Studienarbeit

Realisierung einer Behälterbibliothek in Java
Die *j*ConLib

Holger Bohlmann
Schimmelmannstr. 2
22926 Ahrensburg
E-Mail: 3bohlman@informatik.uni-hamburg.de

Juni 1998

Betreuung: Prof. Dr. Heinz Züllighoven

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Inhaltsverzeichnis

I. Einleitung 5

1.1 Übersicht 5
1.2 Dank an… 5
1.3 Semantisches 6

II. Die ConLib 7

2.1 Behälterhierarchie 7
2.2 Implementationsvarianten von Behältern 9
2.3 Konstruktionsmechanismen 10
2.3.1 Statische Typsicherheit durch Templates 10
2.3.2 Entwurfsmuster in der ConLib 12
2.4 Das Cursor-Konzept 12

III. Von C++ zu Java: Gemeinsamkeiten und Unterschiede 16

3.1 C++ - Templates für Polymorphie 16
3.1.1 Pizza - Eine Java-Erweiterung 17
3.2 Java - Oberklasse Object 17
3.3 Java - Typecast 18
3.4 Java - Meta-Informationen 18
3.5 C++ - Zusicherungen 19
3.6 Java - Garbage Collection 19
3.7 Java - Vererbung und Interfaces 20
3.8 Java - Serialisierbarkeit 20
3.9 Java - Synchronisation 20
3.10 Allgemeines zu Java und C++ 21

IV. Realisierung der *j*ConLib 22

4.1 Die *j*ConLib-Behälterhierarchie 22
4.2 Typsicherheit 23
4.2.1 Erzeugung der Behälter mit Klassenobjekt 23
4.2.2 Dynamische Typsicherheit als Vorbedingung 24
4.2.3 Typecasts bei der Rückgabe 25
4.2.4 Probleme der dynamischen Typisierung 25
4.3 Die Behälterfabrik 26
4.4 Der stabile *j*ConLib-Cursor 27
4.4.1 Konzeptuelle Probleme der Stabilität 29
4.4.2 Typisierung des Cursors 30
4.5 Interfaces 30
4.6 Integration in das Framework 31
4.6.1 Package-Aufteilung 31
4.6.2 Dokumentation 32
4.7 Die Portierung 32

V. Vergleich mit anderen Behälterbibliotheken 33

5.1 Das java.util-Package 33
5.2 Vorgehensweise des Vergleiches 33
5.3 Collection API des JDK 1.2 34
5.3.1 Aufbau der Behälterhierarchie 34
5.3.2 Methoden zur Manipulation der Behälter 35
5.3.3 Cursor- bzw. Iterator-Konzepte 35
5.3.4 Einbindung technischer Gegebenheiten 35
5.3.5 Handhabung und Benutzung der Collection API 36
5.4 JGL von Objectspace 37
5.4.1 Aufbau der Behälterhierarchie 37
5.4.2 Methoden zur Manipulation der Behälter 39
5.4.3 Iterator- bzw. Cursor-Konzepte 39
5.4.4 Einbindung technischer Gegebenheiten 40
5.4.5 Handhabung und Benutzung der JGL 40
5.4.6 Die JGL als Portierung der STL 41
5.5 Abschließender Vergleich 42

Literaturverzeichnis 44

I. Einleitung

Am Arbeitsbereich Softwaretechnik wurde in der Programmiersprache Java ein Framework nach dem Werkzeug-Automat-Material-Ansatz (WAM-Ansatz) entwickelt [Lippert97][JWAM98]. Dieses Frameworks namens ‚JWAM' unterstützt die Werkzeugkonstruktion aus dem WAM-Ansatz [GKZ93], besitzt eine abstrahierendes GUI-Konzept und ein umfangreiches Nachrichtenvermittlungssystem. Im Zuge dieser Entwicklung wurde deutlich, daß für den Einsatz des Frameworks, erstmals in der Lehre im Rahmen eines Projektseminars, eine Behälterbibliothek vonnöten war. Die Behälterfunktionalität der standardmäßig im Java-Development-Kit (JDK) vorliegenden Java-Bibliotheken war bis einschließlich Version 1.1 sehr rudimentär und besaß kaum softwaretechnische Konzepte. Da Behälter innerhalb des Frameworks und vor allem von Benutzern des Frameworks benötigt wurden, sollte eine Behälterbibliothek als Bestandteil des JWAM-Frameworks entwickelt werden. Am Arbeitsbereich Softwaretechnik wurde schon im Rahmen einer Diplomarbeit von Horst-Peter Traub [Traub96] eine Behälterbibliothek in C++ entworfen wurde: die ConLib. Diese Behälterbibliothek besitzt zahlreiche softwaretechnische Konzepte und hat sich im Einsatz in einem früheren C++-Rahmenwerk bewährt: Die ConLib ist Bestandteil der ‚BibV30' (letzte Version), eines am Arbeitsbereich Softwaretechnik entwickelten und eingesetzten Frameworks. So lag es nahe, mit den erarbeiteten Konzepten eine Behälterbibliothek in Java zu implementieren, die ConLib sozusagen nach Java zu ‚portieren'. Diese neue Bibliothek namens *j*ConLib sollte ein Bestandteil des JWAM-Frameworks werden und in der Benutzung dem Original ähneln, damit die guten Erfahrungen der alten Bibliothek im JWAM-Framework weitergeführt werden konnten.

1.1 Übersicht

In meiner Arbeit gehe ich zuerst auf die von Horst-Peter Traub erarbeiteten Konzepte ein und erläutere diese kurz, um einen Überblick zu gewinnen. Für weitere, vertiefene Informationen sei auf die Diplomarbeit von Horst-Peter Traub verwiesen. Im nächsten Teil mache ich dann den Schritt von C++ zu Java unter dem Gesichtspunkt, was für Gemeinsamkeiten und was für Unterschiede zwischen den beiden Sprachen bestehen. Wichtig ist mir hierbei, welche Sprachunterschiede Änderungen im Design der Bibliothek bewirken. Im nächsten Kapitel erläutere ich das Design der *j*ConLib ausführlich. Im fünften Kapitel vergleiche ich die *j*ConLib mit anderen Behälterbibliotheken in Java und gehe dabei hauptsächlich auf konzeptionelle Unterschiede und Gemeinsamkeiten ein.

An den Anfängen der Kapitel werde ich, wie in diesem Unterkapitel, kurz den folgenden Inhalt erläutern und wesentliche Punkte aufgreifen. Zum Verständnis der Arbeit setze ich C++- und Java-Sprachkenntnisse voraus. Soweit es zur Erläuterung wichtiger Konzepte geht, greife ich die entsprechenden Programmiersprachenkonzepte auf.

1.2 Dank an…

Ich möchte allen danken, die an der Entstehung der *j*ConLib beteiligt waren und mitgewirkt haben: Niels Fricke, Carola Lilienthal, Martin Lippert, Stefan Roock, Henning Wolf und noch den weiteren Mitgliedern der ‚JWAM-Diskussions-Gruppe' und natürlich Horst-Peter Traub für seine Ratschläge und für die Vorbild-Bibliothek ‚ConLib'.Außerdem möchte ich mich bei den Kommilitonen aus dem PJS „Corba, Java & WWW" im WS97/98 bedanken, die erkannt haben, daß die *j*ConLib doch ein *wenig* besser ist, als ‚Arrays' und ‚Vektoren'.

1.3 Semantisches

Sind Textstellen mit dem `Courier-Font` geschrieben, so handelt es sich um Klassen oder Methoden aus dem entsprechenden Programmcode, wobei Methoden noch zusätzlich durch nachfolgende Klammern gekennzeichnet sind, z.B. ‚`newList()`'. Code-Beispiele sind ebenfalls im `Courier-Font` gesetzt. Die Darstellung der Klassen und Vererbungsbeziehungen in den Klassendiagrammen für Java als auch für C++ sind nach dem folgenden Schema gestaltet:

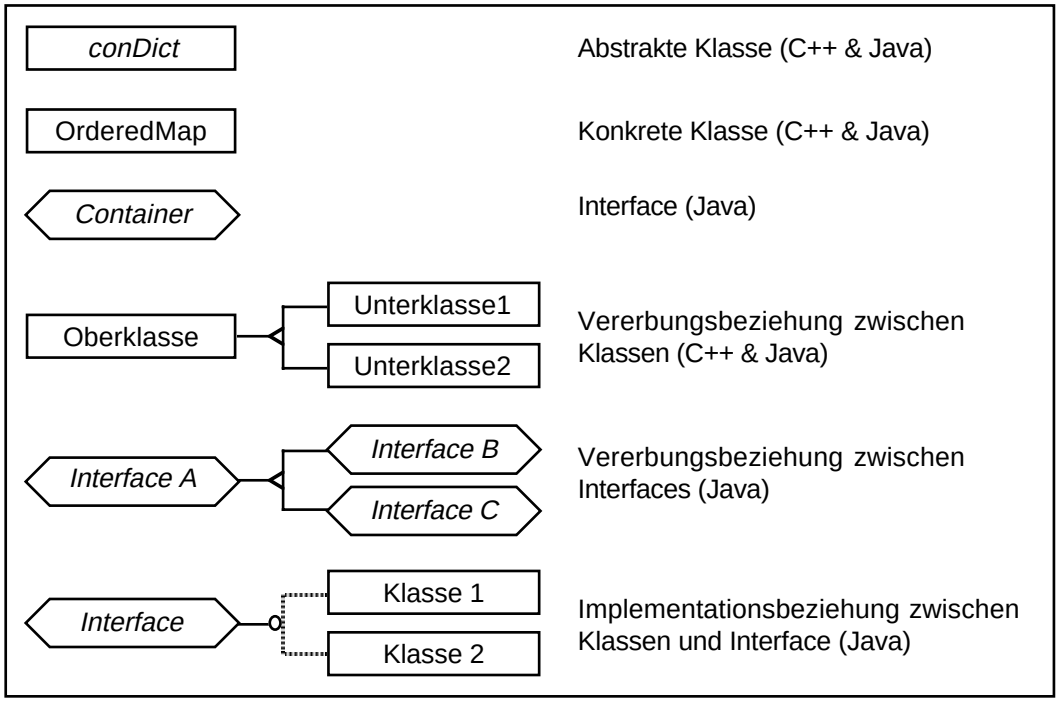

Abbildung 1.1 - Übersicht der Klassendarstellungen in den Abbildungen

II. Die ConLib

Die wesentliche Grundphilosophie der ConLib ist, ein Behältermodell fachlich zu erstellen und danach eine Behälterbibliothek technisch zu modellieren. Dabei wurden Behälter zuerst auf ihre fachlichen Eigenschaften hin untersucht, wie auf sie zugegriffen wird, wie und ob sie überhaupt geordnet sind oder ob sie über Duplikate verfügen dürfen. Aus dieser fachlichen Modellierung entstand eine Behälterhierarchie, auf die ich im ersten Teil näher eingehe. Ein weiteres Konzept ist die Trennung der Behältervarianten von der eigentlichen Implementation. Die Behälter sollten das gleiche Verhalten besitzen, egal wie sie implementiert wurden. Nur bei der Erzeugung wird der Benutzer mit der eigentlichen Implementation in Kontakt gebracht, indem er eine passende wählt, die für seine Zwecke optimal ist: Ein Behälter, der aus einer verketteten Liste besteht, aus einem Array oder gar einer Hashtable.

Weiter erläutere ich dann einige Konstruktionsmechanismen der ConLib. So wurde, z.B. für den Gleichheitstest über Elemente, das Strategie-Entwurfsmuster [GHJ94] verwendet, um die Bibliothek für andere Vergleichsstrategien offen zu lassen. Ebenfalls möchte ich hier auf die Benutzung von C++-Templates eingehen, die in der ConLib eine statische Typsicherheit der Behälter garantieren. Die Behälter sind polymorph und werden durch Templates zur Übersetzungszeit typisiert (Parametrischer Polymorphismus).

Abschließend gehe ich dann auf einen weiteren, wesentlichen Baustein der ConLib ein: das Konzept des Cursors. Im Vergleich zu anderen Bibliotheken, auch der im letzten Kapitel dieser Studienarbeit vorgestellten Bibliotheken in Java, besitzt die ConLib keine klassischen Iteratoren, die einfach über Behälter laufen, sondern dauerhafte Cursor, die ein Element referenzieren, sich frei über die Elemente im Behälter bewegen können und die robust, d.h. vor Veränderungen im Behälter geschützt, sind.

2.1 Die Behälterhierarchie

Die Konstruktion der ConLib ist Horst-Peter Traub von der fachlichen Seite angegangen. Er hat sich zuerst die Frage gestellt, wie mit einem Behälter umgegangen wird, was es überhaupt für Behälterarten gibt und wie sie zueinander in Beziehung stehen. Was für technische Datenstrukturen zur Realisierung verwendet werden, ist kein Punkt, der hier beachtet wird. Es sollten Behälter nach ihren fachlichen Umgang klassifiziert und modelliert werden, nicht Datenstrukturen im Computer. Horst-Peter Traub hat eine erste Klassifizierung der Behälter nach ihrer Zugriffsart vorgenommen und eine erste Spezialisierung entwickelt:

- **Stack**
  Ein Behälter, der dem Stapel-Prinzip folgt: Die Elemente können nur an der obersten Position eingefügt und auch nur dort wieder entfernt werden. Der Behälter hat keine fachliche Begrenzung in der Größe.
- **Queue**
  Dieser Behälter folgt dem Schlangen-Prinzip: An einem Ende werden Behälterelemente eingefügt und am anderen Ende werden diese Element wieder aus dem Behälter entfernt. So können die Behälterelemente immer nur in der Reihenfolge entfernt werden, in der sie auch in den Behälter getan wurden.

- **Array**
  Ein Array-Behälter besitzt eine vom Benutzer festgelegte, maximale Anzahl von Elementen. Auf diesen Behälter kann über einen Index frei zugegriffen werden. Die Positionen im Behälter müssen nicht unbedingt besetzt sein, so daß es freie, unbesetzte Positionen geben kann.
- **Collection**
  Eine Collection ist in der allgemeinsten Form ein Behälter, in dem Elemente auf einfache Weise eingefügt und wieder entfernt werden können, ohne daß der Benutzer eine fachliche Maximalanzahl an Behälterelementen angeben muß.
- **Table**
  Eine Table hat die Eigenschaften einer Collection, nur daß es für jedes Behälterelement ein Schlüsselelement gibt, mit dem auf die eigentlichen Behälterelemente zugegriffen werden kann (Wert/Schlüssel-Paare).

Neben der Zugriffsart lassen sich Behälter noch weiter klassifizieren. So ist bei einer Collection noch nicht die Frage geklärt, ob dieser Behälter geordnet ist oder nicht. Bei einer Ordnung stellt sich außerdem die Frage, ob diese Ordnung durch die Elemente selbst oder explizit durch den Benutzer festgelegt wird. Ist der Behälter ungeordnet, d.h. Elemente sind nicht über eine Positionsangabe erreichbar, so ist das Vorhandensein von mehrfach vorkommenden Behälterelementen von Bedeutung (Duplikate). Eine Queue könnte noch um die Eigenschaft erweitert werden, daß das Einfügen und Herausnehmen von Elementen an beiden Enden möglich ist. Aus alle diesen Behälterarten läßt sich eine einfache Hierarchie entwickeln, die von einem allgemeinen Behälter zu immer mehr spezialisierten Behältern führt. Diese Behälterhierarchie schlägt sich in der ConLib in der folgenden Klassenhierarchie nieder:

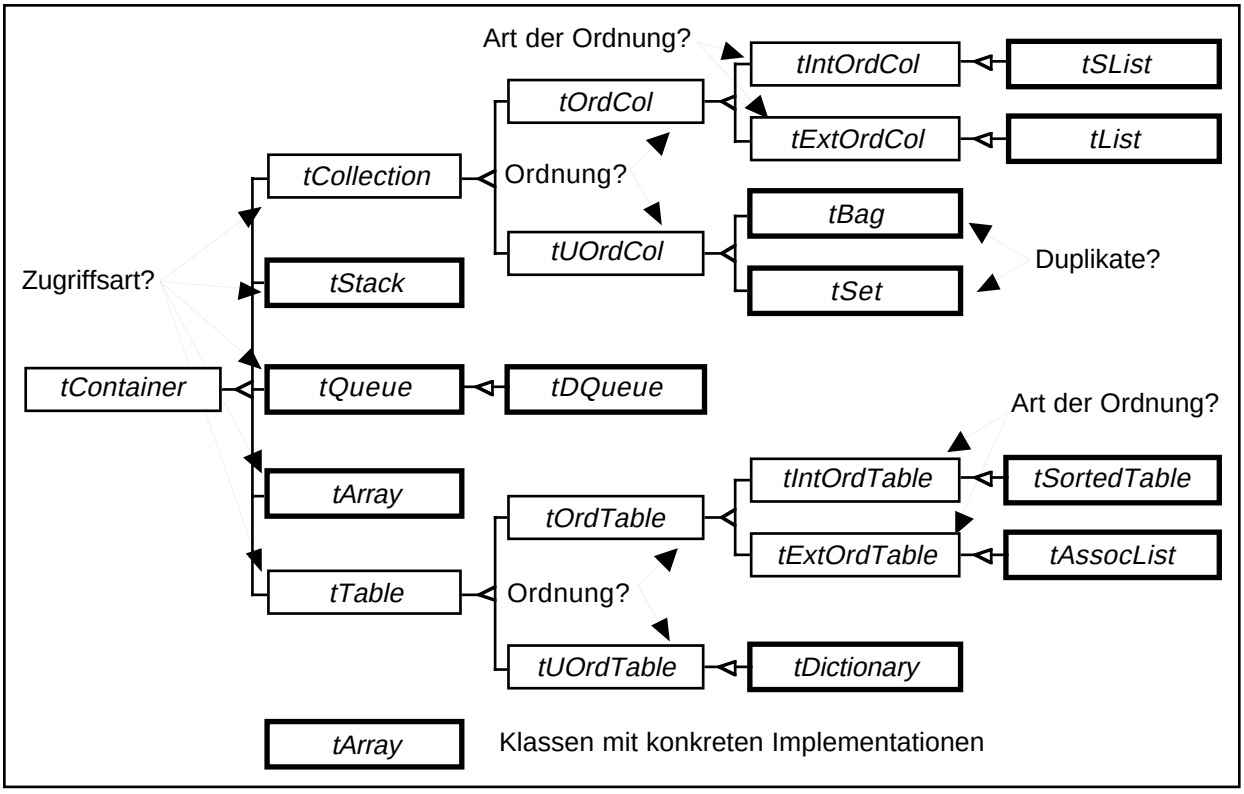

Abbildung 2.1 - Ausschnitt der ConLib-Klassenhierarchie

Aus der Abbildung 2.1 ist `tContainer` als die Oberklasse aller Behälter zu erkennen, danach erfolgt eine Gliederung in die verschiedenen Spezialisierungen. Zu beachten ist, daß alle fachlichen Klassen abstrakt sind, d.h., es kann von ihnen kein Exemplar erzeugt werden. Denn die konkrete Implementation, was für eine technische Datenstruktur der Behälter nun besitzt, wird in der ConLib durch Implementationsklassen realisiert, die von den abstrakten Klassen erben (Dazu mehr im nächsten Unterkapitel). Die dick umrahmten Klassen verfügen über solche Implementationsklassen, d.h., man kann sich eine konkrete Implementation einer dick umrahmten Klasse erzeugen und damit einen konkreten Behälter mit den entsprechenden fachlichen Eigenschaften.

Bei der Klassenhierarchie wurde bewußt Mehrfachvererbung vermieden, um eine klare Spezialisierung der Behälter zu erhalten und den Sematikerhalt der Behälter zu sichern. So sollte in der Klassenhierarchie ein ‚vertikalen Semantikerhalt' existieren: Die Bedeutung der Umgangsformen behalten beim Hinabsteigen durch die Behälterhierarchie ihre Bedeutung. Es wurde vermieden, daß Umgangsformen in weiter spezialisierten Behältern eine neue Bedeutung bekommen oder gar für den Benutzer nicht mehr verfügbar sind und eine Fehlermeldung provozieren. Die Oberklassen beinhalten somit gemeinsame, abstrahierende Umgangsformen. Ebenfalls sollte ein horizontaler Semantikerhalt sichergestellt sein. Ähnliche klingende Umgangsformen in unterschiedlichen Behältern, die nicht in einer Vererbungsrelation stehen, haben eine ähnliche Semantik. So setzen sich einige Namen der Umgangsformen in der ConLib nach einem einfachen Schema zusammen: Umgangsformen zum Entfernen von Elementen fangen mit der Silbe ‚Remove' gefolgt von weiteren Angaben zur Position an. So gibt es in der Klasse `tArray` die Methode `RemoveAtIndex()`, die ein Element an einer bestimmten Indexposition entfernt. In der Klasse `tCollection` gibt es eine Methode `Remove()`, die ein anzugebendes Element aus dem Behälter entfernt. Die Umgangsform `RemoveAtIndex()` wäre in der `tCollection` nicht möglich, da dieser Behälter über keine Indizierung verfügt.

2.2 Implementationsvarianten von Behältern

Ein weiteres Konzept der ConLib ist die Trennung der Implementation von den fachlich modellierten Behältern. So handelt es sich bei den in Abb. 2.1 vorgestellten Klassen um abstrakte Klassen, von denen kein Exemplar erzeugt werden kann. Sie bilden eine Hierarchie von Behältervarianten und spezifizieren sie, je tiefer man den Baum hinabsteigt. Die Vererbungshierarchie in Abb. 2.1. zeigt aber nur einen Ausschnitt, denn für jede dick markierte Klasse gibt es noch Implementationsklassen, die die technische Realisierung des Behälters übernehmen. In den abstrakten Oberklassen werden Methoden zwar soweit wie möglich implementiert, jedoch gibt es einige Methoden, die nur abstrakt definiert werden können und von der konkreten Implementation erfüllt werden müssen. So sind in der ConLib beispielsweise für `tList` zwei technische Implementationsmöglichkeiten vorhanden: eine verkettete Liste und ein Array.

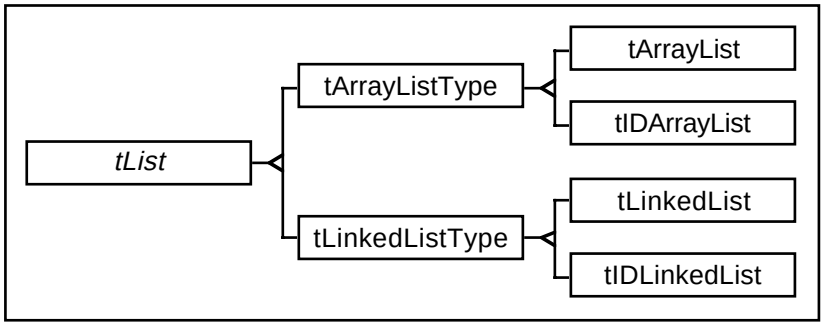

Abbildung 2.2 - Unterklassen von tList

Beide Varianten, `tArrayListType` und `tLinkedListType`, lassen sich erzeugen und erben von `tList`. Es sind verschiedene technische Realisierungen: einmal die Implementation als verkettete Liste und einmal als Array. `tList` definiert für sie die fachliche Schnittstelle, deren Verhalten beide Klassen erfüllen. Bei den Klassen `tArrayList` und `tIDArrayList` handelt es sich um eine weitere Spezialisierung, die angibt, wie die Elemente im Behälter verglichen werden. `tArrayList` führt einen Test nach Wertgleichheit der Objekte durch, während `tIDArrayList` eine Behältervariante ist, die auf Objektidentität testet.

Für alle in Abb. 2.1 dick umrahmten Klassen gibt es Implementationsklassen, allerdings noch mit weiteren technischen Realisierungen. Für `tSet`, den ungeordneten Behälter ohne Duplikate, ist noch eine Realisierung über eine Hashtable in der ConLib vorhanden, die ein besonders effizientes Suchen im Behälter erlaubt. `tArray` hat nur eine Implementationsvariante: `tArrayAsVectorType`, eine Vektor-Implementierung[1]. Da die Listenvariante `tArrayListType` ein Array als technische Implementation verwenden muß, bietet es sich an, den Array-Behälter der ConLib wiederzuverwenden. Die Array-Implementation des Listen-Behälters verwendet daher ein Objekt der Klasse `tArrayAsVectorType`, so daß ConLib-Behälter in ConLib-Behältern wiederverwendet werden. Technische Implementation können aber auch schon definierte Methoden aus den abtrakten Klassen überschreiben, um diese durch effizientere Algorithmen zu ersetzen, falls das vorhandene Wissen um die technische Realisierung ein solches Verhalten ermöglicht.

2.3. Konstruktionsmechanismen

Bei der Konstruktion der ConLib kamen zahlreiche Entwurfsmuster, wie das Brücken-Entwurfsmuster [GHJ94] zum Einsatz. Von großer Bedeutung ist auch das Strategie-Muster zur Realisierung der Gleichheitsprüfung und zum Größenvergleich. Ein weiterer, wichtiger Konstruktionsmechanismus war die Benutzung von Templates, um statische Typsicherheit zu erzwingen.

Ein Merkmal der ConLib ist auch die konsequente Verwendung von Vorbedingungen. So wird bei fast jeder Methode der ConLib über Vorbedingungen so weit wie möglich die Korrektheit der Übergabeparametern und/oder die Zulässigkeit des entsprechenden Methodenaufrufes überprüft. Dieses geschieht über C++-Assertions, die Fehler mit sofortigen Programmabbruch quittieren. Die Assertions lassen sich für eine endgültige Version ausschalten.

2.3.1 Statische Typsicherheit durch Templates

Behälter müssen polymorph sein, d.h. jeder beliebige Datentyp muß in einem Behälter gespeichert werden können. Es sind allerdings auch Sammlungen von gleichartigen Elementen, so daß in einem Behälter nur Elemente (Objekte) eines Typs passen. Sie sind somit auch durch den Elementtyp typisiert. Um Behälter korrekt zu typisieren, sind die ConLib-Behälter als C++-Templates konstruiert, die einen parametrischen Polymorphismus zulassen [Stroustrup91]: Bei der statischen Typisierung eines Behälters muß ein Typparameter für die Elemente angegeben werden, der einen Platzhaltertypen der Elemente des Behälters in der Klassendefinition ersetzt. Durch diese statische Typisierung kann zur Übersetzungszeit der Typ der an den Behälter übergebenen Elemente geprüft werden, gleiches gilt für die vom Behälter zurückgegebenen Elemente. Hierbei muß noch beachtet werden, daß der Vererbungspolymorphismus erhalten bleibt. Ist beispielsweise eine Klasse Person die Oberklasse einer Klasse Student, so können in einem Behälter von Personen Objekte des Typs Student eingefügt werden, da sie auch vom Typ Person sind. Behälter, die mit zwei unterschiedlichen Behälter-

[1] Ein Array, dessen Größe veränderbar ist. Wird über ein C++-Array realisiert, das über Speicherallokation und –freigabe seine Größe verändert.

elementstypen parametrisiert sind, stehen untereinander nicht in Beziehung. Es sind Objekte unterschiedlichen Typs, da es in C++ keine kovariante (und ebenso kontravariante) Vererbungsbeziehung gibt. Somit könnte ein Behälter, der mit Studenten gefüllt ist, nicht auf einen Bezeichner zugewiesen werden, der mit einem Behälter typisiert ist, dessen Elemente als Person parametrisiert sind.

Der Template-Mechanismus hat bei den C++-Compilern aber noch ein Problem: Für jeden, mit einem neuen Typen benutzen Behälter, wird eine Kopie des Codes angelegt, in der der generische Typ einfach durch den konkreten Typ textuell ersetzt und jeder so neu erschaffene Behälter auch übersetzt wird[2]:

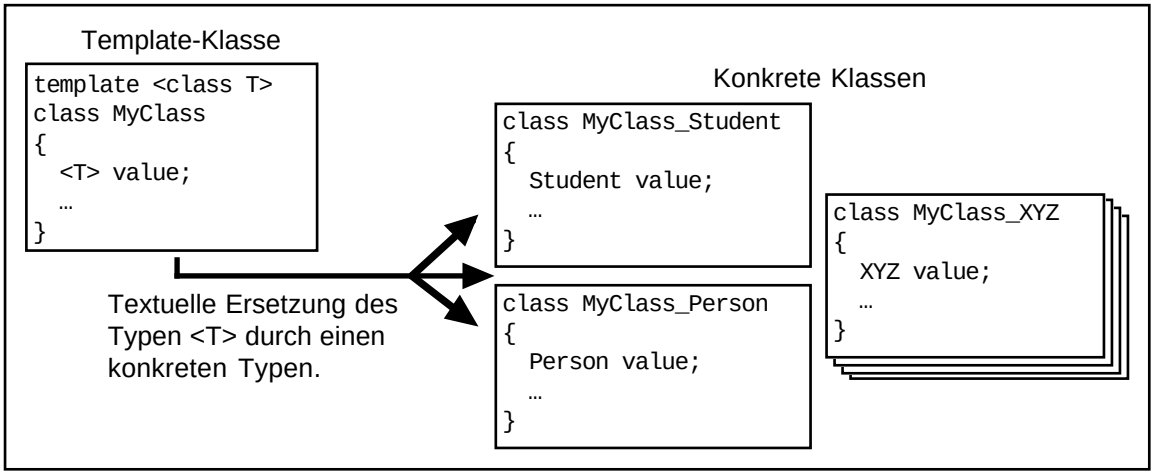

Abbildung 2.3 - Textuelle Ersetzung des Template-Parameters

Um das daraus resultierende Anwachsen von Code möglichst klein zu halten, folgen die Template-Klassen in der ConLib dem Brücken-Muster [GHJ94]: Die Template-Klassen erzeugen jeweils ein Objekt einer korrespondierenden, generischen Basisklasse und reichen die Methoden an dieses Objekt einfach durch. Der Code der Basisklassen ist nur einmal vorhanden, da sie nicht wie die Template-Klassen typisiert sind, sondern die Behälterelemente als untypisierte void-Pointer behandeln. Durch die Template-Kapsel ist die Typsicherheit allerdings gewährleistet. Die Template-Klassen enthalten daher nur sehr wenig Logik, die kopiert werden muß, während die Basisklassen die komplette Behälterlogik implementieren.

Die ConLib macht für Behälterelemente eine Einschränkung: Es muß sich bei den Elementen um Referenzen (Pointer) handeln. Würden auch Werte zugelassen werden, so gäbe es erhebliche technische Schwierigkeiten bei der Realisierung, da im Behälter Kopien der Werte lagern müssen. Die Realisierung eines Kopiermechanismuses für Behälterelementsklassen ist in einer Sprache ohne Garbage-Collector wie C++ kompliziert, da entschieden werden müßte, ob die Objekte flach (shallow-copy) oder mit allen Referenzen (deep-copy) kopiert werden sollten und es ohne Garbage-Collection schwer fällt, zu entscheiden, wer nicht mehr gebrauchte Objekte endgültig aus dem Speicher löscht. Somit ist die gesamte Problematik der Speicherverwaltung der Objekte im Behälter auf den Benutzer des Behälters abgeschoben. Er muß sich um das eigentliche Entfernen der Objekte im Speicher kümmern, da nur er entscheiden kann, ob es noch Referenzen auf das Objekt gibt oder nicht.

[2] Auch heterogene Translation genannt, siehe auch Abschnitt 3.1.1 zur Pizza-Spracherweiterung für Java.

2.3.2 Entwurfsmuster in der ConLib

Das Brücken-Muster wird neben der Template-Konstruktion in der ConLib auch noch zur weiteren Code-Minimierung durch Wiederverwendung von ConLib-Behältern in ConLib-Behältern benutzt. So verwendet z.B. der Stack für seine Array- bzw. Linked-List-Implementierungen wieder einen ConLib-Behälter, nämlich die Liste als Array- oder Linked-List-Implementierung (Abb. 2.4). Dabei werden Methoden-Aufrufe, die in der Oberklasse `tContainer` definiert sind, einfach weitergereicht, da die unterliegende Implementation ebenfalls ein ConLib-Behälter ist und diese Funktionalität schon anbietet. Die Benutzung geschieht über eine abtrakte Behälter-Schnittstelle.

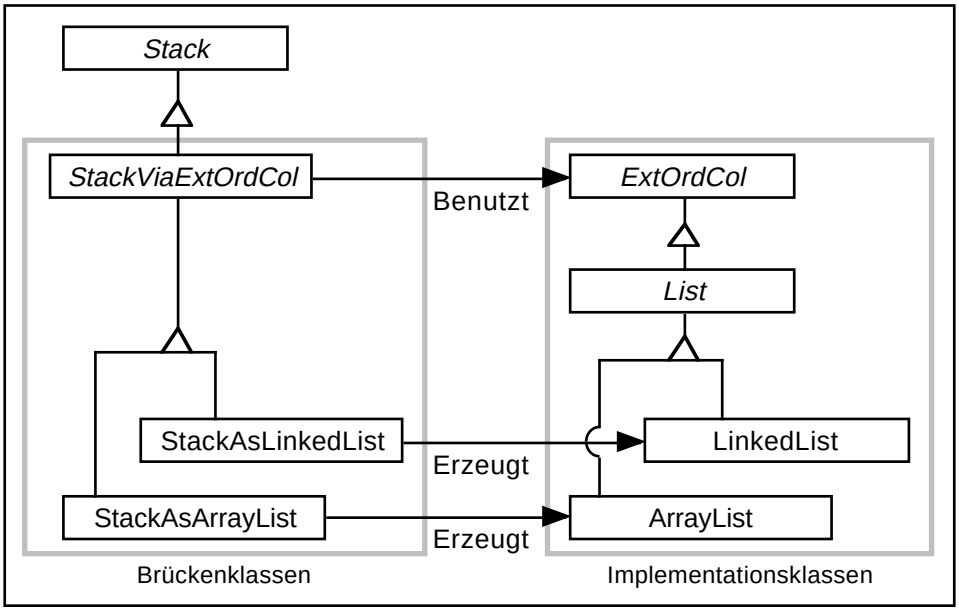

Abbildung 2.4 - Verwendung abstrakter Brückenklassen

Ein weiteres Entwurfsmuster ist u.a. für den Gleichheitstest verwendet worden: das der Strategieklasse [GHJ94] nach dem White-Box-Verfahren. Es gibt eine Oberklasse `tEqualTester`, die zwei Methoden zum Vergleichen von Objekten anbietet: Wertvergleich und Identitätsvergleich. In Unterklassen wird dann eine Strategie für diesen Vergleich weiter verfeinert, indem definierte Strategien aus der Oberklasse überschrieben werden. So muß es z.B. für Behälter mit Wert/Schlüssel-Paaren eine besondere Strategie für den Wertvergleich geben, da die Elemente im Behälter aus Assoziationsobjekten mit Wert und Schlüssel als Inhalt bestehen. Die für Assoziationselemente vorgesehene Vergleichsstrategie vergleicht natürlich nicht die Assoziationsobjekte selbst, sondern deren Referenzen auf Wert und Schlüssel. Für automatisch sortierte Listen, siehe `tIntOrdCol`, muß es eine Strategie für den Größenvergleich der Elemente untereinander geben, die beispielsweise auf- oder absteigendes Sortieren realisieren. Für Hash-Operationen gilt das gleiche.

2.4 Das Cursor-Konzept

Ein wesentliches, weiteres Konzept der ConLib ist das des Cursors. So sollen die ConLib-Behälter nicht nur passive Sammlungen von Elementen sein, sondern auch Aktionen mit diesen Elementen ermöglichen, sie sollen ein aktives Verhalten besitzen [Meyer97]. Ein in Behälterbibliotheken oft verwendetes Konzept ist das des Iterators, mit dem man alle Elemente des Behälters der Reihe nach abfragen und mit diesen Aktionen ausführen kann. Die ConLib erweitert dieses Konzept und führt

einen sogenannten ‚Cursor' ein. Cursor referenzieren Behälterelemente, sofern sie auf eines gesetzt sind und können sich frei über den Behälter bewegen, also z.B. über alle Elemente eines Behälters iterieren.

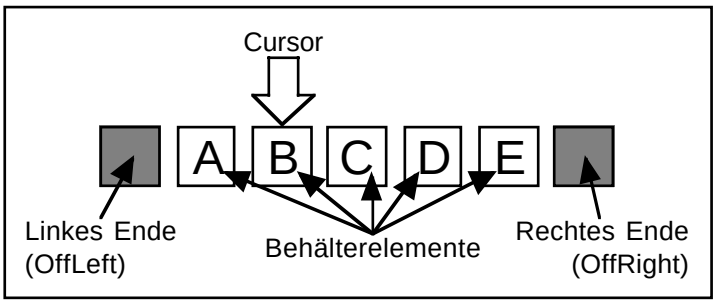

Abbildung 2.5 - Cursor und Behälter

So besteht jeder Behälter aus einer linearen Anordnung seiner Elemente, da diese in irgendeiner technischen Ordnung stehen müssen (Abbildung 2.5). Allerdings kann nur bei sortierten oder indizierten Behältern ein Zusammenhang zwischen technischer und fachlicher Ordnung hergestellt werden. Die ConLib-Cursor referenzieren nun ein Element aus diesem Behälter, anderes gesagt ‚stehen' sie auf einem Element (Cursorelement). Cursor können sich aber auch am linken oder rechten Ende eines Behälters befinden, wobei sie dann in diesem Zustand natürlich kein Element referenzieren. An einem Cursor kann, sofern er auf einem Element steht, dieses erfragt werden. Ferner kann sich ein Cursor immer um ein Element vor- und zurückbewegen oder, zum Zwecke des Zurücksetzens, sich direkt an das linke oder rechte Ende setzen.

In der ConLib können Cursorobjekte durch Instanziierung der Klasse `tCursor` erzeugt werden, wobei auch hier über einen Template-Mechanismus der Typ der referenzieren Cursorelemente statisch festgelegt wird. Ein solcher Cursor ist zwar schon typisiert, muß aber noch durch eine `login()`-Methode mit dem passenden Behälter verbunden werden.[3] Der Cursor kann nun über die Methoden `Forth()`, `Back()`, `GoOffLeft()` und `GoOffRight()` über den Behälter laufen. Über die Methode `Current()` kann ein Cursor das Element erfragen, auf dem er gerade steht, sofern der Cursor sich nicht am linken oder rechten Ende befindet (Vorbedingung!).

[3] Sind Cursor- und Behältertyp nicht identisch, kommt es zu einem statischen Typfehler.

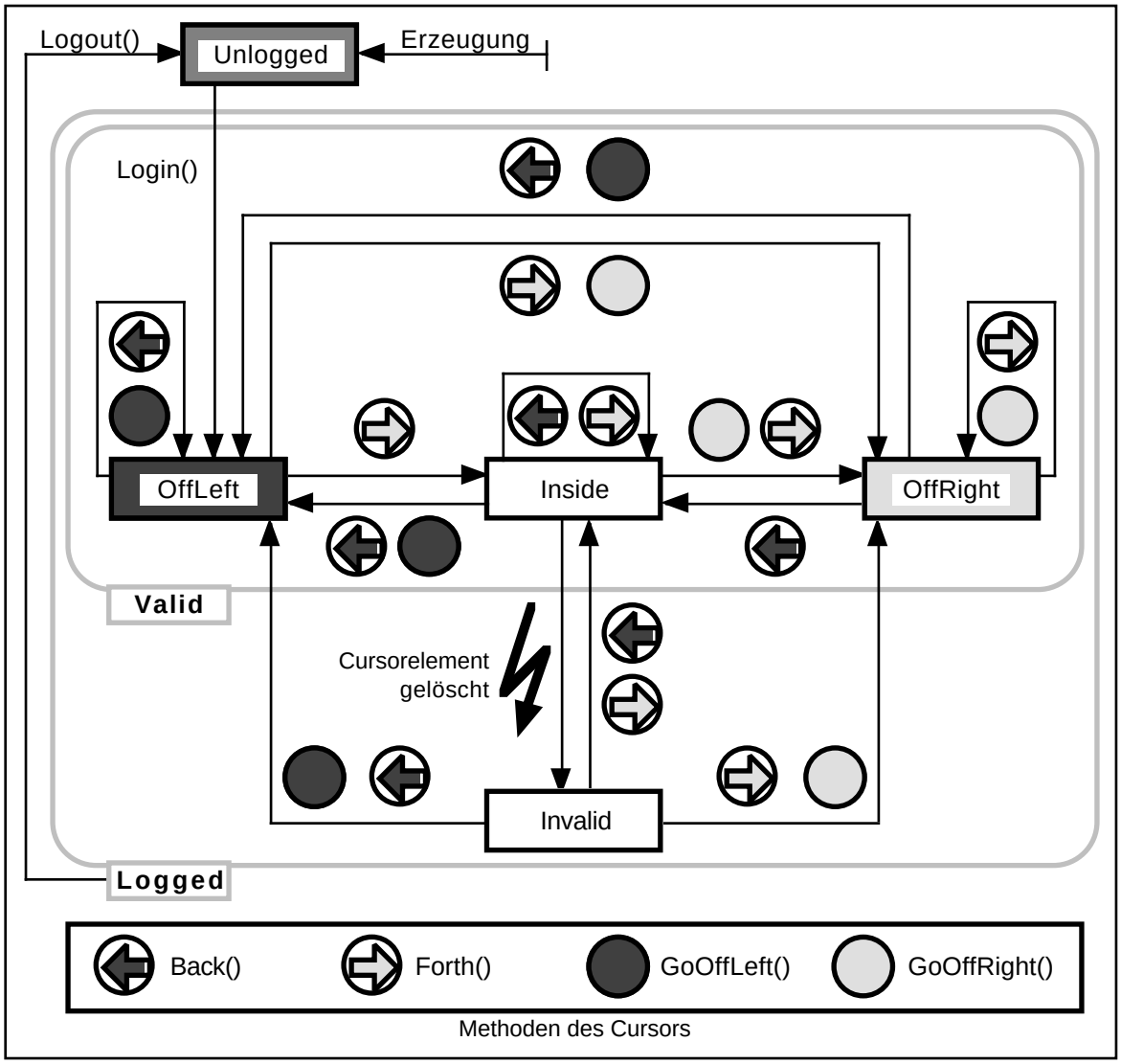

Abbildung 2.6 - Mögliche Zustandübergänge des Cursors

Abbildung 2.6 gibt einen Überblick auf alle Zustände und möglichen Übergänge des Cursors. Die Zustände Unlogged-Logged, Valid-Invalid, Inside-OffLeft-OffRight sind am Cursor über entsprechende Methoden abzufragen.

Cursor in der ConLib besitzen ein ‚robustes' Verhalten. D.h., sie sind gegen jegliche Mutation im Behälter geschützt (Hinzufügen und Löschen von anderen Elementen, neue Sortierung von Elementen, beispielsweise bei Reorganisation eines Behälters mit Hashtable) und behalten immer ihre Position auf dem aktuellen Element oder am linken bzw. rechten Ende bei, egal was mit dem Behälter passiert. Wird das aktuelle Cursorelement gelöscht, geht der Cursor in den Zustand ‚Invalid' über und sein Element ist über den Cursor nicht mehr zu referenzieren. Eine Vorbedingung der `Current()`-Methode des Cursors ist die, daß er sich im ‚validen' Zustand befindet, welche bei einem Zugriff auf einen invaliden Cursor verletzt und zum Programmabbruch führen würde.

Durch Neupositionierung gelangt der Cursor aber wieder auf den Behälter, wobei er mittels `Forth()` bzw. `Back()` auf die Elemente rechts bzw. links des ehemaligen Cursorelementes kommt. Der Benutzer eines Cursors kann sich also sicher sein, immer einen wohldefinierten Zustand vorzufinden und, falls ihm sein Cursorelement anderweitig ‚unter der Hand weggezogen wurde', er zumindest mit einer ordentlichen Fehlermeldung rechnen kann, falls er eine solche Möglichkeit nicht

bedacht hat. Durch Neupositionierung gelangt der Cursor auf jeden Fall wieder in einen validen, Zustand.

Das Cursor-Konzept unterscheidet sich von dem Iterator-Konzept anderen Bibliotheken hauptsächlich durch seine Langlebigkeit. Iteratoren dienen dazu, den Behälter einmal zu durchlaufen, während der ConLib-Cursor durch seine freie Positionierbarkeit und Robustheit auch für einen langfristigen Gebrauch nützlich ist.

Zwischen Cursor und Behälter findet teilweise eine direkte Kommunikation statt. So wird das Vor- und Zurückgehen auf dem Behälter von diesem selbst implementiert, da nur dieser das Wissen über die entsprechende Funktionalität hat. Die aktuelle Position und der damit verbundene Zustand des Cursors wird in einem eigenen Positionsobjekt gehalten, dessen Klasse eine Strategieklasse ist und für die es je nach Behälterimplementationen verschiedene Strategien gibt. Für indizierte Behälter kann die Position über einen numerischen Wert vermerkt werden, während bei verketteten Listen eine Referenz auf das passende Kettenglied gehalten werden muß. Welche Strategie passend ist, muß wiederrum der Behälter entscheiden, der auch eine entsprechende Fabrik-Methode besitzt und je nach Implementationsart ein passendes Strategieobjekt erzeugt. Die für alle Strategien allgemein geltenden Zustände (OffLeft-Inside-OffRight, Invalid-Valid) werden vom Cursor und Behälter am Positionsobjekt manipuliert und erfragt. Die Logik für diese Operation kann in die allgemeine Strategieklasse verlagert werden.

III. Von C++ zu Java: Gemeinsamkeiten und Unterschiede

Java ist genau wie C++ eine objektorientierte Sprache, die allerdings eine Neuentwicklung ist und nicht wie C++ als eine Spracherweiterung auf einer alten Sprache, nämlich C, aufbaut. Java ist aber auch eine Sprache, die sich in einer Entwicklung befindet und die in zukünftigen Versionen sicher noch einige Änderungen erfahren wird. So benutzt beispielsweise die *j*ConLib die erweiterte Funktionalität von Java 1.1 in bezug auf Meta-Informationen über Klassen. Diese war in der Version 1.0 noch nicht vorhanden. Interessant für die Portierung ist auch der Anspruch von Java, sich am C++-Syntax zu orientieren und daher einen ‚Umstieg' leicht zu machen. Auf diesen Gesichspunkt gehe ich am Ende dieses Kapitels nochmals ein.

In dem dritten Kapitel dieser Studienarbeit stelle ich unter dem Gesichtspunkt der Portierung der ConLib nach Java dar, was die beiden Sprachen unterscheidet. Ich gehe daher nicht auf alle Unterschiede oder gar ganz auf die Sprache Java ein, der Leser sei hier auf geeignete Literatur verwiesen [Flanagan97]. Auf die Konstruktion der *j*ConLib werde ich in Kapitel 4 genau eingehen. Zuerst erläutere ich die Probleme für die statische Typsicherheit, die durch den fehlenden Template-Mechanismus in Java entstehen. Eine Lösungsmöglichkeit wäre die sogenannte Spracherweiterung Pizza, deren Konzepte ich kurz erläutere, die aber aus Kompatibilitätsgründen nicht verwendet wurde.

Eine andere Lösungsmöglichkeit zur Realisierung von Typsicherheit ergibt sich aus der Kombination der Java-Spracheigenschaften, die ich in den weiteren Kapiteln erläutere: der Oberklasse `Object` in Java, den Java-Typecasts und der Möglichkeit in Java, Meta-Informationen über Klassen und Objekte zu erhalten. Von Bedeutung ist auch die fehlende Zusicherungsmethodik in Java, die aber durch eine eigene Konstruktion im JWAM-Framework größtenteils ersetzt wird [JWAM98]. Ein weiterer, wesentlicher Unterschied ist die in Java vorhandene Garbage-Collection, die für die Cursor-Konstruktion eine neue Möglichkeit eröffnet. Da Java über keine ‚klassische' Mehrfachvererbung verfügt, sondern eine Einfachvererbung mit sogenannten ‚Interfaces' bereitstellt, führt dieses auch zu einer kleinen Neuerung in der *j*ConLib. Erwähnenswert ist auch die eingebaute Serialisierbarkeit von Objekten und die in der Sprache Java ebenfalls eingebaute Möglichkeit, Nebenläufigkeit durch Threads zu realisieren (Stichwort: Synchronisation gemeinsamer Ressourcen).

3.1 C++- Templates für Polymorphie

Die ConLib macht reichlich Gebrauch von Templates zur statischen Typisierung der Behälter. Durch diese Realisierung eines parametrischen Polymorphismus kann schon zur Übersetzungszeit die Typrichtigkeit des Behälters und seiner Elemente überprüft werden. Auch die Verwendung von Templates bei der Cursor-Klasse erzeugt statische Typsicherheit.

In dem derzeitigen Java-Standard gibt es keinerlei Möglichkeit, Templates von Klassen zu kreieren. Das wirft für die Konstruktion von Behältern im Hinblick auf die Typsicherheit natürlich neue Probleme auf. Ich werde aber zeigen, daß durch die Kombination mehrerer Sprachfeatures in Java zumindest eine dynamische Typsicherheit hergestellt werden kann.

3.1.1 Pizza - Eine Java-Erweiterung

Eine Möglichkeit, Java den benötigten parametrischen Polymorphismus beizubringen, wäre die Benutzung einer Spracherweiterung von Java namens ‚Pizza' [OW97]. Pizza bietet neben parametrischen Polymorphismus noch Funktionen höherer Ordnung und algebraische Datentypen an. Pizza erweitert Java dabei um die entsprechenden Features und übersetzt den Pizza-Quellcode in Java-Quellcode. Dieser kann dann problemlos in Maschinencode der Java-Virtual-Machine (JVM) weiter übersetzt werden.

Interessant sind die Übersetzungsstrategien von Pizza im Hinblick auf die Ersetzung des generischen Typen durch den konkreten. Pizza bietet zwei Übersetzungsstartegien nach Java an: eine homogene und eine heterogene. Die heterogene Übersetzung nach Java erzeugt wie bei den gängigen C++-Compilern für jeden neuen Template-Typen eine schlichte textuelle Code-Ersetzung des generischen Typen durch den konkreten Typen in der Klassendefinition (Siehe Abb. 2.3). Ebenso wie in C++ führt dieses zu einem starken Anwachsen des Codes durch die neuen Klassen, aber auch zu schnellen Code. Die homogene Übersetzung erzeugt nur eine generische Template-Klasse, deren polymorpher Parameter durch ‚`Object`' ersetzt wird, die Oberklasse aller Klassen in Java. So können an ein Objekt einer solchen Klasse alle möglichen Java-Objekte über- und zurückgegeben werden, wobei bei der Rückgabe durch Typecasts eine Typumsetzung passieren muß und im übersetzen Java-Code auch eingefügt wird. Der statische Typ wird während der Übersetzung nach Java überprüft. Diese Variante erinnert an die explizite Konstruktion der ConLib, Template-Klassen schlank zu halten und die Logik in generische Klassen zu verlegen, die über void-Pointer Behälterelemente spezifizieren. Eine solche Methodik wird von Pizza automatisch gewährleistet, ist aber durch die Typecasts, die in Java den Typ überprüfen und daher Zeit kosten, im Vergleich zur heterogenen Strategie langsamer. Diese Typersetzung funktioniert allerdings nicht, wenn als Typparameter ein Basistyp angegeben wird und kein Objekttyp.

Die Spracherweiterung von Pizza wäre für die Konstruktion der *j*ConLib ideal gewesen, jedoch ist Pizza nur eine *Spracherweiterung* von Java und nicht ein Bestandteil von Java selbst. Somit müßte die Benutzung einer Pizza-Behälterbibliothek (*pj*ConLib) auch über Pizza stattfinden, da nur dann der Polymorphismus sinnvoll genutzt werden kann. Für die Integration in das Framework würde das bedeuten, daß auch dieses mittels Pizza übersetzt werden müßte. Man würde sich in eine weitere Abhängigkeiten begeben, was in Hinblick auf ein portables Framework und eine portable Behälterbibliothek unerwünscht ist. So müßten Entwickler neben Java noch über eine Pizza-Erweiterung verfügen und diese auch in ihre Entwicklungsumgebungen einbinden können. In Zukunft könnte parametrischer Polymorphismus allerdings in den Java-Sprachstandard aufgenommen werden, was die Entwickler der Collection-API des JDK 1.2 nicht unbedingt ausschließen [CollAPI-FAQ97].

3.2 Java - Oberklasse Object

Im Gegensatz zu C++ gibt es in Java eine allgemeine Oberklasse `Object`, die jede Klasse ohne Vererbungsbeziehung implizit als Oberklasse besitzt. In `Object` sind Standardmethoden vordefiniert, von denen einige für die *j*ConLib nützlich sind:

- **`boolean equals(Object obj)`**
  Diese Methode sollte in den eigenen Klassen überschrieben werden und den Wert des übergebenen Objektes auf Gleichheit überprüfen. In `Object` wird in dieser Methode ein Vergleich der Referenzen durchgeführt.

- `int hashCode()`
  Gibt einen rein technisch ermittelten Hashcode wieder. Dieser kann sogar zwischen zwei Programmaufrufen variieren und sollte in Unterklassen durch einen geeigneteren Algorithmus ersetzt werden.

Objekte gehorchen in Java nur der Referenzsemantik. Allerdings gibt es Basistypen, wie beispielsweise `int`, `float`, `boolean`, etc., die eine Wertsemantik besitzen. Für diese Basistypen gibt es aber korrespondierende Klassen, die die Basistypen kapseln.

Für die Konstruktion der *j*ConLib kann man die Behälterelementen über `Object` statisch typisieren, um die Behälter generisch wie bei der homogenen Übersetzung in Pizza für alle möglichen Elementezugänglich zu machen [4]. Das mag auf den ersten Blick wie das Benutzen von void-Pointern in C++ erscheinen, die den Aspekt der Typsicherheit ausblenden, allerdings werden ich noch ein paar Mechanismen erläutern, die eine gewisse Typisierung des Behälters doch ermöglichen.

3.3 Java - Typecasts

Durch Klammerung eines Typen kann man ganz wie in C++ einen Typecast an einem Wert oder einer Objektreferenz durchführen. Im Gegensatz zu C++ findet bei einem Typecast an einer Objektreferenz zur Laufzeit eine Typprüfung statt. Es kann nur ein Typecast in den eigentlichen Typ des Objektes oder in einen Super-Typen der Objektes stattfinden (Vererbungspolymorphie). Somit ist ein Typecast nach `Object` immer erlaubt, da dieses der oberste Typ aller Objekte ist. Wird eine Objektreferenz beispielsweise einem Bezeichner eines Super-Typen des Objektes zugewiesen, muß nicht explizit ein Typecast stattfinden. Erfüllt das Objekt des Typecasts zur Laufzeit nicht die korrekten Typanforderungen, wird dieses von Java mit einer Exception (Runtime-Exception, siehe Abschnitt 3.5) quittiert.

Bis hier sind wir an dem Punkt, daß die Behälterelemente völlig uneingeschränkt statisch als `Object` typisiert sind, sowohl Übergabeparameter als auch Rückgabeparameter. Werden nun Behälterelemente zurückgegeben und diese Bezeichnern übergeben, die einen konkreteren Typ als `Object` haben, muß ein Typecast durchgeführt werden, der automatisch zur Laufzeit den Rückgabetypen überprüft.

3.4 Java - Meta-Informationen

Java bietet eine ganze Bandbreite von Meta-Informationen über Objekte und Klassen. So existiert in Java zu jeder Klasse ein Klassenobjekt (Instanzen der Klasse `Class`), welches über die instanziierten Objekte oder über den Klassennamen an der Klasse `Class` zu bekommen ist. An den Klassenobjekten lassen sich Meta-Informationen über Methoden, Konstruktoren und Vererbungsbeziehungen ermitteln und damit dynamisch Objekte von Klassen erzeugen oder dynamisch Methoden aufrufen. Das Klassenobjekt zu einer Klasse läßt sich aber auch einfacherweise durch das Anfügen des Textes „`.class`“ an einer Klasse ermitteln. An Klassenobjekten läßt sich überprüfen, ob ein bestimmtes Objekte von dieser Klasse instanziiert wurde.

Über diese Meta-Informationen können in der *j*ConLib Informationen über die Behälterelemente gesammelt werden, indem man einem Behälter bei seiner Erzeugung das Klassenobjekt der zukünftigen Behälterelemente übergibt. Damit ist der Weg frei, die Elemente eines Behälters zu typisieren und übergebene Elemente auf den korrekten Typ zu prüfen.

[4] Elemente im Sinne von Objekten. Werten können über korrespondierenden Klassen gekapselt werden.

3.5 C++ - Zusicherungen

In C++ gibt es über Assertions die Möglichkeit, das Vertragsmodell nach Meyer [Meyer97] teilweise nachzubilden. Assertions definieren allgemeine Zusicherungen, die an ihren Punkt im Programm erfüllt sein müssen. Falls die Bedingung nicht erfüllt wird, wird das laufende Programm mit einer passenden, erläuternden Fehlermeldung abgebrochen. Bei der Übersetzung des Programmes kann entschieden werden, ob die Assertions aktiviert werden oder nicht, so daß sie aus ‚fertigen' Programmen vollständig entfernt werden können. Hiermit lassen sich also Vor- und Nachbedingungen realisieren.

In der Sprache Java ist zwar ein Exception-Modell fest verankert, allerdings gibt es weder eine Zusicherungsmethodik noch eine entsprechende Bibliothekserweiterung für jene. Das JWAM-Framework wurde deshalb um ein solches Zusicherungsmodell erweitert, von dem auch die *j*ConLib regen Gebrauch macht. Dabei handelt es sich um Aufrufe von Klassenmethoden an einer Klasse namens `Contract`, die bei Verletzung der Bedingung zu einem kommentierten Programmabbruch führen. Ein Vertragsbruch wird mit einer ‚Runtime-Exception' quittiert. Runtime-Exceptions unterscheiden sich von einfachen Exceptions dahingehend, daß sie jederzeit erzeugt werden können und nicht wie bei einfachen Exceptions explizit in Methoden die Möglichkeit einer solchen Exception angekündigt werden muß. Beide Exceptions können aber durch eine ‚`try/catch`'-Klausel abgefangen und bearbeitet werden, wobei einfache Exceptions entweder durch die `try/catch`-Klausel abgefangen oder weitergereicht werden müssen. Im letzeren Falle muß eine benutzende Methode aber wiederum die Möglichkeit einer Exception in der Methodendefinition angeben.

Nebenbei findet bei jeder Vertragsüberprüfung ein Test einer Invariantenbedingung statt, die in der aufgerufenen Klasse einfach als Methode mit den Namen `invariant()` definiert sein kann. In der fertigen Version der *j*ConLib gibt es allerdings keine solche Invarianten, da diese nur zum Überprüfen der inneren Konsistenz der Behälter benutzt wurden.

3.6 Java - Garbage-Collection

In Java gibt es eine Garbage-Collection, d.h. nicht mehr referenzierte Objekte werden automatisch aus dem Speicher entfernt, eine explizite Löschung muß nicht mehr vorgenommen werden. Somit entfallen alle Probleme, wer, wann und wo für das Löschen von Objekten zuständig ist, die typischen Probleme ‚Lost Pointer' oder ‚Dangling Pointer' wie in C++ treten nicht mehr auf. Für den Benutzer der *j*ConLib bedeutet das, daß er sich nicht explizit um das eigentliche Löschen von Behälterobjekten kümmern muß. Wird ein Objekt aus dem Behälter gelöscht und nicht mehr anderweitig referenziert, wird es früher oder später automatisch aus dem Speicher entfernt.

Die Garbage-Collection eröffnet aber noch eine neue Möglichkeit: die eines stabilen Cursors. Bei robusten Cursorn kommt es zu einem Fehler, wenn das Element, auf dem er stand, gelöscht wird und anschließend versucht wird, dieses Element über den Cursor zu ermitteln. Das Element befindet sich nicht mehr im Behälter, jedoch ist ein stabiler Cursor immer noch auf dieses Element eingestellt und es kann solange referenziert werden, bis der Cursor durch Neupositionierung wieder auf dem Behälter aufsetzt. In der ConLib ist ein stabiler Cursor durch die fehlende Garbage-Collection nur schwer zu realisieren. Das Konzept des stabilen Cursors wird noch im nächsten Kapitel unter 4.4 ausführlich erörtert.

3.7 Java - Vererbung und Interfaces

In Java gibt es keine Mehrfachvererbung, dafür aber das Konzept von Interfaces: Klassen können beliebig viele abstrakte Schnittstellen implementieren, deren Umgang durch Methoden in Interfaces abstrakt definiert wird. Mit Interfaces lassen sich also Eigenschaften definieren, die von einer Klasse erfüllt werden müssen. Die abstrakten Methoden müssen von der implementierenden Klasse überschrieben werden, ansonsten muß die Klasse selbst als abstrakt deklariert werden. Das Interface wird von allen Spezialisierungen der Klasse erfüllt. Die Interfaces schlagen sich auch im Typ der Objekte nieder, denn es läßt sich statisch und auch dynamisch nachprüfen, ob Objekte ein Interface implementieren oder nicht, also ob sie auch von deren Typ sind. So können Bezeichner den Typ eines Interfaces haben, so daß sie nur Objekte von Klassen zugewiesen bekommen können, die das Interface implementieren.

Auf den ersten Blick scheint die Einfachvererbung in Java für die *j*ConLib kein Problem: Die Behälterhierarchie ist einfach aufgebaut, es gibt bewußt keine Mehrfachvererbung. Es existieren aber Behältervarianten, die eine implizierte Sortierung besitzen, die durch die Elemente selbst definiert wird. In der Klasse der Behälterelemente muß dafür eine Möglichkeit zum Größenvergleich gegeben sein, die in Java nicht vorhanden ist, da Vergleichsoperatoren wie ‚<' oder ‚>' in Java für Objekte nicht erlaubt sind. Ein Interface kann nun über eine einheitliche Methode eine Vergleichsoperation festschreiben, die von Klassen erfüllt werden kann, die vergleichbare Objekte erzeugen sollen.[5] Die Behälter der *j*ConLib sind für den Benutzer über Interfaces definiert, die die Benutzung des Behälters aus rein fachlicher Sicht beschreiben.

3.8 Java - Serialisierbarkeit

Java bietet eine sehr einfache Methodik, um Objekte zu serialisieren. Klassen, deren Objekte serialisiert werden sollen, müssen nur das Interface `Serializable` erfüllen. Die Klasse selbst muß keine Methoden implementieren, das Interface `Serializable` dient nur zum Markieren der serialisierbaren Objekte (durch die Klassen). Serialisierbare Objekte können dann über einen Serializer-Stream, der durch Verknüpfung seine Daten in beliebige I/O-Kanäle schreiben kann, weggeschrieben werden. Das Lesen funktioniert in gleicher Weise.

Die *j*ConLib ist durch Markierung aller nötigen Oberklassen, z.B. `conContainer`, vollständig serialisierbar, wenn auch die Behälterelemente serialisierbar sind. Interessant ist diese einfache Technik im Vergleich zur aufwenigen Konstruktion, die man in C++ betreiben muß, wenn man die ConLib um Serialisierbarkeit erweitern will. Ein großes Problem bildet dabei die Template-Konstruktion, nähere Information in [Zühlke98].

3.9 Java - Synchronisation

Java ist eine Multi-Threading-Sprache, die schon in der Sprachdefinition das Problem der Nebenläufigkeit berücksichtigt. So können Threads gemeinsame Objekte nebenläufig verändern, was zu Problemen führt, wenn Operationen an Objekten atomar sein sollen. Durchläuft ein Objekt beispielsweise eine Methode, in der der interne Zustand des Objektes mehrfach verändert wird und zwischenzeitlich nicht konsistent ist, so könnte ein weiterer, nebenläufiger Zugriff auf dieses Objekt fatale Folgen haben. Daher bietet die Sprache Java das Attribut ‚`synchronized`' an, um einen exklusiven Zugriff auf Ressourcen (Objekte) zu sichern. Wird es auf Methoden angewendet, so wird

[5] Eine Vergleichsmethodik wird im JDK 1.2 enthalten sein, ebenfalls über ein Interface

das aufgerufene Objekt exklusiv für den Aufrufer reserviert und es kann keine weitere Sperrung des Objektes durch einen anderen `synchronized`-Abschnitt stattfinden, der ebenfalls auf diese Objekt zugreift. Programmabschnitte lassen sich ebenfalls mit einer `synchronized`-Klammer versehen, innerhalb der ein oder mehrere anzugebende Objekte ebenfalls exklusiv gesperrt sind.

In der *j*ConLib mußten, um sie gegen Threads zu ‚sichern', nicht-atomare und den Objektzustand verändernde Methoden mit dem `synchronized`-Attribut ‚synchronisiert' werden. So sind alle Methoden synchronisiert, die das Objekt verändern, egal wie komplex sie sind. Methoden, die den Zustand des Objektes nur abfragen, sind ebenfalls synchronisiert, falls sie ihr Ergebnis aus mehreren lesenden Operationen zusammenstellen.

3.10 Allgemeines zu Java und C++

Die in Abschnitt 3.8 erläuterte, einfache Möglichkeit der Serialisierbarkeit zeigt schon auf, daß Java eine neuentwickelte Sprache ist, die über keine ‚Altlasten' verfügt und moderne Sprachkonzepte verwirklicht hat. Serialisierbarkeit von Objekten, Meta-Object-Protokolle, Garbage-Collection und eine, nicht wie in C++ an Pointern orientierte Referenzsemantik machen eine objektorientierte Konstruktion doch erheblich einfacher. Java wurde von Grund auf objektorientiert entworfen und damit konnten auch entsprechende Konzepte ihren Eingang in die Sprache finden. Allerdings vermisse ich für die Konstruktion der *j*ConLib weitere Konzepte wie eine Zusicherungsmethodik oder die Möglichkeit von Templates, die mehr oder minder nachkonstruiert werden müssen. Andererseits gibt es in Java die Option, Typen zur Laufzeit nachzuprüfen, was automatisch bei Typecasts funktioniert oder beispielsweise durch den Java-Ausdruck `instanceof`[6] überprüft werden kann. Ich möchte die Vermutung äußern, daß Java aus Sicht der Typisierung eine sehr dynamische Sprache ist, was sich auch in der Konstruktion der *j*ConLib niederschlägt.

Die Ähnlichkeit der Syntax von Java und C++ ist bewußt und soll einen einfachen Umstieg von C++ nach Java ermöglichen. Außerdem gehen die Java-Entwickler davon aus, daß C++ eine recht bekannte Sprache ist und der Syntax vielen Programmierern vertraut ist. Da Java eine ähnliche Syntax hat, soll die Sprache mit entsprechenden C++-Kenntnissen leicht erlernbar sein, was ich aus meiner persönlichen Erfahrung heraus bestätigen kann. Allerdings darf man im Vergleich zu C++ nicht vergessen, daß Java eine andere Sprache mit anderen Konzepten ist. Ein ‚ad hoc'-Umsatteln von C++ nach Java ist nicht möglich, schließlich müssen wie bei jeder neuen Sprache Erfahrungen mit deren Konzepten gemacht werden, auch wenn ein ähnlicher Syntax dort helfen mag. Bei der Umsetzung der *j*ConLib mußte ich mir Gedanken darüber machen, wie ich denn einen parametrischen Polymorphismus ohne die in Java fehlenden Templates ersetzen oder zumindest nachbilden könnte. Die Lösung des Problems durch eine dynamische Typprüfung stelle ich im folgenden Kapitel vor.

[6] ‚`object instanceof class`' prüft, ob das Objekt ‚`object`' von der Klasse ‚`class`' instanziiert wurde. Der Ausdruck wird zu einem Wahrheitswert ausgewertet.

IV. Realisierung der *j*ConLib

In diesem Kapitel gehe ich vor allem auf die wesentlichen Neuerungen und Unterschiede gegenüber der ConLib ein. Ein erster Punkt ist die Nachbildung von Typsicherheit, die in der *j*ConLib zur Laufzeit geprüft wird. Dazu muß bei der Erzeugung der Behälter ein Klassenobjekt der Klasse der Elementobjekte übergeben werden, an dem der korrekte Typ der Behälterelemente geprüft werden kann. Die Erzeugung der Behälter funktioniert in der *j*ConLib an einer Fabrikklasse, auf die ich im nächsten Unterkapitel eingehe. Die Fabrikmethoden prüfen dabei dynamisch, welche Behälterimplementationen vorhanden sind und wählen auch dynamisch eine möglichst passende Variante aus.

Der Cursor hat in der *j*ConLib eine wesentliche Neuerung erfahren: Er ist stabil geworden, das heißt, er kann sein Element, falls es aus dem Behälter entfernt wurde, noch referenzieren. Über Möglichkeiten und Probleme dieses Konzeptes referiere ich in einem eigenen Abschnitt, sowie über weitere Änderungen an dem Cursor. In der *j*ConLib müssen Objekte gewisse Anforderung erfüllen, z.B. müssen die Elemente von intern sortierten Klassen natürlich vergleichbar sein. Dieses wird über Interfaces realisiert, denen ich einen Abschnitt widme.

Abschließend erläutere ich noch die Integration der *j*ConLib in das Framework und spreche über den Vorgang der Portierung und deren Probleme.

4.1 Die *j*ConLib-Behälterhierarchie

Die Behälterhierarchie der *j*ConLib ist der der ConLib natürlich ähnlich (Vergleiche Abbildung 2.1). Die *j*ConLib beinhaltet nicht alle Behälterarten (aktueller Stand) und hat eine andere Benennung. Nach den für das JWAM-Framework entworfenen Java-Styleguides [JWAM98] fangen alle Behälterklassen mit der Silbe ‚con' an, um die Vererbungshierarchie auch im Namen kenntlich zu machen. Bei den Behälternamen kam es ebenfalls zu einigen kleinen Umbenennungen und Vereinheitlichungen:

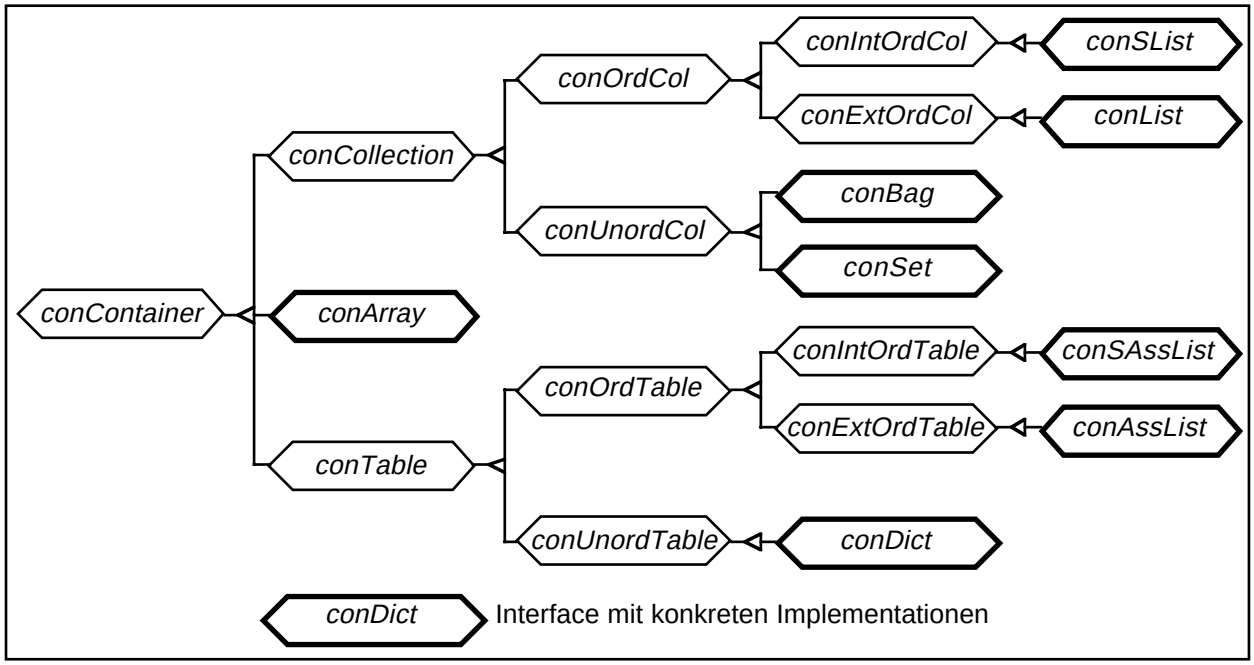

Abbildung 4.1 - Behälterhierarchie in der jConLib

Bei den Behältern handelt es sich um Interfaces, die einen fachlichen Umgang mit den Behältern definieren. Jedes Interface wird von genau einer abstrakten Klasse implementiert (z.B. `conAbsContainer` für `conContainer`), die, soweit es geht, allgemeine Methoden definieren. Von diesen abstrakten Klassen erben dann wieder die konkreten Implementationen. Es findet die gleiche Unterscheidung nach Behältern mit Wertvergleich und Behältern mit Referenzvergleich statt (siehe Kapitel 2.2). Allerdings macht sich hier schon der fehlende Template-Mechanismus bemerkbar: Die Aufgliederung der Implementationsklassen entspricht dem Vorbild der ConLib (siehe Abb. 2.2), jedoch sind in der ConLib die Oberklassen der Wert- und Referenzbehälter (z.B. `tArrayListType`) Templates mit der Vergleichsstrategie, neben dem Elementtypen, als weiteren Typparameter. Die *j*ConLib verfügt über keine solche Typisierung nach Vergleichsstrategien, es wird ein Objekt der entsprechenden Vergleichsstrategie als Konstruktorparameter übergeben. Das Strategie-Muster für den Wertvergleich findet sich aber genauso wieder wie in der ConLib.

Die Benennung der Methoden ist mit der der ConLib identisch, bis auf die Tatsache, daß in der *j*ConLib die Namen der Methoden mit einem Kleinbuchstaben anfangen, was den Java-Styleguides entspricht.

4.2 Typsicherheit

Die Elemente der *j*ConLib-Behälter sind, wie schon in Kapitel 3 erläutert, statisch mit dem allgemeinsten Typen `Object` typisiert. Um Typsicherheit zu erreichen, muß der Typ der Übergabe- als auch die Rückgabeobjekte eines Behälters zur Laufzeit überprüft werden. Da Rückgabeelemente von `Object` auf ihren konkreten Typ gecastet werden müssen, findet hier schon dynamisch eine Typprüfung statt. Übergabeelemente müssen jedoch auf ihren Typ hin überprüft werden, da ansonsten den Behälter ein beliebiges Element übergeben werden könnte.

4.2.1 Erzeugung der Behälter mit Klassenobjekt

Der Behälter muß wissen, wie seine Elementtypen aussehen. Dazu werden in der jConLib Klassenobjekte verwendet, die als Konstruktorparameter dem Behälter bekannt gemacht werden. Damit wird der Typ der Elemente einmal bei der Erzeugung festgelegt und kann für den ganzen Lebenszyklus des Behälters nicht mehr verändert werden. Soll beispielsweise eine Liste als Array-Implementation erzeugt werden, die Integer-Objekte beherbergen soll und deren Elemente über ihren Wert verglichen werden, gestaltet sich der Aufruf folgendermaßen:[7]

```
conList list;

list = new conListAsArray(Integer.class);
```

Man beachte, daß die Referenz von `list` den Typ `conList` besitzt. Der statische Typ des Bezeichner hat keine Typisierung in bezug auf die Behälterelemente. Bei der Erzeugung wird ein Klassenobjekt übergeben, welches in diesem Fall einfach durch Nennung der Klasse mit dem Zusatz ‚`.class`‘ ermittelt wird. Die Erzeugung weist eine gewisse syntaktische Ähnlichkeit mit der Erzeugung von Behältern der ConLib auf:

```
tList<int*>* list;

list = new tArrayList<int*>;
```

[7] Normalerweise sollten Behälter über eine Fabrik erzeugt werden (Kapitel 4.3). Der Behälter wird hier nur zur Verdeutlichung der Typsicherheit direkt erzeugt.

Hier wird ‚`int*`' als statischer Typparameter bekannt gegeben (‚übergeben'), in der *j*ConLib ist der Typ ein Konstruktorparameter. Der Typparameter wird bis in die allgemeinste Behälterklasse `conContainer` weitergereicht, wo der Wert zentral gespeichert wird und auch nicht mehr verändert werden kann.[8]

4.2.2. Dynamische Typsicherheit als Vorbedingung

Die allgemeinste Behälterklasse `conContainer` speichert nicht nur den Wert des Klassenobjektes, sondern bietet auch eine Methode zur Typprüfung an, die von jeder Behältermethode benutzt werden sollte, um den Typ ihres Übergabelementes zu überprüfen. Die Typprüfungsmethode sieht folgendermaßen aus:

```
protected final void testType(Object obj)
{
    if(Factory.checkType)
    {
        if(obj != null && !_elemClass.isInstance(obj))
        {
            throw new TypeError("Element of " + obj.getClass() +
                " not instance of " + _elemClass);
        }
    }
}
```

Die Methode ist `protected`, damit nur der Klasse selbst und Unterklassen zugänglich, und `final`, kann somit in Unterklassen nicht mehr überschrieben werden. `checkType` ist ein boolscher Wert der Klasse `Factory`, der zentralen Fabrikklasse der Behälterbibliothek. Er ist standardmäßig wahr, kann aber durch den Benutzer verändert werden, womit ein Ausschalten der Typprüfung bewirkt wird (Dazu mehr in Abschnitt 4.2.4). Wenn ein zu überprüfendes Objekt existiert (`obj != null`), wird über eine Methode (`isInstance`) des Klassenobjektes der Behälterelemente (`_elemClass`) an dem Objekt überprüft, ob es von diesem Klassenobjekt instanziiert wurde. Dabei werden natürlich auch eventuelle Subtyp-Beziehungen beachtet: Ist die Klasse Person die Oberklasse von Student, so kann ein Objekt von Student auch von einem Bezeichner des Typs Person referenziert werden.

Falls diese Typbedingung nicht erfüllt wird, wird ein ‚Java-Error' initiiert, der das Programm auf jeden Fall zum Abbruch bringt. Ein Typfehler ist ein Programmierfehler. Würde er statisch erkannt, hätte sich das Programm nicht kompilieren lassen. Ein Java-Error kann im Gegensatz zu einer Exception nicht abgefangen werden, das Programm kommt zum Abbruch.

Die Typüberprüfung muß von jeder Methode im Behälter aufgerufen werden, die ein Element als Übergabeparameter hat. Hier ein Beispiel eines Arrays, bei dem an einer Position ein Behälterelement ersetzt wird:

```
public Object replaceAtIndex(int i, Object obj)
{
    …                           // Definition lokaler Variablen
    testType(obj);
    …
}
```

[8] Bei allen Schlüssel/Wert-Behältern (Oberklasse `conTable`) wird der Typ der Werte an die Klasse `conContainer` übergeben. Der Typ der Schlüssel, der bei der Erzeugung dieser Behälter auch noch übergeben werden muß, wird in `conTable` vorgehalten.

Arrays lassen auch Lücken im Behälter zu, so daß als Element eine leere Referenz übergeben werden kann (`null`). Dieses wird bei der Typprüfung berücksichtigt. Diese dynamische Typprüfung hat den Character einer Vorbedingung, da sie für jede Methode explizit angegeben werden muß, zur Laufzeit geprüft wird und noch abgeschaltet werden kann.

4.2.3 Typecasts bei der Rückgabe

Die Behälter gewährleisten Typsicherheit für Übergabeparameter. Werden nun Elemente vom Behälter zurückgegeben, so sind diese wie die Übergabeparameter vom statischen Typ `Object`. Erfolg eine Zuweisung auf einen weiter spezialisierten Bezeichner, muß ein expliziter Typecast durchgeführt werden, sonst kommt es zu einem statischen Typfehler (Jede Spezialisierung von `Object` ist Subtyp von dieser, aber nicht umgekehrt). Bei der Liste aus Abschnitt 4.2.1 sähe das folgendermaßen aus:

```
Integer     value;
value = (Integer)list.atNo(6);
```

Java-Typecast vollziehen, wie in Kapitel 3 schon beschrieben, eine Typüberprüfung. Würde in dem vorigen Beispiel ein falscher Typ angegeben, käme es zu einem Laufzeitfehler (ClassCast-Exception).

4.2.4 Probleme der dynamischen Typisierung

Soviel Mühe man sich auch gibt: Die dynamische Typisierung kann Fehler nur zur Laufzeit erkennen. Um sicherzustellen, daß eine Programmteil Typ-sicher ist, muß dieser ausgeführt werden. Kommt es zu einem Typfehler, indem der *j*ConLib Werte eines falschen Typs übergeben werden, wird das Programm auf jeden Fall angehalten. Werden Elemente aus dem Behälter zurückgegeben, so müssen diese einen Typecast durchlaufen, der aber durch das Abfangen seiner Exception im Fehlerfall wieder ausgehebelt werden kann. Dieser Fall könnte unbewußt eintreten, wenn z.B. ein größerer Programmabschnitt, in dem durch andere Methoden Exceptions zu erwarten sind, durch eine `try`/`catch`-Klausel geklammert ist, die alle Exceptions abfängt. Auch die ClassCast-Exception würde in diesem Fall abgefangen werden und der Typfehler schlimmstenfalls nicht bemerkt.

Aber auch im Behälter kann die Typprüfung ‚aus Versehen' untergehen. So sollte jeder Übergabeparameter auf seinen korrekten Typ geprüft werden, der Programmierer des Behälters wird dazu aber nicht gezwungen. In der aktuellen *j*ConLib findet zwar überall eine Typprüfung statt, ausschließen kann man aber gerade in Hinblick auf Erweiterungen der Bibliothek solche Fehler nicht[9]. Die Typprüfung ist in der *j*ConLib eben nur eine zusätzliche, weitere Vorbedingung, die neben den ‚üblichen' Vorbedingungen in den Methoden steht und sich, um den Gedanken der Vorbedingung konsequent weiterzuführen, über einen statischen Wert an der Klasse `conContainer` wie die Vorbedingungen aus dem JWAM-Framework ausschalten läßt.

Da keine statische Typisierung der Behälter durch deren Elemente vorkommt, tritt ein weiteres Problem auf: Die Behälter können lediglich über ihre Behälterart statisch typisiert werden. So könnte beispielsweise eine Methode, die einen Behälter mit `Integer`-Werten erwartet, nur die Behälterart als Typ des Übergabeparameters angeben. Somit kann ein Behälter mit jeden beliebigen Elementstypen übergeben werden, eine statische Überprüfung findet nicht statt. Spätestens bei der Benutzung eines Behälters mit den falschen Elementen tritt aber der Fehler zu Tage, somit allerdings auch wieder erst zur Laufzeit.

[9] Anzumerken ist, daß durch die Wiederverwendung von Behälter der *j*ConLib die Gefahr von Programmfehlern in Hinblick auf fehlende Typüberprüfung minimiert wird.

4.3 Die Behälterfabrik

In Abschnitt 4.2.1 wurde ein Behälter durch Erzeugung eines Objektes einer bestimmten, ausgewählten Klasse erzeugt. Dabei muß sich der Benutzer darüber im klaren sein, was für eine Behälterimplementation er nehmen will, wie diese heißt und was für Vorteile ihm diese bringt. Die Behälter werden aber nicht als Objekte der konkreten Implementationen benutzt, sondern auf Bezeichner mit entsprechenden abstrakten Behältertypen zugewiesen. Den Benutzer interessiert die konkrete Implementation nur bei der Erzeugung des Behälters. In der *j*ConLib können die Behälter zwar durch direktes Instanziieren an der passenden Klasse erzeugt werden, es existiert aber noch eine Fabrik für Behälter [GHJ94] in Form einer Fabrikklasse, die die Erzeugung der Behälter übernimmt. Eine Erzeugung einer Liste an der Fabrikklasse gestaltet sich folgendermaßen (Vergleiche das Beispiel in 4.2.1):

```
conList list;

list = Factory.newList(Integer.class, Factory.array);
```

Über weitere Klassenmethoden (`newBag()`, `newDict()`,...) lassen sich alle Behältervarianten erzeugen. Der statische Typ des Rückgabewertes ist immer ein entsprechender abstrakter Behälter, in unserem Beispiel eine `conList`. Neben den Klassenobjekt wurde noch ein Wert übergeben (`Factory.array`), der die Fabrik veranlaßt, die Array-Implementation für den Listen-Behälter zu wählen. Alle Fabrikmethoden sind überladen und haben eine unterschiedlich stark spezialisierte Signatur. Die allgemeinste Methode läßt die meisten Optionen offen, die in den anderen Methoden schon fest definiert sind. Hier die allgemeinste Fabrikmethode für Listen:

```
static public conList newList(   Class cls,
                                 int fastSearchLevel,
                                 boolean useEqual)
```

`cls` muß bei allen Methoden angegeben werden und ist das Klassenobjekt der Behälterelemente, `useEqual` bewirkt, falls wahr, die Benutzung des Wertvergleiches (Standardmäßig wahr) zwischen Behälterelementen, ansonsten findet eine Vergleich über die Referenzen der Objekte statt. `fastSearchLevel` kann mit den Klassenwerten `slow`, `average` und `fast` (`fast` wird standardmäßig benutzt) belegt werden, die in gleicher Reihenfolge den Klassenwerten `linkedList`, `array` und `hashTable` entsprechen. Dieser Wert bestimmt die konkrete Behälterimplementation: verkettete Liste, Array oder Hashtable. Diese Reihenfolge richtet sich, wie der Name des Parameters schon sagt, nach der Zeit-Komplexität einer Suche, die bei einer verketteten Liste am größten und bei einer Hashtable am geringsten ist. Gegenläufig dazu ist die Komplexität einer Einfügeoperation, die bei einer verketteten Liste am geringsten und bei einer Hashtable am komplexesten im Bezug auf den Zeitaufwand ist.

Um nun die passende Behälterimplementation zu finden, versucht die Fabrik dynamisch eine Klasse mit passenden Namen zu finden. So setzen sich in der *j*ConLib die Namen der Behälterimplementation nach einem einheitlichen Schema zusammen:

```
Behälter        :=  "con" [ "ID" ] Variante "As" Implementation
Variante        :=  "Array" | "AssList" | "Bag" | "Dict" |
                    "List" | "SAssList" | "Set" | "SList"
Implementation  :=  "Array" | "Hashed" | "LkList"
```

„ID“ steht für Behälter mit Referenzvergleich, sonst Wertvergleich. Diese Notation eröffnet eine große Kombinationsmöglichkeit, die aber nicht unbedingt von der *j*ConLib erfüllt wird[10]. So kann es passieren, daß bestimmte Behälterimplementationen nicht vorhanden sind, denn es ist nur vorgeschrieben, daß es zu einer Behältervariante mindestens eine Implementation geben muß. Oft möchte der Benutzer auch nur irgendeine Behälterimplementation erhalten, oft einfach ‚möglichst schnell‘. So geht auch die Fabrikmethode vor: Es wird versucht, eine passende Implementation zu finden und falls diese nicht vorhanden ist, wird eben versucht, eine andere zu finden. Es ist also nicht garantiert, daß der Benutzer die exakt vorgegeben Behälterimplementation erhält. Diese Vorgehensweise hat aber einen entscheidenden Vorteil: Das Einhängen neuer Implementationen geht ohne irgendeine Änderung im Programmcode vonstatten. Die neue Implementationsklasse muß einfach nur vorhanden sein, um genutzt zu werden. Wird beispielsweise eine schnellere Implementation in bezug auf die Suche als die bisher vorhandene in die *j*ConLib eingefügt und geben die Benutzer der *j*ConLib immer an, sie wollen die schnellstmögliche Implementation nutzen, so wird die neue Implementation automatisch beim nächsten Einbinden der *j*ConLib benutzt. Damit ist ein hoher Grad der Entkopplung von Implementation und abstrakter Benutzung von Behältern erreicht.

Neben Behältern werden auch die Cursor der *j*ConLib an der Fabrikklasse erzeugt. Damit übernimmt die Fabrikklasse die komplette Erzeugung von *j*ConLib-Objekten.

4.4 Der stabile *j*ConLib-Cursor

In der ConLib sind Cursor robust: Ihre Position bleibt bei jeder Veränderung des Behälters erhalten, es sei denn, das Cursorelement wird gelöscht. In diesem Fall muß für den Benutzer die Neupositionierung des Cursors klar und eindeutig geregelt sein, der Cursor ist solange invalide. Die ConLib verfolgt dabei den Ansatz, daß Cursorelemente solange nicht mehr referenziert werden können (würde zum Laufzeitfehler führen), bis der Cursor auf den Behälter neu positioniert ist. Die Methoden zum Vor- und Zurückbewegen des Cursors lassen ihn wieder auf seinen Nachbarelementen aufsetzen, oder auf deren Nachbarelementen, sofern diese auch gelöscht wurden, usw. Somit kommt der Cursor nie in einen undefinierten Zustand, der für den Benutzer nicht klar aus dem aktuellen Cursorzustand hervorgeht.

Die Stabilität erweitert nun dieses Konzept darum, daß ein Cursor auf jeden Fall sein Element referenziert, auch wenn es aus dem Behälter gelöscht wurde [Traub96]. Wird also in der *j*ConLib das Element eines Cursors aus dem Behälter gelöscht, so kann es nur noch über den Cursor erreicht werden, aber nicht mehr über den Behälter oder über andere Cursor, die nicht auf dem Element standen. Die Methode `current()` zum Ermitteln des Cursorelementes behält aber ihre vorherige Bedeutung bei: Sie gibt das Cursorelement wieder, solange sich dieses auch im Behälter befindet. Ist das Element aus dem Behälter gelöscht, so führt das Benutzen zu einem Laufzeitfehler, um kenntlich zu machen, daß das Abfragen eines nicht mehr zum Behälter gehörigen Elementes eine besondere Aktion ist. Dieses kann nun über eine Methode `currentOutsideElement()` noch für den Benutzer des Cursors zugänglich gemacht werden und bleibt solange referenziert, bis der Cursor neu positioniert wurde[11]. So kann der Benutzer des Cursors bei Bedarf dem ‚unter der Hand‘-Löschen von Elementen vorbeugen. Durch die Garbage-Collection wird das Element automatisch aus dem Speicher entfernt, falls es nicht mehr referenziert wird. Das Zustandsdiagramm der *j*ConLib gestaltet sich folgendermaßen:

[10] Eine Ausnahme bildet die Behältervariante Array. Hier gibt es nur eine sinnvolle Implementation durch einen Java-Arrays (conArrayAsArray bzw. conIDArrayAsArray), weshalb die Fabrik diese Implementation driekt aufruft.

[11] Diese Methode hat natürlich die Vorbedingung, daß sich der Cursor im invaliden Zustand befindet

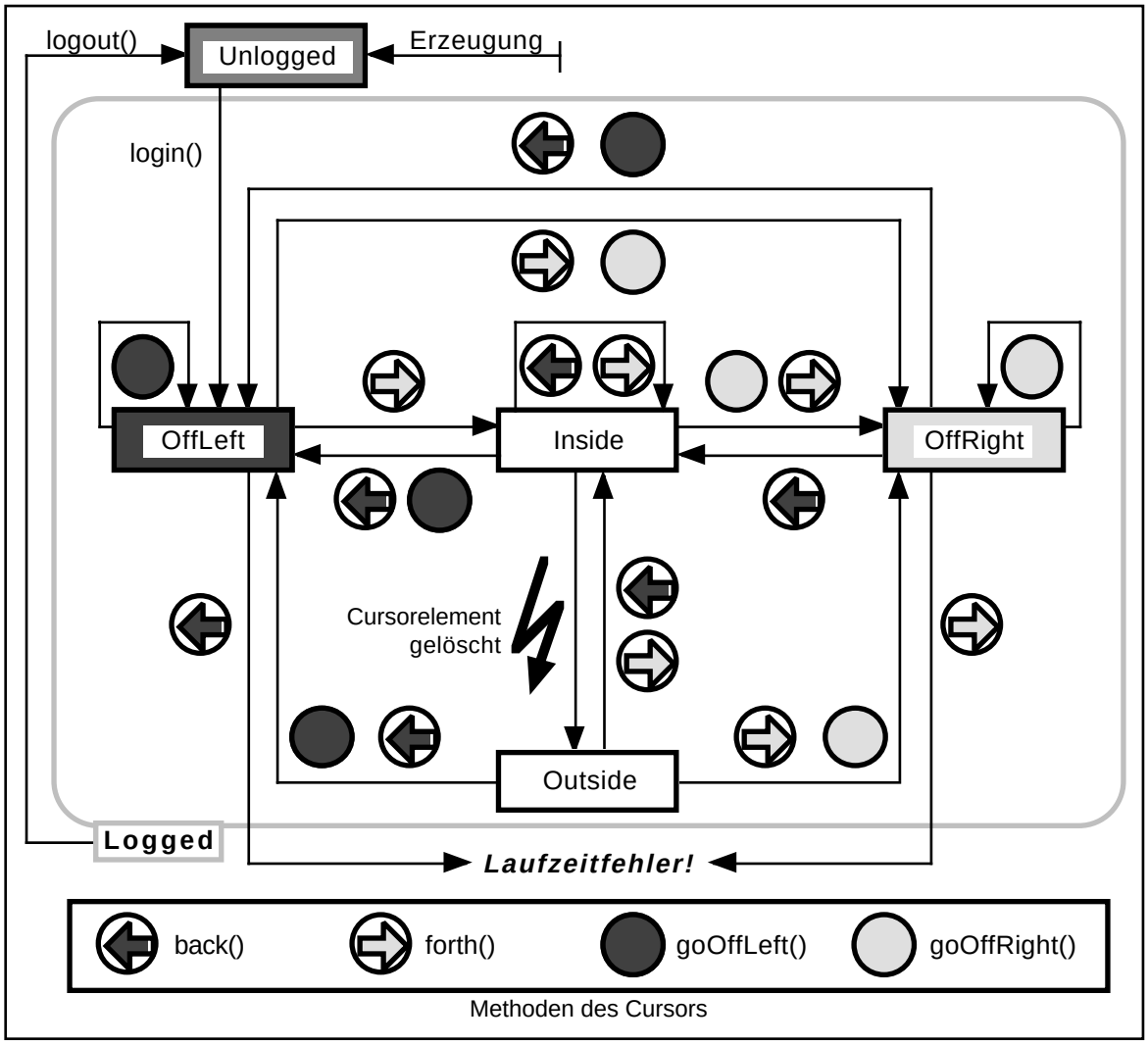

Abbildung 4.2 - Zustandsdiagramm der jConLib (Vergleiche dazu Abb. 2.6)

Vergleichend mit dem Modell des ConLib-Cursors fehlt hier die explizite Aufteilung in Invalid/ Valid als Zustand des Cursors. In der *j*ConLib soll damit deutlich werden, daß alle vier Zustände gleichwertig sind. Die Abfrage `isOffLeft()` kann in der *j*ConLib auch dann stattfinden, wenn sich der Cursor im Zustand ‚Outside' befindet, was mit dem ConLib-Cursor nicht möglich ist, da dort eine Vorbedingung verletzt würde. Der ConLib-Cursor verhindert durch sein robustes Verhalten jede Aktion, wenn er sich in einem invaliden Zustand befindet, außer Handlungen zum Verlassen des invaliden Zustandes. Der stabile *j*ConLib-Cursor hingegen eröffnet ein weniger restriktives Verhalten.

Befindet sich ein Cursor im Zustand ‚OffLeft' bzw. ‚OffRight', so blieb man über `back()` bzw. `forth()` bei der ConLib in diesen Zuständen (siehe Abbildung 2.6). Das Benutzen dieser Methoden in diesem Fall deutet aber auf einen Programmierfehler hin, da das Vor- und Zurückgehen durch den Behälter immer mit einer Zustandsänderung des Cursors verbunden sein sollte. Die Methoden `goOffLeft()` und `goOffRight()` dienen dem expliziten Zurücksetzen des Cursors an die Grenzen des Behälters und sollten unabhängig vom Zustand ausführbar sein. Somit führt in der *j*ConLib ein Benutzen von `back()` und `forth()` in den Zuständen ‚OffLeft' und ‚OffRight' über eine entsprechende Vorbedingung zu einem Laufzeitfehler.

4.4.1 Konzeptuelle Probleme der Stabilität

Zum Erreichen der Stabilität sind allerdings noch andere Ansätze möglich. Dabei geht es um die Frage, wann ein Element, das noch durch einen Cursor referenziert wird, gelöscht werden kann und wie mit dem gelöschten Elementen dann umgegangen wird, ohne daß der Cursor durch das Löschen sein Element verliert. Über das Entfernen des Objektes aus dem Speicher braucht man sich in Java keine Sorge zu machen, da diese Aufgabe durch die Garbage-Collection übernommen wird. Dadurch wird eine einfache Realisierung eines stabilen Cursors erst ermöglich, da das Entfernen des Behälterelementes aus dem Speicher nicht durch, möglicherweise komplexe Algorithmen vom Benutzer, Behälter oder Cursor entschieden werden muß, sondern durch das System selbst.

Eine Methode zur Realisierung von Stabilität ist die sogenannte ‚restriktive Stabilität': Elemente können erst aus dem Behälter gelöscht werden, falls keine Cursor mehr auf dem Behälter eingelogged sind. Da die *j*ConLib-Cursor aber explizit auf Langlebigkeit ausgerichtet sind, ist diese Möglichkeit für die *j*ConLib allerdings nicht praktikabel. Aus gleichem Grunde scheidet auch die Alternative aus, daß gelöschte Elemente erst dann aus dem Behälter entfernt werden, wenn kein Cursor mehr mit dem Behälter verbunden ist. Somit müssen Elemente gelöscht werden können, falls Cursor mit dem Behälter verbunden sind. Eine Methode, die dieses beachtet, wäre, ein Löschen nur dann zuzulassen, falls gerade kein Cursor auf dem Element steht. Das käme einem impliziten Sperren von Elementen durch Cursor gleich und würde den Umgang mit Behältern als einfache Datensammlung einschränken. Solche Überlegungen werfen Fragen bezüglich eines Transaktionskonzeptes auf und würden die Funktionalität eines Behälters doch erheblich erweitern, so daß man diese Überlegungen doch konzeptuell eine Ebene über Behältern ansiedeln sollte (Archive und Datenbanken).

Somit sollte ein Entfernen von Elementen jederzeit möglich sein. Es muß aber geklärt werden, inwieweit das entfernte Element sichtbar bleibt. Denkbar wäre, daß nicht nur der Cursor, der auf dem Element steht, es referenzieren kann, sondern auch noch weitere Cursor, solange der Cursor mit dem gelöschten Element nicht neu positioniert wurde. Das kann aber zu undurchsichtigen Zuständen führen, da das Element trotz Entfernens sichtbar bleibt, bis sich der Cursor des Elements bewegt hat und dieses dann ‚plötzlich' nicht mehr im Behälter vorhanden ist. So sollte nur dem Cursor, der ein entferntes Element referenziert, dieses noch bekannt sein und das Element sofort aus dem Behälter entfernt werden. Ansonsten kommt es zu undefinierten Zuständen für andere Benutzer des Behälters. Ein solcher stabiler Cursor, wie er in der *j*ConLib realisiert ist, hat damit soetwas wie einen ‚Sicherungsmechanismus', der ein Löschen seines Elementes zwar ermöglicht, aber dieses Element noch solange behält, bis ihm explizit durch eine Bewegung mitgeteilt wird, daß er das Element aufzugeben hat.

Dieses Konzept der Stabilität ist auch bei Behältern über Objekte, die Systemressourcen verwalten, nicht unbedingt problematisch. Jedes Objekte kann eine `finalize()`-Methode besitzen, die vor dem Entfernen des Objektes aus dem Speicher aufgerufen wird und in diesem Fall oft zum Freigeben noch benutzter Systemressourcen dient. Löscht ein Benutzer aus einem Behälter ein solches Objekt, könnte er davon ausgehen, daß das Objekt auch entfernt wird und seine Systemressourcen freigegeben werden. Ein stabiler Cursor könnte aber dieses Objekt noch referenzieren, so daß es nicht aus dem Speicher entfernt würde und auch nie die `finalize()`-Methode aufgerufen würde. Da aber Java eine Garbage-Collection besitzt, die Objekte bei Bedarf entfernt, ist es gefährlich, Vermutungen über den Zeitpunkt der Objektfreigabe zu machen. Sollten also Ressourcen unbedingt freigegeben werden, so sollte dieses explizit durch Methodenaufruf geschehen. Damit würde ein Behälter über solche Objekte mit stabilem Cursor kein Problem mehr darstellen.

4.4.2 Typisierung des Cursors

Ein Cursor kann sich bei jedem beliebigen Behälter anmelden: Bei der Erzeugung eines Cursors wird nicht der Typ der Elemente angegeben, die der Cursor liefert. Somit kann beispielsweise ein Cursor einmal auf einem Behälter über Personen arbeiten und anschließend über einen Behälter von Stühlen iterieren (Person und Stuhl stehen wahrscheinlich in keiner Vererbungsbeziehung). Eine dynamische Typisierung wie bei Behältern macht auch wenig Sinn, denn ein Typfehler könnte von einem Cursor nur dann ausgehen, wenn er sich bei einem Behälter anmeldet, da nur dann ein Vergleich der beiden Typen stattfinden könnte. Die Typüberprüfung von Behälterelementen, die vom Cursor zurückgegeben werden, muß durch Typecasts vom Benutzer selbst vorgenommen werden. Eine Typisierung des Cursors ist daher nicht unbedingt erforderlich.

4.5 Interfaces

Interfaces werden in der *j*ConLib einmal zum Realisieren von Serialisierbarkeit benutzt: Alle notwendigen Oberklassen implementieren das Bibliotheksinterface `Serializable`, wobei keine Methode zur Serialisation implementiert werden muß, da diese Aufgabe vollständig vom Java-System übernommen wird. Das Implementieren des Interface `Serializable` dient lediglich dazu, eine serialisierbare Klasse zu markieren.

Für intern sortierte Listen müssen die Behälterelemente eine eigene Ordnungsrelation definieren. Da es bis zum JDK 1.2 keine solche Vergleichsmethodik für Objekte gibt, müssen diese Klassen um die Fähigkeit des Größenvergleichs erweitert werden. Ein ideales Mittel stellt dafür ein Interface dar, welches eine Schnittstelle für eine solche Vergleichsmethodik definiert:

```
public abstract interface Comparable
{
    public abstract boolean isLessThan(Comparable obj);
}
```

Jede Klasse, die nun dieses Interface implementiert, muß die Methode `isLessThan` implementieren und damit eine Ordnungsrelation zwischen den Elementen definieren. Die Methode `isLessThan` erwartet natürlich ein Objekt, welches das Interface `Comparable` erfüllt, `obj` besitzt den Typ `Comparable`.

Die ConLib bietet noch die Möglichkeit an, eine Iterationsfunktion über alle Behälterelemente auszuführen. Diese kann noch mit einer Bedingung gekoppelt werden, so daß eine Iterationsfunktion nur mit Behälterelementen ausgeführt wird, die eine Bedingung erfüllen. In der *j*ConLib wird diese Methodik mit Objekten realisiert, die das Interface `Iterable` und `Condition` erfüllen. `Iterable` beinhaltet lediglich eine Methode, die einen Wert vom Typ `Object` annimmt und nichts zurückgibt. `Condition` nimmt auch einen Wert vom Typ `Object` an und gibt einen Wahrheitswert zurück. Was für Methoden diese Iterations- bzw. Konditionsobjekte haben und mit was für Parametern sie erzeugt werden, bleibt dem Benutzer überlassen. Die Iteration wird wie bei der ConLib durch Methoden an der Klasse `conContainer` ermöglicht.

Jede Behälterart hat in der *j*ConLib ein Interface, welches den fachlichen Umgang mit dem Behälter definiert. Dieses Interface wird jeweils durch eine korrespondierende abstrakte Klasse so weit implementiert, wie sich allgemeine Methoden finden lassen. In den konkreten Klassen, die eine technische Implementation des Behälters darstellen, werden dann auch die verbleibenden Methoden implementiert. Da in diesen abstrakten Klassen auch Methoden vorhanden sind, die eine Kommunikation

zwischen Behälter und Cursor regeln, macht eine Schnittstellendefinition für Benutzer der Behälter Sinn. Diese Methoden müssen das Attribut `public` besitzen, da Cursor und Behälter in unterschiedlichen Packages liegen und nur über `public`-Methoden aufeinander zugreifen können. Da diese Methoden in der Schnittstellendefinition nicht mehr vorkommen, sind sie aber für den Benutzer nicht sichtbar. Gleiches gilt für den Cursor, der auch eine Benutzungsschnittstelle besitzt.

4.6 Integration in das Framework

Die *j*ConLib soll ein fester Bestandteil des JWAM-Frameworks sein. Sie kann aber auch ohne das Framework benutzt werden und ist daher auch von ihm unabhängig. Die einzige Ausnahme ist dabei das Vertragsmodell, von dem die *j*ConLib regen Gebrauch macht.

4.6.1 Package-Aufteilung

Die Package-Struktur der *j*ConLib befindet sich innerhalb des JWAM-Frameworks. So sind sämtliche Packages in der Hierarchie unter ‚wam.system.collection' zu finden. Das JWAM-Framework ist in einer ersten Stufe unter ‚wam.' gruppiert und teilt sich dann in verschiedene Framework-Schichten auf, wobei die Collection in der sogenannten ‚Systembasisschicht' angelegt ist, die als ‚system' abgekürzt wird. [WAM-Buch]

- **wam.system.collection**
  Hier stehen die Fabrikklasse und einige Hilfsklassen und -interfaces der *j*ConLib.
- **wam.system.collection.abstraction**
  Beinhaltet sämtliche Behälterinterfaces. Diese Klassen sollte sich der Benutzer importieren.
- **wam.system.collection.implementation.abst**
  Hier befinden sich die abstrakten Behälterklassen, die ihre korrespondierenden Interfaces implementieren. Diese Klassen sollten nicht vom Benutzer importiert werden.
- **wam.system.collection.implementation.conc**
  Hier sind die Implementationsklassen zu den abstrakten Behälterklassen. Aus diesem Package sollte der Benutzer ebenfalls keine Klassen importieren. Die Instanziierung der Klassen wird von der Fabrik übernommen.
- **wam.system.collection.cursor**
  Cursorinterface und -implementation, sowie Positionsklasse zur Kommunikation zwischen Cursor und Behälter. Nur das Cursorinterface ist für den Benutzer interessant.
- **wam.system.collection.strategy**
  In diesem Package sind sämtliche Strategieklassen der *j*ConLib enthalten. Ist für den Benutzer nur interessant, falls er eine eigene Strategie einfügen möchte.
- **wam.system.collection.utility**
  Eine Assoziationsklasse für Wert/Schlüssel-Behälter und Tupel-Klassen, die vom JWAM-Framework benutzt werden. Die Klassen könnten auch für den Benutzer interessant sein, sofern eine der gebotenen Datenstrukturen gebraucht wird.

Aus dem JWAM-Framework wird das Vertragsmodell, welches sich unter ‚wam.system.contract' befindet, in der *j*ConLib benutzt. Wird neben der *j*ConLib noch das Vertragsmodell aus dem JWAM-Framework herausgenommen, so ist die *j*ConLib eigenständig lauffähig.

4.6.2. Dokumentation

Java bietet über das Tool ‚javadoc' ein Werkzeug an, mit dem sich Klassen- und Methodenbeschreibungen in HTML-Form bringen und mit einem Text beschreiben und dokumentieren lassen. Somit läßt sich auf einfache Weise eine Dokumentation erstellen, die durch die Möglichkeiten in HTML auch noch verknüpfte Dokumente auswirft. In der *j*ConLib wird davon Gebrauch gemacht. So sind alle Klassen und ihre Methoden vollständig dokumentiert, sofern diese von einem Benutzer verwendet werden. Zwar ersetzt dieses keinen Leitfaden zur Benutzung, der sollte noch immer nebenbei erstellt werden, wobei sich wieder HTML anbietet. Die HTML-Dokumentation der Klassen und Methoden bieten, so die Erfahrung mit dem JWAM-Framework, eine gute Nachschlagemöglichkeit. [JWAM98].

4.7 Die Portierung

Wenn man von den in diesem Kapitel erläuterten Änderungen absieht, konnte ich die ConLib ohne ‚rauhen Ecken' nach Java übertragen. In der Benutzung sollte sich die *j*ConLib von der ConLib nicht wesentlich unterscheiden.

Was den Vorgang der Portierung angeht, so konnte ich zumindest bei der Implementation der Behälterhierarchie den vorhandenen Code relativ einfach übersetzen. Natürlich mußte ich dabei auf die Gegebenheiten der Sprache Java eingehen. Einige Änderungen warf die dynamische Typprüfung auf, denn jeder Übergabewert an die Behälter muß auf seinen korrekten Typ überprüft werden. Auch der stabile Cursor verlangte einige Änderungen. Die Erzeugung von Behältern durch die Fabrikklasse erwirkte, daß die *j*ConLib frühzeitig, auch ohne viele Implementationen von Behältern, eingesetzt werden konnte. In der aktuellen Version sind nur Behälterimplementationen vorhanden, die auf einem Array basieren. Hier müssen allerdings noch weitere Implementationsklassen hinzugefügt werden, die Dank der Fabrikklassenkonstruktion sehr einfach in bestehende Systeme integriert werden können.

Ein weiterer Aspekt, der erst durch Java hinzukam, ist die Synchronisation von Behälter. Java ist eine Sprache die Nebenläufigkeit unterstützt: Mehrere Threads können sich eine Systemressource teilen. Damit es wenigstens nicht innerhalb von Behältermethoden zu inkonsistenten Zuständen durch parallel am Behälter arbeitenden Threads kommt, müssen die Behältermethoden durch Sperrung ihrer Systemressourcen mit dem Sprachkonstrukt ‚`synchronized`' gegen inkonsistente Zustände geschützt werden. So besitzen alle Methoden dieses Schlüsselwort, sofern sie den Behälter verändern oder keine atomaren Aktionen ausführen. Referenzen des Objektes oder Übergabereferenzen mußten auch synchronisiert werden, falls mit ihnen ebenfalls keine atomaren Aktionen ausgeführt oder diese verändert werden. Eine solche Problematik tauchte bei der Konstruktion der ConLib mangels Nebenläufigkeit der Sprache C++ überhaupt nicht auf.

V. Vergleich mit anderen Behälterbibliotheken

Abschließend vergleiche ich die *j*ConLib noch mit existierenden Behälterbibliotheken für Java. Nach einem kleinen Überblick über die Standardklassen in JDK 1.0/1.1, die eine sehr rudimentäre Behälterfunktionalität liefern, erläutern ich zuerst die Vergleichsmethodik. Der Vergleich soll vor allen Dingen folgende Gesichtspunkte betrachten: Aufbau der Behälterhierarchie, Iterator- bzw. Cursor-Konzepte, Methoden zur Manipulation der Behälter, Einbindung alter Java-Strukturen sowie die Einbindung technischer Gegebenheiten in Java (Serialisierung und Synchronisierung) und die Handhabung und Benutzung der jeweiligen Bibliothek. Dabei betrachte ich die Behälterbibliothek des kommenden JDK 1.2, die Collection API, und die frei verfügbare und von vielen Java-Entwicklern genutzte Java-Generic-Library (JGL) von der Firma Objectspace. Die JGL erhebt den Anspruch, eine Portierung der Standard-Template-Library (STL) aus der C++-Welt zu sein. Ein Umstand, der eine genauere Betrachtung verdient, da die *j*ConLib ja auch eine Portierung aus der C++-Welt ist. Abschließend werde ich noch alle Merkmale zusammenfassen und die drei Behälterbibliotheken gegenüberstellen.

5.1 Das java.util-Package

Java verfügt bis einschließlich des JDK 1.1 über keine nennenswerte Behälterfunktionalität. Es gibt zwar einige Klassen, die soetwas wie eine Behälterfunktionalität anbieten, jedoch kann von abstrakten Konzepten nicht die Rede sein, auch gibt es keine Behälterhierarchie. Da aber die folgenden Behälterbibliotheken die Möglichkeit zur Integration dieser alten Strukturen bieten, hier eine kleine Übersicht der Klassen:

- **`Vector`**
  Bietet einen Behälter ohne fachliche Größenbeschränkung an. Die Behälterelemente sind über einen Index zugreifbar.
- **`Stack`**
  Erbt (!) von `Vector` und ist um eine Stapel-Funktionalität erweitert.
- **`Dictionary`**
  Eine abstrakte Klasse, die einen Umgang für Wert/Schlüssel-Behälter vordefiniert.
- **`Hashtable`**
  Ein Behälter für Wert/Schlüssel-Paare ohne fachliche Größenbeschränkung. Erbt von `Dictionary`.

Die Behälterelemente sind als `Object` typisiert, der dynamische Typ wird in allen Behälterbibliotheken nicht eingeschränkt. Über die Behälter kann nur mit Hilfe eines `Enumerator`-Objektes iteriert werden. Dieses Objekt läßt man sich über eine Methode des Behälters erzeugen um über alle seine Elemente zu iterieren. Eine Veränderung des Behälters ist mit Enumeratoren nicht möglich.

5.2 Vorgehensweise des Vergleiches

Die vom java.util-Package gebotene Struktur ist kaum als Behälterbibliothek zu bezeichnen. Deswegen findet sich im JDK 1.2 auch eine Collection-API wieder, die im Vergleich zu dem bisher vorhandenen wesentlich erweitet ist. Von vielen Java-Entwicklern wurde bislang die JGL der Firma

Objectspace als Behälterklassensammlung benutzt, die auch frei verfügbar ist. Beide Bibliotheken möchte ich jeweils kurz beschreiben und mit der *j*ConLib vergleichen, wobei ich auf folgende Gesichtspunkte achten werde:

- Aufbau der Behälterhierarchie
- Methoden zur Manipulation der Behälter
- Iterator- bzw. Cursor-Konzepte
- Einbindung technischer Gegebenheiten in Java (Serialisierung und Synchronisierung)
- Handhabung und Benutzung der jeweiligen Bibliothek.

Ein dynamische Typprüfung gibt es in keiner der beiden Bibliotheken. Immerhin wird in der Collection-API zu diesem Thema darauf verwiesen, daß wenn parametrisierten Polymorphismus in Java unterstützt werden sollte, er auch in der Bibliothek benutzt werden wird [CollAPI-FAQ97].

Beiden Bibliotheken besitzen die Gemeinsamkeit, daß sie Strukturen wie Arrays oder Vektoren in ihre neuen Behälter einbinden können und dafür Umformungs- und Adaptionsfunktionen bereitstellen. Beide Behälterbibliotheken unterstützen die Enumeratoren aus dem alten java.util-Package.

5.3 Die Collection API des JDK 1.2

Die Collection-API hegt den Anspruch, eine kleine Behältersammlung zu sein, die ohne „konzeptuelle Schwergewichte“ auskommt [Collection-API97]. Es soll dem Benutzer eine überschaubare Klassensammlung angeboten werden, die essentielle Behältermethoden beinhalten. Die Collection-API soll die bisher im JDK fehlende, grundlegende Behältersammlung sein.

5.3.1 Aufbau der Behälterhierarchie

Die Behälterhierarchie ist recht einfach aufgebaut und ist der der *j*ConLib vom Prinzip her ähnlich. Es gibt drei Behälterarten und einen allgemeinen Behälter:

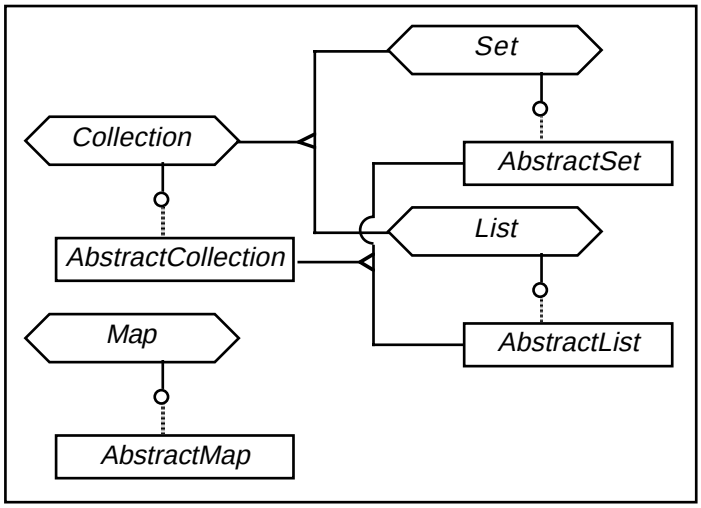

Abbildung 5.1 - Collection-API-Behälterhierarchie

Die Hierarchie bewegt sich auf zwei Ebenen: Es gibt einmal die Interfaces, die den Umgang mit den Behälter abstrakt und fachlich definieren. Eine `Collection` definiert einen allgemeinen Behälter, ein `Set` spezialisiert einen Behälter nach der Mengen-Metapher und eine `List` ist ein indizierter Behälter mit expliziter Ordnung. `Map` ist eine eigenständige Behältervariante für Wert/Schlüssel-Paare. Sie ist nicht fachlich unter der `Collection` gefaßt.

Auf der zweiten Ebene werden diese Schnittstellen durch abstrakte Klassen implementiert, die noch keine Aussage über die technische Implementierung des Behälters machen: `AbstractCollection`, `AbstractSet`, `AbstractList`, und `AbstractMap`. In diesen Klassen wird eine Behälterfunktionalität skelettartig implementiert. Von den abstrakten Klassen erben dann konkrete Klassen, die eventuell (siehe nächster Abschnitt) letzte Methoden implementieren. Diese Klassen können verschiedene technische Realisierungen besitzen: Hashtable, in der Größe veränderbares Array, verkettete Liste oder einen ‚balanced Tree'. Für die verschiedenen Behälterarten ist nicht jede Implementation sinnvoll, so daß in der Collection-API nicht jede Kombination vorkommt.

5.3.2 Methoden zur Manipulation des Behälters

Die Collection API bietet nur eine sehr kleine Auswahl an Methoden zur Behältermanipulation. Diese Methoden müssen nicht einmal implementiert sein, denn die Collection API bietet eine Runtime-Exception namens `UnsupportedOperationException` an, die eine nicht vorhandene Verfügbarkeit der Methode meldet. So kann der in der *j*ConLib strikt vermiedene Fall vorkommen, daß Methoden in tiefer liegenden Vererbungshierarchien nicht mehr zu benutzen sind. Dementsprechend sind einige Methoden in den abstrakten Klassen, die von dem Interface `Collection` vorgegeben werden und die von konkreten Implementation erfüllt werden müssen, nicht wieder abstrakt definiert, sondern lösen bei Benutzung eine solche Exception aus. Dieses Verhalten wird damit begründet, daß auch unmodifizierbare Behälter von abstrakten Klassen erben und dort diese Funktionen nicht vorhanden seien. Alle modifizierbaren Behälter überschreiben aber solche Methoden der abstrakten Klassen. Unmodifizierbare Behälter sind nicht sichtbar Bestandteil der Behälterhierarchie sondern müssen an einer Fabrikklasse (`Collections`) mit einem bereits erzeugten Behälter kreiert werden. Es handelt sich dabei um eine ‚anonymous Class', dazu aber mehr in Kapitel 5.3.4.

5.3.3 Cursor bzw. Iterator-Konzepte

Die Collection-API-Iteratoren sind sehr an dem Modell der Enumeratoren angelehnt. Sie dienen dem einfachen Durchfahren des Behälters, sind aber noch durch eine `Remove()`-Methode erweitert, die das aktuelle Element löscht. Wird der Behälter, außer durch den Iterator selbst, irgendwie verändert, wird bei Benutzung des Iterators eine Runtime-Exception ausgelöst, falls diese Änderung auch den Iterator betrifft.

Neben diesem allgemeinen Iterator gibt es noch einen speziellen `ListIterator`, der nur über Listen laufen kann. Dieser kann sich vor- und zurückbewegen, Elemente hinzufügen oder verändern, sowie den Index der Elemente erfahren. Das Verhalten bei sich anderweitig verändernden Behältern ist identisch.

Die Umgangsformen der Iteratoren sind in Interfaces definiert, die durch konkrete Iteratoren für verschiedenen Behälter realisiert werden müssen. So gibt es in der Collection-API eine Vielzahl von Iterator-Klassen, die das gleichnamige `Iterator`-Interface implementieren. Iteratoren müssen für alle verschiedene Behälter implementiert werden, wobei für die vorhandenen Behälterarten diese teilweise zusammengefaßt werden können.

5.3.4 Einbindung technischer Gegebenheiten

Die Behälter der Collection-API lassen sich komplett serialisieren. Dieses gilt allerdings nicht für die Iteratoren, sie implementieren nicht das `Serializable`-Interface. Das entspricht auch dem Konzept, daß Iteratoren die Elemente der Behälter nicht über einen längeren Zeitraum referenzieren, sondern Anwendungen die Möglichkeit geben, über alle Elemente des Behälters eine Funktion auszuüben. Sie sind nicht auf Dauerhaftigkeit ausgerichtet.

Die Behälter der Collection-API sind nicht synchronisiert. Wie nicht veränderbare Behälter können Behälter aber synchronisiert werden, indem Behälter-Objekte entsprechenden Methoden an einer Fabrikklasse (`Collections`) übergeben werden und man einen synchronisierten Behälter erhält. Die synchronisierten Behälter stehen wie die unmodifizierbaren Behälter außerhalb der Behälterhierarchie. Diese Vorgehensweise bei der Synchronisation wird damit begründet, daß man Applikationen, die kein Multi-Threading benutzen, durch das Synchronisieren keine Zeit stehlen will.

Die Behälter der Collection-API implementieren das Interface `Cloneable` und können somit kopiert werden. Das Kopieren beschränkt sich auf den Behälter, die Elemente des Behälters werden nicht kopiert, sie müssen also nicht kopierfähig sein.

Die Collection-API benutzt im großen Stil die Möglichkeiten der sogenannten ‚Inner Classes', die es seit Java 1.1 im Sprachstandard gibt. Diese Klassen werden grundsätzlich innerhalb anderer Klassen definiert, so daß ihr Namensraum innerhalb dieser Klassen liegt. Es gibt drei Arten dieser Klassen: ‚Member Classes' sind Klassen, die textuell auf Methodenebene definiert sind und deren Instanzen Zugriff auf alle Methoden und Werte der Instanz der umschließenden Klasse haben. Eine Objekt der ‚Member Class' ist dabei immer mit einem Objekt der umschließenden Klasse verbunden und wird auch an diesem umschließenden Objekt instanziiert. ‚Local Classes' werden innerhalb von Blockstrukturen definiert und können zusätzlich Zugang zu lokalen, konstanten Variablen haben. Eine ‚Anonymous Class' ist eine ‚Local Class' ohne vordefinierten Namen, wobei sie entweder ein bestimmtes Interface implementiert oder eine Klasse über einer Vererbungsbeziehung spezialisiert. Die Definition der Methoden wird durch einen geklammerten Block hinter der Erzeugung durch `new` durchgeführt. In der Collection-API wird von diesen Klassen reichlich Gebrauch gemacht. Da z.B. Iteratoren für die unterschiedlichen Behälter geschrieben werden müssen, werden sie mittels ‚Anonymous Classes' innerhalb der Behälter definiert. Diese Technik vereinfacht die Konstruktion, denn die Iteratoren können innerhalb der Behälter definiert werden, deren private Werte und Methoden nutzen und damit einfach auf die unterschiedlichen Gegebenheiten der Behälter eingehen. Die Klassensammlung wird außerdem nicht durch unterschiedliche Iterator-Implementierungen aufgeblasen. Da die Iteratoren abstrakte Interfaces implementieren, bleibt ihr Verhalten für den Benutzer einheitlich.

5.3.5 Handhabung und Benutzung der Collection API

Die Benutzung der Collection-API gestaltet sich recht einfach. Will man beispielsweise einen Listen-Behälter erzeugen, so ist das kurz zu formulieren:

```
List    list;

list = new ArrayList();
```

`List` ist das abstrakte Interface, über welches man auf den Behälter zugreifen sollte. Die Methodenaufrufe sehen der *j*ConLib ähnlich, wegen fehlender dynamischer Typisierung können jedoch in einen Behälter beliebige Elemente eingefügt werden:

```
list.add(new Integer(3));
if(list.contains(new Integer(3)))       // Ist Wahr
{
    list.add("Text");
}
```

In den Behältern findet immer ein Vergleich über die `equals()`-Methode der enthaltenden Behälterelemente statt. Bei den Objektkapseln der Basiswerte ist die `equals()`-Methode überschrieben und es findet ein Wertvergleich statt. Gleiches gilt für Strings, die einen Wertvergleich ebenfalls eingebaut haben. Iteratoren werden beispielsweise so benutzt:

```
Iterator iter = list.iterator();
while(iter.hasNext())
{
    System.out.println(iter.next());
}

iter = list.iterator();
while(iter.hasNext())
{
    if(iter.next() == "Text")
    {
        iter.remove();      // Loescht das aktuelle Element
    }                       // des Iterators
}
```

Die Behälter sind Fabriken für Iteratoren und auch für die Implementation dieser zuständig. Das zweite Beispiel zeigt, wie ein Iterator verändernd auf die Behälterstruktur einwirken kann. Soll nun ein Behälter gegen Modifikationen geschützt werden, wird dieses an einer Fabrikklasse erledigt:

```
list = Collections.unmodifiableList(list);

list.add(new Float(1.3));
// Fehler:
// Fuehrt zu einer Runtime-Exception:
// java.lang.UnsupportedOperationException
```

Das Synchronisieren wird ebenfalls an derselben Klasse durchgeführt. In beiden Fällen kommen wieder ‚Inner Classes' zum Einsatz, die in der Collection-Klasse definiert sind. Sie ummanteln die übergeben Collections mit den entsprechenden Fähigkeiten, wie im obigen Beispiel, daß sie die `add()`-Methode nicht an die Collection weiterreichen, sondern eine Exception erzeugen.

5.4 Die JGL von Objectspace

Die JGL ist in der aktuellen Version 3.0 eine recht umfangreiche Bibliothek mit ca. 200 Klassen und Interfaces[12]. Sie hat den Anspruch, eine reichhaltige Behälterbibliothek zu sein und Benutzern ebenfalls reichhaltige Möglichkeiten zur Manipulation der Behälter zu geben. In der JGL gibt es eine strikte Trennung zwischen Datenstrukturen (den Behältern) und Algorithmen, die die Behälter manipulieren. Die Ursprünge der JGL liegen in der Standard Template Library (STL), die aus der C++-Welt kommt.

5.4.1 Aufbau der Behälterhierarchie

Die Behälterhierarchie der JGL ist komplexer im Vergleich zur der der Collection-API:

[12] Zum Vergleich: Die Collection-API hat 23 neue Klassen und Interfaces (ohne Inner Classes), die *j*ConLib umfaßt 88 Klassen und Interfaces.

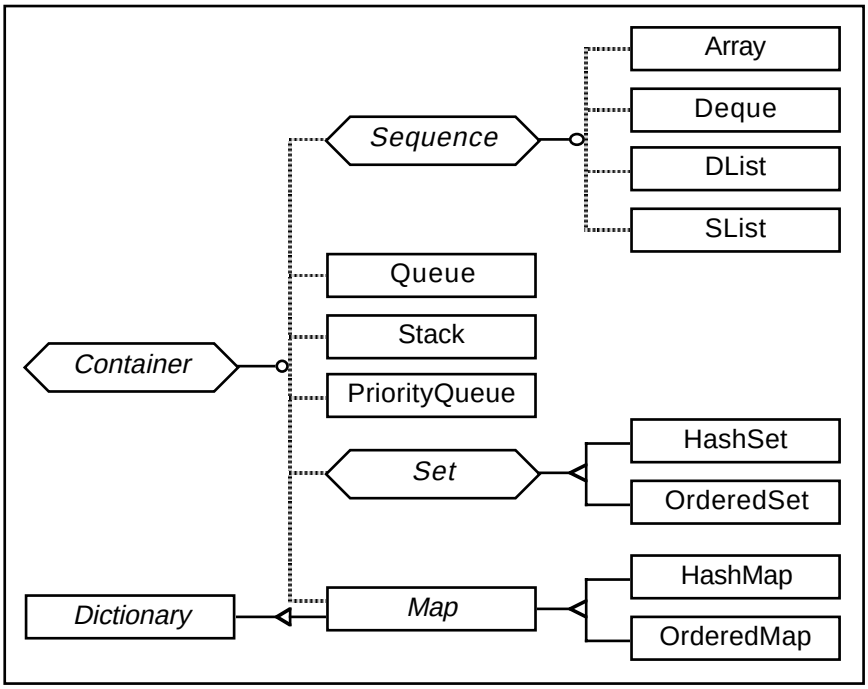

Abbildung 5.2 - Die Behälterhierarchie der JGL

Die Behälterhierarchie wird zunächst über Schnittstellen definiert, die aber gleich von konkreten Behältern implementiert werden. `Container` ist die Schnittstelle für einen allgemeinen Behälter, `Sequence` steht für Behälter mit einer externen Ordnung über Indizes und `Set` für Mengen-Behälter. `Map` bildet eine abstrakte Oberklasse, da es von `Dictionary` aus dem java.util-Package erbt und daher kein Interface sein kann, und steht für allgemeine Wert/Schlüssel-Behälter. Die konkreten Behälter implementieren die passenden Interfaces und unterschieden sich in der technischen Realisierung oder bilden neue Behältervarianten. Die Behälter, die die Interfaces `Sequence` und `Set` implementieren bzw. die von der Oberklasse `Map` erben, sind technische Spezialisierungen: `Array` benutzt ein klassisches Array, `Deque` steht für ein Array aus Blöcken, die Elemente enthalten, `Dlist` für eine doppelt verkettete Liste und `Slist` für eine einfach verkettete Liste. `HashMap` und `OrderedMap` sind Implementierungen einer `Map`, erstere benutzt eine Hashtable, letztere ordnet die Elemente nach ihrem Schlüssel für eine schnelle Suche. Gleiches gilt für die Implementierungen des Mengen-Behälters, `HashSet` und `OrderedSet`. Die restlichen drei Behälter implementieren nur das Interface `Container` und erweitern ihr fachliches Verhalten: `Stack` ist ein Stapel-Behälter und verwendet intern einen JGL-`Array`, `Queue` realisiert einen Schlangen-Behälter, der intern eine `Slist` benutzt, während `PriorityQueue` der Reihenfolge in der Schlange noch eine Ordnungsrelation beibringt und intern ein JGL-`Array` benutzt. Allerdings können bei Instanziierung einer `Queue` oder eines `Stack` andere technische Realisierungen einer `Sequence` als darunterliegender Behälter übergeben werden.

Neben diesen Behältern bietet die JGL noch Adaptoren-Klassen an, die native Java-Arrays über Basiswerte und `Object` als JGL-Behälter kapseln können. Objekte der Klasse `Vector` aus dem java.util-Package können auf diese Weise ebenfalls zu JGL-Behältern werden. Für jedes unterschiedliche Array existiert eine eigene Adaptor-Klasse, sowie noch eine `Vector`-Adaptor-Klasse. Bei der Instanziierung einer solchen Klasse wird eine entsprechende Datenstruktur übergeben. Alle Klassen sind konkret und erben von einer abstrakten Oberklasse `ArrayAdaptor`, die wiederum eine Sequence implementiert. Durch die Einbindung der `Dictionary`-Klasse in die Hierarchie ist noch eine Brücke zum java.util-Package geschlagen.

5.4.2 Methoden zur Manipulation der Behälter

In der JGL gibt es eine strikte Trennung zwischen Datenstrukturen und Algorithmen auf diesen Datenstrukturen. So sind die Behälter reine Datenstrukturen, die Objekte beinhalten. Sie sollen nur Umgangsformen anbieten, die dem Verwalten von Elementen dienen und die je nach Behälterart verschieden spezialisiert sind.

Zur Manipulation der Behälter stehen einige Algorithmen-Klassen zur Verfügung, in denen Klassenmethoden aufgerufen werden: Sortieren und andere Umgruppierungsarten, verschiedene Arten des Ersetzen von Elementen, Ermittlung des kleinsten und größten Elementes, aber auch Algorithmen zum Durchzählen der Elemente oder zum Auffinden von Elementen, etc. Manche Algorithmen kommen ‚doppelt vor': Sie sind als Algorithmen, aber auch in den Methoden der Behälter vorhanden. Die Algorithmen können bzw. müssen noch mit weiteren Funktionen verbunden werden. Soll das minmale oder maximale Element eines Behälters gefunden werden, muß eine Vergleichsmethodik bereitgestellt werden. In der JGL geschieht jenes mit der Hilfe von binären Prädikatsfunktionen, die zwei Objekte vom Typ `Object` vergleichen und und einen Wahrheitswert zurückgeben. Zum Einschränken von Behälterelementen gibt es noch unäre Prädikatsfunktionen und zum direkten Manipulieren der Behälterelemente binäre und unäre Funktionen, die ebenfalls Objekte von Typ `Object` erwarten und ein solches als Ergebnis zurückgeben. Alle vier Funktionsarten sind über ein Interface spezifiziert, die konkreten Funktionen und Prädikate werden durch eigenständige Klassen definiert, mit deren Objekten letztendlich gearbeitet wird.

Die JGL macht von einer `InvalidOperation`-Exception Gebrauch, falls Methoden in spezialisierten Behältern nicht mehr vorkommen sollen. Diese Technik wird allerdings nur bei Array-Adaptoren verwendet, denn sie lassen keine Strukturveränderungen des Behälters zu.

5.4.3 Iterator- bzw. Cursor-Konzepte

In der JGL existiert eine ganze Iterator-Hierarchie. JGL-Iteratoren können Elemente verändern, sich in eine oder zwei Richtungen bewegen, oder ihre Position relativ zum Anfang des Behälters ermitteln. Daraus ergibt sich eine Hierarchie von Schnittstellen, die schließlich durch konkrete Iteratoren für die verschiedenen Behälter implementiert werden. So erfüllt der Iterator eines JGL-Arrays das Interface `RandomAccessIterator`, welches den Index der aktuellen Iteratorposition liefert. Bei der technischen Realisierung eines Arrays ist dieses einfach über den Index des Array möglich. Eine doppelt verkettete Liste implementiert das weiter generalisierte Interface namens `Bidirectional-Iterator`, welches ein Vor- und Zurückbewegen ermöglicht. Das Interface `ForwardIterator`, einen Schritt weiter generalisiert, ermöglicht nur die Bewegung in eine Richtung und wird vom Iterator einer einfach verketteten Liste implementiert. Hier ist nur die Bewegung in einer Richtung technisch einfach zu modellieren. Die Schnittstelle `Container` definiert eine Methode namens `start()`, die einen Iterator vom allgemeinen statischen Typ `ForwardIterator` zur Verfügung stellt, der am Anfang (am linken Rand) des Behälters positioniert ist. Eine `finish()`-Methode liefert einen Iterator, der, um weiter in Cursor-Zuständen der *j*ConLib zu sprechen, am rechten Rand des Behälters positioniert ist. Iteratoren können untereinander ihren Abstand ermitteln.

Iteratoren über Behälter können seine Elemente verändern, indem sie das aktuelle Element durch ein anderes ersetzen. Ein JGL-Behälter kann in seiner Struktur verändert werden, solange ein Iterator auf ihm aktiv ist. Werden Elemente hinzugefügt oder gelöscht, reagieren die Iteratoren für die entsprechenden Behälter unterschiedlich und verändern je nach Iterator ihre Position. Somit eignen sich auch in der JGL Iteratoren nur zum kurzfristigen Verändern des Behälters.

5.4.4 Einbindung technischer Gegebenheiten

Die JGL-Behälter sind synchronisiert und daher auch Thread-Safe. Das gilt aber nicht für die Iteratoren. Die Konstrukteure der JGL nahmen den Geschwindigkeitsverlust für Single-Task-Applikationen in Kauf. Iteratoren und Behälter implementieren die Schnittstelle `Cloneable` und können somit kopiert werden. Wird ein Behälter kopiert, so wir nur der Behälter an sich kopiert und nicht seine Elemente. Interessanter ist die Zuweisung des `Serializable`-Interface: Sämtliche Behälter lassen sich serialisieren, aber auch einige Iteratoren. Ob ein Iterator serialisierbar ist oder nicht entscheidet immer der konkrete Iterator, es ist in keiner der abstrakten Schnittstellen festgeschrieben. Serialisierbar sind auch sämtliche unäre und binäre Funktionen und Prädikate, obwohl diese manchmal keinen serialisierbaren Zustand besitzen.

5.4.5 Handhabung und Benutzung der JGL

Um nun einen Behälter zu erzeugen, suche man sich einen passenden konkreten Behälter und sollte auf ihn über seine Schnittstelle zugreifen. Hier ein Beispiel mit einer doppelt verketteten Liste, die ich als Sequenz betrachten möchte:

```
Sequence list;

list = new DList();
```

Die Behälter verfügen über einige Standard-Methoden, hier beispielsweise das Hinzufügen von Elementen und Ausleeren des Behälters:

```
list.add("Eins");
list.add(new Float(2.01));
if(list.contains("Eins"))       // Ist Wahr
{
    list.clear();
}
```

Auch in der JGL verwaltet der Behälter nur Objekte vom Typ `Object`. Somit lassen sich auch unterschiedlichste Objekte in den Behälter stopfen. Um nun einen Behälter zu verändern, bietet die JGL Algorithmen und Funktionen an. Gerade im Hinblick auf das Fehlen einer Typisierung eröffnet dieses völlig neue Möglichkeiten:

```
list.add(new Long(1));
list.add("Eins");
list.add(new Float(2));
list.add("Zwei");
Sorting.sort(list, new LessString());
```

Hier wird eine Sortierfunktion an der Klasse `Sorting` aufgerufen, die eine `Sequence` mit Hilfe eines binären Prädikates sortiert, in diesem Falle ein Vergleich über Strings. Die Sortierung erfolgt hier aufsteigend. Die numerischen Typen werden dabei ebenfalls sortiert, indem sie über ihre `toString()`-Methode in einen String umgewandelt werden. Da diese Methode in `Object` definiert ist und somit in jedem Objekt benutzt bzw. überschrieben sein sollte, läßt sich diese Sortierfunktion auf jeden beliebigen Behälter anwenden. Andere Prädikate, z.B. `LessNumber`, versuchen zur Laufzeit den Typ des Übergabeparameters in einen numerischen Typ umzuwandeln, was in diesem Fall wegen der im Behälter befindlichen Strings nicht funktionieren würde.

Iteratoren sind im Vergleich zur *j*ConLib und zur Collection-API etwas anders zu handhaben:

```
ForwardIterator iter = list.start();
while(!iter.atEnd())
{
    System.out.println(iter.get());
    iter.advance();
}

iter = list.start();
while(!iter.atEnd())
{
    if(iter.get() == "Zwei")
    {
        iter.put(new Integer(2));
    }
    iter.advance();
}
```

So muß der Schritt zum Weiterziehen des Iterators durch einen eigenen Methodenaufruf vollzogen werden und kann nicht zusammen mit der Abbruchbedingung in der Schleife formuliert werden. Die zweiter Iterator-Schleife ersetzt jedes Vorkommen des Strings „Zwei“ durch ein Integerobjekt. Die Iteratoren verhalten sich je nach technischer Implementierung sehr unterschiedlich:

```
Sequence    array = new Array();
Sequence    dlist = new DList();

array.add("Eins");
dlist.add("Eins");
array.add("Zwei");
dlist.add("Zwei");
ForwardIterator     arrayIter = array.start();
ForwardIterator     dlistIter = dlist.start();
array.remove("Eins");
dlist.remove("Eins");
System.out.println(arrayIter.get());     // Ausgabe: Zwei
System.out.println(dlistIter.get());     // Ausgabe: Eins
```

Während sich der Array-Iterator die Position des Indexes merkt und somit das Element „Zwei“ beim Löschen auf die Position des Indexes des Iterator rutscht, merkt sich der Iterator der verketteten Liste einen Eintrag als Referenz. Dieser geht für den Iterator auch durch das Löschen nicht verloren, weshalb die „Eins“ auch nach dem Löschvorgang ausgegeben wird. Um solche, nicht nachzuvollziehenden Fälle zu vermeiden, sollte der Behälter auf keinen Fall während eines Iterator-Vorganges verändert werden.

5.4.6 Die JGL als Portierung der STL

Die Konzepte der JGL kommen nicht aus dem Nichts. Die JGL ist eine Umsetzung der Standard Template Library (STL) aus der C++-Welt, genau wie die *j*ConLib eine Java-Umsetzung der ConLib ist. Die Umsetzung erfolgte aber nicht immer 1:1, es gibt in der JGL einige Unterschiede in der Konstruktion, jedoch nicht unbedingt in der Konzeption. So unterscheidet sich die Container-Hierarchie der JGL von der der STL, was wohl vorallem auf die Einbeziehung des java.util-Package zurückzuführen ist. Die Konstruktion von Algorithmen, Funktionen, Prädikaten und Iteratoren entspricht aber im wesentlichen der STL-Konstruktion. Das Konzept von Adaptor-Klassen kommt auch

in der STL vor, allerdings gibt es hier ebenfalls Unterschiede. In der JGL gibt es zusätzliche Adaptor-Klassen, um beispielsweise native Java-Arrays an die JGL anzubinden. Die Behälter `Stack` und `Queue` können optional einen beliebigen Behälter, der die Schnittstelle `Sequence` benutzt, als technischen Behälter einbinden (adaptieren). Im Gegensatz zu STL funktioniert dieses nicht über Templates, da diese in Java ja bekanntermaßen nicht vorhanden sind. In der JGL gibt es auch keine Möglichkeit, die statische Typsicherheit der STL nachzubilden. Im Gegenteil, denn JGL-Behälter können Objekte unterschiedlicher Typen beherbergen, da einzig und allein eine Typisierung über `Object` erfolgt.

5.5 Abschließender Vergleich

Die drei Behälterbibliotheken haben unterschiedliche Ansprüche: Bei der *j*ConLib habe ich ganz nach dem Vorbild der ConLib, viel Wert auf eine fachlich korrekte Modellierung gelegt. Außerdem handelt es sich bei der *j*ConLib im Hinblick auf das Verwenden alter Strukturen um eine Neuentwicklung, denn durch die Einbindung in das JWAM-Framework müssen keine alten Strukturen adaptiert werden. So gibt es in der *j*ConLib keine Mittel, native Java-Arrays oder Objekte aus dem java.util-Package zu adaptieren. Eine solche Einbindung könnte aber ohne große Probleme in die *j*ConLib integriert werden. Denkbar wäre eine Erweiterung der Fabrikklasse, um Arrays oder Vektoren in *j*ConLib-Behälter umzuwandeln.

Die Collection-API hat sich ein anderes Ziel gesteckt: Die mangelhafte Behälterfunktionalität der bisherigen JDKs sollte um eine neue Behältersammlung ergänzt werden. Die Collection-API sollte ein einfaches Design besitzen und zumindest einige rudimentäre Behälter-Funktionen bieten. Das ist auch gelungen, allerdings gibt es meiner Meinung nach auch ein paar Probleme. So ist die Behälterhierarchie zwar sehr einfach, dafür stehen dann beispielsweise unmodifizierbare Behälter außerhalb der sichtbaren Hierarchie. Da die Wert/Schlüssel-Behälter eine eigene Behälterhierarchie entwickeln, bilden sie gesonderte Behälter und stehen nicht unter dem Interface `Collection`. Allerdings gibt es in der Collection-API eine strikte Trennung zwischen Implementation und Verhalten des Behälters, was die Austauschbarkeit von Implementation vereinfacht. Warum in der Collection-API Behälter nicht grundsätzlich synchronisiert sind, obwohl Behälter eine grundlegende Position in Programmen inne haben, kann ich nicht ganz nachvollziehen. Offensichtlich werden hier Performance-Gründe in den Vordergrund gestellt, falls Applikationen ohne Nebenläufigkeit Behälter benutzen wollen.

Die JGL kommt nun mit dem Anspruch daher, eine recht umfangreiche und komfortable Behälterbibliothek zu sein. So bietet sie auch vielfältige Möglichkeiten an, mit Behältern umzugehen und sie zu verändern. Dafür ist die JGL auch sehr groß und der Klassenbaum der JGL wirkte auf mich anfänglich wie ein ‚Klassengestrüpp'. So gibt es Unmengen an Algorithmen, die in keiner Hierarchie stehen. Die Behälterhierarchie der JGL ist auch kein großes Vorbild: So stehen fachliche Behälter wie `Stack` oder `Queue` relativ gleichwertig neben technisch motivierten Behältern, wie `Array` oder `DList`. Ein Grundkonzept der JGL, die Trennung von Algorithmen und Datenstrukturen, ist gerade in Hinblick auf die Objektorientiertheit der Sprache Java nicht zu vertreten. Auch das Konzept der JGL-Iteratoren, nebenläufige Änderungen auf konzeptueller Ebene schlichtweg zu ignorieren, ist gerade bei einer Sprache wie Java, die von Anfang an die Möglichkeit von Threads besitzt, gefährlich.

Bei beiden Bibliotheken fehlt irgendeine Form der Typisierung von Behältern, sie können beliebig unterschiedliche Objekte beinhalten. Offensichtlich hat man sich mit der Gegebenheit abgefunden, daß es in keine Templates gibt. Zumindest die Konstrukteure der Collection-API schließen eine statische

Typisierung nicht aus, sofern Java in Zukunft mittels Templates diese Möglichkeit anbietet. Warum aber gerade bei der JGL die Typisierung der Behälter völlig unter den Tisch fällt, obwohl sie ein wichtiger Bestandteil der STL ist, ist mir schleierhaft.

Die *j*ConLib ist die einzige Bibliothek, die dauerhafte und stabile Cursor unterstützt. Ein weiterer Vorteil der *j*ConLib-Cursor ist, daß nicht wie bei den anderen Bibliotheken praktisch für jeden Behälter ein neuer Iterator programmiert werden muß, sondern daß die Behälter nur die Methoden für den Cursor anbieten, die sie aufgrund ihres Wissens über die Behälterstruktur selbst realisieren können. Das vermeidet Programmierfehler, andererseits erfordert das stabile Verhalten der *j*ConLib-Cursor eine Berücksichtigung von Cursorn bei Veränderungen des Behälters, was einigen zusätzlichen Programmieraufwand darstellt. Die Methoden zur Kommunikation mit den Cursor müssen auch nach außen hin sichtbar sein, allerdings sollte sie ein Benutzer der *j*ConLib nicht verwenden. Die JGL und die Collection-API entwickeln ihre Iteratoren an der technischen Implementierung, indem die Iteratoren für die verschiedenen technischen Implementierungen des Behälter unterschiedliche Verhalten in Form von anderen Schnittstellen besitzen. Allerdings haben alle Iteratoren durch ein entsprechendes Interface ein gemeinsames fachliches Verhalten. Etwas Ähnliches wie die Behälterfabrik der *j*ConLib sucht man bei der JGL vergebens. Zumindest die Collection API bietet eine Fabrikfunktionalität an, um Behälter mit zusätzlichen Eigenschaften zu versehen. Die JGL und die Collection-API ist leider auch nicht frei von der Methodik, daß allgemeine Methoden in spezialisierten Behälter nicht mehr vorkommen und deren Benutzung eine Exception provoziert, obwohl sich diese Art der Redefinition in Grenzen hält.

In dem folgenden Diagramm möchte ich alle Bibliotheken einmal gegenüberstellen und eine Bewertung der entsprechenden Teile der Bibliotheken untereinander durchführen. Bei ‚Behälterhierarchie' und ‚Cursor/Iterator' bewerte ich fachliche Modellierung und bei ‚Ergänzende Funktionen' und ‚Einbettung alter Strukturen' den Umfang der vorhanden Methoden.

	jConLib	java.util	JDK 1.2	JGL
Behälterhierarchie	√ (++)	-	√ (+)	√ (o)
Iteratoren/Cursor	√ (++)	(√) (1)	√ (+)	√ (o)
Typsicherheit	√	-	-	-
Ergänzende Funktionen	- (2)	-	-	√ (++)
Einbettung alter Strukturen	-	X (3)	√ (+)	√ (++)
Thread-Safe	√	√	√ (4)	√
Serialisierbar	√	√	√	√

Erläuterung: (1) Enumerator statt Iterator (2) Funktionen wie z.B. Sortierung durch sortierte Behälter nicht mehr nötig (3) Das java.util-Package ist eine alte Struktur (4) Optional durch Benutzer
Bewertung: (++) Sehr gut (+) Gut (o) Ausreichend. Ein Häkchen (√) bedeutet ‚ist vorhanden'

Abbildung 5.3 - Vergleich der Bibliotheken

Literaturverzeichnis

- **[Collection-API97]**
  Javasoft: „Java Collections API Overview, API Enhancement Summary & API Reference Documentation“, Sun Microsystems, 1997

- **[CollAPI-FAQ97]**
  Javasoft: „Java Collections API Design FAQ“, Sun Microsystems, 1997

- **[Flanagan97]**
  David Flanagan: „Java in a Nutshell“, Second Edition, O‘Reilly & Associates, May 1997

- **[GHJ94]**
  Erich Gamma, Reinhard Helm, Ralph Johnson, John Vlissides: „Design Patterns: Elements of Reusable Object-Oriented Software“, Addison Wesley, 1994

- **[GKZ93]**
  Guido Gryczan, Klaus Kilberth, Heinz Züllighoven: „Objektorientierte Anwendungsentwicklung“, Vieweg Verlag, 1993

- **[JWAM98]**
  Niels Fricke, Carola Lilienthal, Martin Lippert, Stefan Roock, Henning Wolf, Holger Bohlmann: „Dokumentation zum JWAM-Framework“, unter „http://swt-www.informatik.uni-hamburg.de/Software/JWAM“, Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, April 1998

- **[Lippert97]**
  Martin Lippert: „Konzeption und Realisierung eines GUI-Frameworks in Java nach der WAM-Metapher“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, November 1997

- **[Meyer97]**
  Bertrand Meyer: „Object-Oriented Software Construction“, 2nd Edition, Prentice Hall, 1997

- **[OW97]**
  Martin Odersky, Philip Wadler: „Pizza into Java: Translating theory into practice“, Revised from a paper presented In Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997

- **[Stroustrup91]**
  Bjarne Stroustrup: „The C++ Programming Language“, Second Edition, Addison Wesley, 1991

- **[Traub96]**
  Horst-Peter Traub: „Objektorientierte Behälterklassen-Bibliotheken: Konzepte, Entwurf und Implementation“, Mitteilung Nr. 252 des Fachbereiches Informatik, Universität Hamburg, Februar 1996

- **[WAM-Buch]**
  Heinz Züllighoven, et al.: „Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- und Material-Ansatz“, dpunkt.Verlag, August 1998

- **[Zühlke98]**
  Marco Zühlke: „Späte Erzeugung und Serialisierung in der BibV30“ (vorläufiger Titel), Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, vorraussichtliches Erscheinen: 1999