects of Design Methodology". In: *Information Processing. Proceedings of the IFIP Congress*, 1971.

Parnas 72a: Parnas, D.L. „On the Criteria To Be Used in Decomposing Systems into Modules". *Communications of the ACM*, Dezember 1972, S. 1053–1058.

Parnas 72b: Parnas, D.L. „A Technique for Software Module Specification with examples". *Communications of the ACM*, Mai 1972, S. 330–336.

Pree 95: Pree, Wolfgang. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley Publishing Company, 1995.

Reiser, Wirth 94: Reiser, Martin und Wirth, Niklaus. *Programmieren in Oberon.* Addison-Wesley, Bonn, 1994.

Schmidt et al. 88: Schmidt, J.W., Eckhardt, H., und Matthes, F. „DBPL Report". DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, 1988.

Schmidt, Matthes 91: Schmidt, J. W. und Matthes, F. „Bulk Types: Built-In or Add-On?". In: *dbpl91*, 1991.

Schoett 81: Schoett, Oliver. „Ein Modulkonzept in der Theorie abstrakter Datentypen". Technical Report IfI-HH-B-81/81, Universität Hamburg, Fachbereich Informatik, 1981.

Shen 94: Shen, Kai. „Zwei Implementierungen von B–Bäumen im Vergleich". Studienarbeit, Universität Hamburg Arbeitsbereich Datenbanken und Informationssysteme, 1994.

Siberski 95: Siberski, W. „Meta-Konzepte in Objektorientierten Sprachen - Unter Besonderer Berücksichtigung von C++". Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Juni 1995.

Snyder 86: Snyder, Alan. „Encapsulation and Inheritance in Object-Oriented Programming Languages". In: *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, November 1986, S. 38–45. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

Stolze 73: Stolze, Lothar. „Allgemeines Testsystem für Modultest". Technical Report 1051-2 (1273), SCS Scientific Control Systems GmbH, 1973.

Stroustrup 92: Stroustrup, Bjarne. *Die C++ Programmiersprache.* Addison-Wesley, 1992.

Stroustrup 94: Stroustrup, Bjarne. *Design und Entwicklung von C++.* Addison-Wesley, 1994.

Switzer 92: Switzer, Robert. *Eiffel. An Introduction.* Prentice Hall, 1992.

Traub 95: Traub, Horst-Peter. „Objektorientierte Behälterklassen-Bibliotheken - Konzepte, Entwurf und Implementation". Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 1995.

Weinberg 92: Weinberg, Gerald M. *Quality Software Management. Volume 1, System Thinking*. Dorset House Publishing, New York, 1992.

Wetzel 94: Wetzel, Ingrid. *Programmieren mit STYLE: Über die systematische Entwicklung von Programmierumgebungen.* Dissertation, Fachbereich Informatik, Universität Hamburg, Juni 1994.

Wirfs-Brock et al. 90: Wirfs-Brock, R., Wilkerson, B., und Wiener, L. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

Wirth 71: Wirth, Niklaus. „Program development by stepwise refinement". *Communications of the ACM*, April 1971, S. 221–227.

Wirth 88: Wirth, Niklaus. *Programming in Modula-2*. 4. Auflage, Springer Verlag, New York, 1988.