Eine Evaluierung des Java JDK 1.2 Collections Framework aus Sicht der Softwaretechnik

Mark Evered, Gisela Menger

Abteilung Rechnerstrukturen, Universität Ulm, D-89069 Ulm
{markev,gisela}@informatik.uni-ulm.de

Abstract. Sammlungen von Daten spielen eine wichtige Rolle bei fast jeder Art der Programmierung. Aus diesem Grunde bieten praktisch benutzte objektorientierte Sprachen Standard-Bibliotheken an, die Klassen für die Verwaltung von Objektsammlungen enthalten. Diese Bibliotheken sind besonders dafür kritisiert worden, daß sie softwaretechnische Prinzipien wie Geheimnisprinzip und Orthogonalität nicht hinreichend berücksichtigen. In diesem Beitrag evaluieren wir das neue 'Collections Framework' des 'Java Development Kit' 1.2, das durch die Definition der 'Core Collection Interfaces' einige Vorteile hinsichtlich der Verständlichkeit und Erweiterbarkeit solcher Bibliotheken bietet. Wir diskutieren sowohl Schwächen des Framework, die auf einer inkonsequenten Anwendung von Softwaretechnikprinzipien beruhen, als auch Probleme, die aus der Definition der Sprache Java selbst resultieren.

1 Einleitung

Bei fast jeder Art von Programmierung beschäftigen sich Programmierer in irgendeiner Weise mit Sammlungen von Daten. Viel Arbeitsaufwand ist immer wieder nötig, um die erforderliche Funktionalität und Leistungsfähigkeit von Datenstrukturen für eine bestimmte Anwendung zu realisieren. Wenn einem Programmierer die Möglichkeit gegeben wird, die notwendigen Datenstrukturen als fertige Klassen aus einer Klassenbibliothek zu entnehmen, und wenn diese Bibliothek leicht zu verstehen und zu benutzen ist, dann kann die Effizienz der Software-Herstellung wesentlich gesteigert werden. Solche Klassen sind ideale Kandidaten für die Wiederverwendung von Software.

Die konsequente Benutzung von Sammlungsklassen kann zu einer höheren Abstraktionsebene der Programmierung führen, wie sie mit den sogenannten 'very high level'-Sprachen [21] angestrebt wurde. Programmierer (und Wartungs-programmierer) können Sammlungen dann als selbstverständliche Grundbausteine betrachten und sich auf deren Benutzung, anstatt auf deren Implementierung konzentrieren. Damit dies möglich ist, müssen die Sammlungstypen in allen Programmen gleich aussehen, d.h., sie müssen standardisiert und als 'zur Sprache

gehörig' angesehen werden.

Die Notwendigkeit der Standardisierung und Integration hat dazu geführt, daß für die meisten weit verbreiteten objektorientierten Sprachen inzwischen ein 'Collections Framework' definiert worden ist. Smalltalk-80 [10] schließt Klassen für Sammlungen in seine vordefinierte Systemhierarchie ein. Eiffel [17] besitzt eine Standardbibiliothek von Containerklassen [18]. Viele verschiedene Bibliotheken sind für C++ [23] entwickelt worden, und inzwischen wurde die 'Standard Template Library' (STL) [20] als Standard definiert. Bis zur JDK Version 1.1 hat die Sprache Java [11] nur einige Typen, wie z.B. 'Vector' und 'Hashtable', hilfsweise angeboten. Aber in der Version 1.2, die bald freigegeben werden soll, wird ein umfassendes Collections Framework [12] enthalten sein.

Einfache Benutzbarkeit und Abstraktion von Implementierungsdetails sind wohlbekannte Ziele der Softwaretechnik. Existierende Bibliotheken sind gerade hinsichtlich dieser Aspekte kritisiert und als in der Praxis schwierig benutzbar beurteilt worden. Die Designer des Java Framework haben die Gelegenheit genutzt, aus den Erfahrungen mit anderen Bibliotheken zu lernen. Zudem verfügt Java über sprachliche Konstrukte, die eine bessere softwaretechnik-orientierte Unterstützung für die Definition von Sammlungstypen erlauben als andere objektorientierte Sprachen.

Im folgenden Abschnitt formulieren wir Anforderungen an eine Bibliothek für Sammlungstypen (im folgenden der Einfachheit halber: Bibliothek), die flexibel aber auch einfach zu benutzen sein soll. Wir skizzieren Probleme vergleichbarer Bibliotheken. Im dritten Abschnitt beschreiben wir die Vorteile des Java Collections Framework, weisen aber auch auf Schwächen hin, die sowohl mit dem Entwurf des Framework als auch mit Spracheigenschaften zusammenhängen.

2 Anforderungen der Softwaretechnik

Es hat sich als außerordentlich schwierig erwiesen, Bibliotheken zu definieren, die dem breiten Spektrum von Funktionalitäts- und Effizienzanforderungen an Sammlungen aus Anwendungssicht gerecht werden, ohne die Benutzung so kompliziert zu machen, daß sie von Programmierern nicht mehr als hilfreich betrachtet wird. Existierende Bibliotheken sind kritisiert worden als

- organisiert aus Implementierungs- statt aus Benutzungssicht [3]
- schwer zu verstehen und benutzen [22]
- inhärent unskalierbar [2]

In [19] wird aufgrund dieser Probleme eine Liste von Anforderungen aus Benutzersicht formuliert:

- Sammlungstypen sollen klar und verständlich organisiert sein. Wenn dies nicht der Fall ist, werden sie mehr Mühe kosten als sie einsparen.
- Die Spezifikation eines Sammlungstyps soll seine Funktionalität möglichst vollständig beschreiben, aber auf Implementierungsdetails verzichten. Es soll daher möglich sein, mehrere äquivalente Implementierungen für einen Sammlungstyp zu haben und sie ohne Beeinträchtigung der Korrektheit eines

benutzenden Programms einfach auszutauschen.

- Ähnliche Sammlungstypen sollen ähnliche Schnittstellen haben. Dies ist beispielsweise durch eine gut strukturierte Typhierarchie zu erreichen.
- Die Spezifikation eines Untertyps soll der Spezifikation eines Obertyps nicht widersprechen, sondern lediglich präzisieren. (Prinzip des 'behavioral subtyping' [14]).
- Orthogonale semantische Eigenschaften von Sammlungen, wie z.B. Ordnung oder Zugriff über einen Schlüssel, sollen in allen Kombinationen verfügbar sein.
- Anwendungsspezifische Beschränkungen, wie der Elementtyp einer Sammlung oder das Kriterium für eine automatische Sortierung, sollen auf eine flexible und statisch überprüfbare Weise angegeben werden können.
- Operationen, die mehr als eine Sammlungen betreffen, wie Vergleich oder Vereinigung, sollen in konsistenter und symmetrischer Weise vorhanden sein.

Es ist hier nicht unser Ziel, eine vollständige Evaluierung bestehender Bibliotheken anhand dieser Anforderungen vorzunehmen. Wir skizzieren lediglich die wichtigsten und häufigsten Probleme solcher Bibliotheken als Vergleichsgrundlage für das Java Framework. Für eine detailiertere Darstellung siehe [19].

Das Hauptproblem aus softwaretechnischer Sicht ist ohne Zweifel die Vermischung von Funktionalitätsaspekten mit Implementierungsaspekten. In der STL ist beispielsweise die Klasse 'List' keine abstrakte Listendefinition, sondern eine ganz bestimmte Listenimplementierung, nämlich eine doppelt verkettete Liste. Die Smalltalk-Bibliothek enthält Klassen wie 'LinkedList' und 'ArrayedCollection', die eindeutig auf das Implementierungskonzept hinweisen. Manche Klassifikations-kriterien auf den oberen Ebenen der Eiffel-Hierarchie (wie etwa 'Traversable') beziehen sich nicht auf Funktionalität, sondern auf Eigenschaften einer Implementierung. Ein Programmierer, der einen Sammlungstyp benutzen will, interessiert sich bei der Programmentwicklung in erster Linie für die erforderliche Funktionalität und soll sich nicht mit Implementierungsaspekten beschäftigen müssen:

```
Set s;              // Variable eines abstrakten Mengentyps
s=new SimpleSet();  // Erzeuge Menge mit einfacher Implementierung
...                 // Ab hier kein Kenntnis der Implementierung
```

Auf der anderen Seite muß es möglich sein, daß er später die Implementierung, die für die Prototypversion hinreichend war, austauschen kann, um das Programm gezielt zu optimieren, ohne Auswirkungen auf die Korrektheit seines Programms befürchten zu müssen ('first working, then fast'):

```
s=new SuperHashMitAutoIrgendwasSet();  // nur diese Zeile geändert
```

Um verschiedenen Anwendungsanforderungen gerecht zu werden, sollte es möglichst viele Implementierungen für jeden Typ geben, aber sie sollten nicht Teil der abstrakten Typhierarchie sein. Das Durcheinander von abstrakten Typen und Implementierungen

in Bibliotheken ist teilweise auf das Konzept der Klasse zurückzuführen und teilweise auf die Benutzung von Vererbung. Eine Klasse definiert sowohl einen Typ als auch eine Implementierung für diesen Typ. Wenn eine einzige Vererbungshierarchie für Modellierung, Subtyping und Code-Wiederverwendung benutzt wird, sind Unverträglichkeiten und Kompromisse fast unvermeidlich [4, 15, 6].

Ein weiteres Problem besteht in der Zuordnung von Methodennamen zum Verhalten einer Methode. Die drei sequentiellen Klassen der STL haben ähnliche Schnittstellen, aber weil sie nicht in eine Subtyping-Hierarchie eingeordnet sind, ist nicht leicht zu erkennen, wo sie sich, wenn überhaupt, in ihrem Verhalten unterscheiden. Die Klassen der Eiffel-Bibliothek sind zwar hierarchisch geordnet, aber das Verhalten einer Methode in einer Unterklasse kann dem Verhalten widersprechen, das für die Oberklasse definiert wurde. In Smalltalk-80 unterscheidet sich die 'Verhaltenshierarchie' an vielen Stellen von der Vererbungshierarchie [3].

Ein drittes Problem ist die abschreckend große Anzahl von Sammlungsklassen in manchen Bibliotheken (beispielsweise etwa 80 in Eiffel). Dies ergibt sich teilweise aus der obengenannten Vermischung von Funktionalität und Implementierung, aber auch wenn man davon absieht, können verschiedene Aspekte, wie Ordnung, Behandlung von Duplikaten und Zugriffsmöglichkeiten, zu einer großen Zahl von Typen führen. Wenn andererseits einige der Möglichkeiten nicht unterstützt werden, kann genau das fehlen, was für eine bestimmte Anwendung gebraucht wird. So bietet die C++ STL zum Beispiel keine Möglichkeit an, eine Sammlung ohne sichtbare Ordnung zu benutzen. Eine Lösung dieses Dilemmas kann erreicht werden, wenn ein wohlbekanntes Prinzip der Softwaretechnik angewendet wird: die Identifikation und Trennung orthogonaler Konzepte. So können Ordnungskriterien und die Behandlung von Duplikaten als orthogonale Aspekte von Sammlungen angesehen und in beliebigen Kombinationen angeboten werden. Auf diese Weise kann eine Bibliothek potentiell immer noch groß werden, aber trotzdem konzeptuell einfach bleiben.

Ein letztes Problem betrifft die Flexibilität von Sammlungstypen. Oft wird beschränkte Generizität benutzt, um sicherzustellen, daß ein Elementtyp eine bestimmte Methode besitzt (z.B. eine Methode 'less', die benutzt wird, um die Elemente in einer automatisch geordneten Liste korrekt einzufügen). Das bedeutet aber, daß solche Listen für Objekte, die diese Methode nicht besitzen, auch nicht benutzt werden können, und ebenso, daß zwei Listen mit demselben Elementtyp nicht auf zwei verschiedene Weisen geordnet werden können. Eindeutige Regeln, die für die Organisation einer Sammlung benötigt werden, gehören logisch zum *Sammlungstyp* und nicht zum *Elementtyp.* Sie sollten deshalb offengehalten und für jede konkrete Sammlung neu angegeben werden können [8].

3 Das JDK 1.2 Collections Framework

3.1 Struktur

Bis zur Version 1.1 hat das 'Java Development Kit' nur sporadische Unterstützung für allgemein verwendbare Objektsammlungen geboten. Neben Arrays gibt es nur die Klassen 'Vector' und 'Hashtable. In der Version 1.2 will Javasoft nun mit dem neuen

Package 'java.util.Collection' ein umfassendes Framework für Sammlungen zur Verfügung stellen. Den Kern des Framework bilden sechs 'Interface'-Definitionen, die 'Core Collection Interfaces':

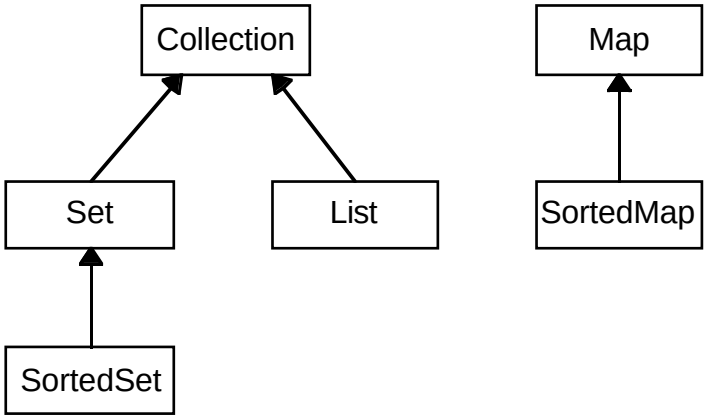

Fig. 1. Die JDK 1.2 'Core Collection Interfaces'

'Collection' ist ein allgemeiner Typ mit Methoden wie 'add', 'clear', 'remove' and 'size'. Bei 'List' hat der Benutzer die Kontrolle über die Anordnung der Elemente, und darum gibt es zusätzlich Methoden wie 'get' und 'set', um ein Element an einer bestimmten Position zu selektieren oder einzufügen. 'Set' steht für das mathematische Mengenmodell, und 'SortedSet' ist ein 'Set', das automatisch geordnet wird. 'Map' ist eine Sammlung von Paaren, wobei jedes Paar aus einem (eindeutigen) Schlüssel und dem zugehörigen Objekt besteht. Das 'Map'-Interface hat Methoden wie 'put', 'get', 'clear' und 'size'. 'SortedMap' sortiert die Elemente automatisch nach dem spezifizierten Schlüssel.

Das Framework bietet verschiedene Klassen an, die diese Interfaces implementieren. Speziell gibt es eine Implementierung für 'Set' ('HashSet'), eine für 'SortedSet' ('TreeSet'), drei für 'List' ('ArrayList', 'LinkedList' und 'Vector'), zwei für 'Map' ('HashMap' und 'Hashtable') und eine für 'SortedMap' ('TreeMap'). Ferner gibt es abstrakte Klassen, die es erleichtern, neue Implementierungen für die Interfaces zu schreiben. Weitere charakteristische Aspekte des Collections Framework sind:

- Iteratoren, die das sequentielle Abarbeiten der Elemente einer Sammlung erleichtern,
- spezielle Umwandlungssoperationen, um Arrays bzw. Maps in einen Collection-Typ zu überführen und umgekehrt,
- spezielle 'Konstruktoren', die synchronisierte oder unveränderbare Versionen einer Sammlung zur Verfügung stellen,
- allgemeine Algorithmen (z.B. Sortieren), die als 'static'-Methoden der besonderen Klasse 'Collections' angeboten werden.

3.2. Vorteile des Java-Ansatzes

Mit seiner einfachen und übersichtlichen Struktur stellt das Java Collection Framework zweifellos einen wichtigen Fortschritt in der Benutzbarkeit von Sammlungstypen dar. Durch die Verwendung von Interfaces ist eine klare Trennung von abstrakter Funktionalität und Implementierung angestrebt, und durch die Unterscheidung der Interface-Hierarchie von der Klassenhierarchie ist es möglich, Aspekte der Typmodellierung weitgehend von Aspekten der Code-Wiederverwendung zu trennen. Die Modellierung der Typen in der Interface-Hierarchie entspricht dem Prinzip des 'behavioral subtyping' mit abstrakteren Typen auf den oberen Ebenen und immer präziser spezifizierten Typen auf den unteren. Die abstrakten Klassen erlauben, die angebotenen Implementierungen relativ leicht durch weitere zu ergänzen.

Das Ordnungskriterium für automatisch sortierte Sammlungen wird flexibel gehalten. Es kann entweder durch die 'natürliche Ordnung' des Elementtyps (d.h. die Methode 'compareTo') bestimmt sein oder durch ein 'Comparator'-Objekt, das dem Konstruktor der Sammlung übergeben wird. z.B.

```
class MatrNrOrderer implements Comparator {
  public int compare(Object o1, Object o2) {
    // Code um nach Matr.-Nr. zu vergleichen
  }
}
...
SortedSet s = new TreeSet(new MatrNrOrderer());
// Erzeuge eine geordnete Menge mit der Implementierung 'TreeSet'
// und automatisch geordnet nach Matrikelnummer
```

Es gibt auch Methoden für den Umgang mit mehreren Sammlungen, wie z.B. 'equal', 'clone' und 'addAll'. Sie werden nicht nur zahlreicher, sondern auch wesentlich konsistenter angeboten als in anderen Sammlungsbibliotheken.

Die speziellen Mechanismen, um synchronisierte und unveränderbare Sammlungen zu erzeugen, verkomplizieren das Framework zwar, bieten aber Möglichkeiten, die in der Praxis wichtig sind, von den meisten gängigen Bibliotheken jedoch nicht unterstützt werden.

3.3. Kritik

Trotz dieser Vorteile gegenüber anderen Bibliotheken gibt es auch Schwächen in der Definition des Java Collections Framework. Wir beschränken uns hier auf die wichtigsten Kritikpunkte aus softwaretechnischer Sicht: die unnötige Vermischung von Funktionalität mit Implementierungsdetails und eine unzureichende Orthogonalität von Sammlungseigenschaften.

Abstraktion. Damit verschiedene Implementierungen für einen abstrakten Typ wirklich austauschbar sind, muß der Typ so genau wie möglich spezifiziert sein. Während für allgemeine Typen wie 'Collection' allgemeine semantische Beschreibungen ausreichen, muß die Funktionalität von Typen, die implementiert und instantiiert werden sollen, möglichst vollständig beschrieben werden. Wenn den

Implementierern 'funktionale Freiheiten' gelassen werden, muß man davon ausgehen, daß verschiedene Implementierungen sich im Verhalten unterscheiden werden.

Die JDK 1.2 'Core Collection Interfaces' lassen einige Fragen offen und sind deshalb nicht wirklich bindende (und daher zuverlässige) Typspezifikationen. Dies wird deutlich in Aussagen wie z.B. zu Listen:

- "Duplicates are generally permitted"
- "The caller generally has precise control over the position of each element in the list"
- "they typically allow multiple null elements if they allow null elements at all".

Noch deutlicher wird es in der grundsätzlichen Aussage, daß manche Implementierungen einschränken können, welche Elemente eingefügt werden dürfen, z.B. nur Elemente eines bestimmten Typs, nur 'non-null'-Elemente oder nur Elemente, die irgendein Prädikat erfüllen. In der Praxis hat dies zur Folge, daß ein Programmierer, der beispielsweise eine 'List'-Variable benutzt, genau wissen muß, welche Implementierung dahinter steckt. Das ist ein klarer Verstoß gegen das Geheimnisprinzip.

Es ist auch erlaubt, Implementierungen zu schreiben, die nicht alle Interface-Methoden realisieren, sondern stattdessen eine 'UnsupportedOperationException' auslösen, wenn eine bestimmte Methode aufgerufen wird. Dies gilt für Methoden, die als 'optional operation' kommentiert sind. Die 'add' Methode für Sammlungen ist eine solche Methode. Der Hauptgrund, 'add' optional zu machen, ist, daß die Sammlung 'unmodifiable' sein könnte. Die praktische Folge ist aber, daß ein Programmierer, der eine 'List'-Variable benutzt, nicht einmal davon ausgehen kann, daß ein 'add'-Aufruf funktionieren wird. Die Interfaces sind daher eigentlich keine Typdefinitionen im Sinne statischer Überprüfbarkeit.

Implementierungen müssen nicht alle Interface-Methoden realisieren, aber sie dürfen Methoden ergänzen, die zu einer bestimmte Art der internen Repräsention besonders gut passen. So ergänzt die Array-Implementierung für 'List' u.a. eine Methode 'trimToSize', während die Linked-List-Implementierung z.B. zusätzlich eine Methode 'addFirst' anbietet. Damit werden Anwendungsprogrammierer geradezu ermuntert, das Geheimnisprinzip und den Appell "Program to an interface, not an implementation" [9] zu vergessen. z.B.:

```
List l;
...
((LinkedList) l).addFirst(x);
// Hier optimiert ein Programmierer, weil er 'weiß', daß eine
// verketterte Liste benutzt wird. Es entsteht ein Laufzeit-
// fehler, falls dies irgendwann nicht der Fall sein sollte.
```

Wenn man die Option offenhalten wollen, die Implementierung zu wechseln, ohne das Programm einer vollständigen Revision zu unterziehen, dürfen sie solche Methoden nicht benutzen.

Orthogonalität. Bei den JDK 'Maps' wird der Schlüssel als getrennter Parameter mit dem zugehörigen Objekt übergeben. Der Schlüssel dient dazu, ein Objekt eindeutig zu identifizieren und (in einer guten Implementierung) schnell wiederfinden zu können. In Datenverarbeitungsanwendungen ist der Schlüssel aber normalerweise Teil des Objekts, wie etwa die Matrikelnummer in einem Studentenobjekt. In diesem Fall erzwingen 'Maps', daß der Schlüssel zweimal gespeichert wird. 'Maps' bieten die Funktionalität von 'Sets' mit der zusätzlichen Möglichkeit des effizienten Suchens. Der schnelle Zugriff auf ein Objekt anhand eines Schlüsselfeldes ist aber ebenso wichtig für Listen und jeden anderen Sammlungstyp. Es ist - neben Ordnung und der Handhabung von Duplikaten - eine weitere orthogonale Eigenschaft eines Sammlungstyps.

Daß das Java Collections Framework dieser Orthogonalität nicht gerecht wird, läßt sich an der Ähnlichkeit von 'SortedSets' und 'SortedMaps' erkennen. Die Tatsache, daß 'Maps' keine 'Collections' sind, macht es notwenig, Umwandlungsoperationen von 'Maps' zu 'Collections' anzubieten. Eine flexible Alternative zu 'Maps' wäre gewesen, für eine Sammlung eine Operation angeben zu können, die den Schlüsselwert eines Elementes bestimmt, ähnlich wie die Operation in einem 'Comparator' die Ordnung bestimmt.

Die Ordnungs- und Duplikat-Eigenschaften von Sammlungen werden im Collections Framework ebenfalls nicht orthogonal behandelt. Ein 'OrderedSet' ist automatisch geordnet und ohne Duplikate. Ein 'List' ist benutzergeordnet und erlaubt ('generally') Duplikate. Es gibt aber beispielsweise keinen Sammlungstyp, der ungeordnet ist und Duplikate erlaubt (d.h. ein Bag). Für einen Benutzer ist schwer einzusehen, warum einige Kombinationen unterstützt werden und andere nicht. Es wäre durchaus möglich gewesen, das mehrfache Subtyping von Interfaces zu nutzen, um die inhärente Orthogonalität von Sammlungseigenschaften zu modellieren (siehe [5]).

3.4. Probleme der Sprache

Die obengenannten Mängel sind auf den Entwurf des Collections Framework zurückzuführen. Weitere Probleme resultieren aus Eigenschaften der Sprache Java selbst.

Java besitzt keinen Mechanismus für eine statisch überprüfbare Beschränkung des Elementtyps einer Sammlung. Wenn ein Programmierer eine Liste von 'Person'-Objekten braucht, kann erst zur Laufzeit sichergestellt werden, daß kein Objekt eines anderen Typs eingefügt wird. Obwohl die Möglichkeit vorgesehen wird, bietet keine der vordefinierten Implementierungen eine Überprüfung des Elementtyps beim Einfügen an. Es bleibt dem Benutzer einer Sammlung überlassen, den Typ zu rekonstruieren, wenn ein Objekt aus einer Sammlung entnommen wird. Inzwischen sind eine Reihe von Vorschlägen [16, 1, 7] veröffentlicht worden, wie statisch überprüfbare parameterisierte Typen in Java integriert werden könnten, aber es bleibt abzuwarten, ob einer davon übernommen wird. Ein verwandtes Problem ist, daß Sammlungen von Elementen eines primitiven Typs nur umständlich über 'wrapper'-Klassen möglich sind.

Ein zweites Problem entsteht aus der Tatsache, daß Interfaces keine Konstruktoren

enthalten dürfen. Dies bedeutet, daß in Interfaces, die als abstrakte Typspezifikationen dienen sollen, keine Konstruktoren spezifiziert werden können. Aus diesem Grund werden in den Interfaces 'Vorgaben' für Konstruktoren lediglich in Form von Kommentaren gemacht, wie z.B.:

> "All general-purpose Collection implementation classes should provide two 'standard' constructors ...".

Ein drittes Problem betrifft die angestrebte Trennung der Typhierarchie (dargestellt durch Interfaces) von der Code-Wiederverwendungshierarchie (dargestellt durch Klassen). Klassen sind in Java, wie in den meisten objektorientierten Sprachen, nicht nur Implementierungen, sondern auch Typen. Dies kann zu unerwarteten Konsequenzen führen. Angenommen, wir wollten eine einfache 'Set'-Implementierung herstellen, indem wir eine Klasse 'LinkedSet' definieren, die Code von 'LinkedList' erbt, einige Methoden überschreibt und einige Methoden hinzufügt, um die 'Set'-Semantik zu erfüllen. Dann wäre es nach dem Java-Typsystem zugelassen, eine 'List'-Variable auf ein 'LinkedSet'-Objekt zeigen zu lassen, obwohl dieses Objekt die Semantik einer Liste gar nicht gewährleistet. Dieses Problem kann allerdings nur durch eine wirklich radikale Trennung von Subtyping und Code-Wiederverwendung (wie etwa in der Forschungssprache Theta des MIT [13]) gelöst werden.

4 Zusammenfassung

Sammlungen von Objekten spielen eine wichtige Rolle in der Programmierung von objektorientierten Systemen. Die Produktivität eines Programmierers kann wesentlich gesteigert werden, wenn er fertige Typen aus einer Standardbibliothek verwendet, anstatt immer wieder das Rad neu zu erfinden. Dies gilt allerdings nur, wenn die Bibliothek so gut strukturiert und verständlich ist, daß der Programmierer sie schnell überblicken und benutzen kann. Notwendig hierfür sind wohlbekannte aber oft vernachlässigte Prinzipien der Softwaretechnik wie das Geheimnisprinzip und die Orthogonalität von Konzepten.

Die Verwendung von Interface-Typen im Collections Framework des 'Java Development Kit' 1.2 stellt einen wichtigen Schritt in diese Richtung dar. Das Framework ist einfach, übersichtlich und gut erweiterbar in bezug auf Implementierungen für die abstrakt definierten Typen. Auch die flexible Handhabung von Ordnungskriterien erleichtert die Benutzung der Sammlungen für komplexere Anwendungen.

Allerdings sind weder das Geheimnisprinzip noch die Orthogonalität so konsequent realisiert, wie es möglich gewesen wäre. Den Implementierern von Sammlungstypen werden Freiheiten hinsichtlich der geforderten Funktionalität gelassen, so daß Bibliotheksbenutzer letztlich doch wissen müssen, um welche konkrete Implementierung es sich bei einem Typ jeweils handelt.

'Maps' sind eine Kombination von 'Set'-Semantik und effizienten Suchmechanismen. Effizientes Suchen sollte aber sinnvollerweise für alle Sammlungstypen angeboten werden. Durch multiples Subtyping in der Interface-Hierarchie wäre die Orthogonalität von Sammlungseigenschaften besser unterstützt worden.

Einige Schwächen des Framework beruhen auf Eigenschaften der Sprache. Das schwerwiegendste Problem aus softwaretechnischer Sicht ist, daß Typfehler bei den Elementen einer Sammlung nicht statisch entdeckt werden können. Ein weiteres Problem ist, daß Konstruktoren nicht in einem Interface-Typ spezifiziert werden können.

References

1. O. Agesen, S.N. Freund and J.C. Mitchell "Adding Type Parameterization to the Java Language", in Proc. OOPSLA '97, pp. 49-65, 1997.
2. D. Batory, V. Singhal, M. Sirkin and J. Thomas "Scalable Software Libraries", in Proc. SIGSOFT '93, Los Angeles, CA, pp. 191-199, ACM, 1993.
3. W. R. Cook "Interfaces and Specifications for the Smalltalk-80 Collection Classes", in Proc. OOPSLA '92, in ACM SIGPLAN Notices, 27, 10, pp. 1-15, 1992.
4. W. R. Cook, W. L. Hill and P. S. Canning "Inheritance is Not Subtyping", in Proc. 17th ACM Symposium on Principles of Programming Languages, San Francisco CA, pp. 125-135, 1990.
5. M. Evered, J. L. Keedy, G. Menger and A. Schmolitzky "A Useable Collection Framework for Java", in Proc. 16th IASTED Conf. on Applied Informatics, Garmisch-Patenkirchen, 1998.
6. M. Evered, J. L. Keedy, A. Schmolitzky and G. Menger "How Well Do Inheritance Mechanisms Support Inheritance Concepts?", in Proc. Joint Modular Languages Conference (JMLC) '97, Linz, Austria, in Lecture Notes in Computer Science 1204, pp. 252-266, 1997.
7. M. Evered, J. L. Keedy, A. Schmolitzky and G. Menger "Genja: A New Proposal for Genericity in Java", in *Proc. Conference on Technology of Object-Oriented Languages and Systems*, 25, Melbourne, pp. 181-193, 1997.
8. M. Evered "Unconstraining Genericity", in *Proc. Conference on Technology of Object-Oriented Languages and Systems*, 24, Beijing, pp. 423-431, 1997.
9. E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA.: Addison-Wesley Publishing Company, 1995.
10. A. Goldberg and D. Robson, Smalltalk-80 The Language, Reading, MA: Addison-Wesley, 1989.
11. J. Gosling, B. Joy and G. Steele, The Java Language Specification, Reading, MA: Addison-Wesley, 1996.
12. Javasoft WWW Page (JDK 1.2 Beta 3) http://www.javasoft.com/ products/jdk/1.2/docs/ guide/collections/index.html, 1998.
13. B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson and A. C. Myers "Theta Reference Manual", Report Number 88, MIT Lab. for Computer Science, Cambridge, MA, 1995.

14. B. H. Liskov, J. M. Wing: “A Behavioral Notion of Subtyping”, *ACM Transactions on Programming Languages and Systems,* Vol. 16:6, pp. 1811-1841, 1994.

15. W. LaLonde and J. Pugh "Subclassing ≠ subtyping ≠ Is-a", in Journal of Object-Oriented Programming, 1991, pp. 57-62.

16. A. C. Myers, J. Bank, B. Liskov: "Parameterized Types for Java", *Proc. Symposium on Principles of Programming Languages '97*, Paris, France, pp.132-145, 1997.

17. B. Meyer, Eiffel: The Language, New York: Prentice-Hall, 1992.

18. B. Meyer "Reusable Software, The Base Object-Oriented Component Libraries (Version 3.2.2)", Report Number TR-EI-44/LI, ISE, Santa Barbara, 1994.

19. G. Menger, J. L. Keedy, M. Evered and A. Schmolitzky "Collection Types and Implementations in Object-Oriented Software Libraries", in *Proc. Conference on Technology of Object-Oriented Languages and Systems*, 26, Santa Barbara, 1998 (to appear).

20. D. R. Musser and A. Saine, STL Tutorial and Reference Guide, C++ Programming with the Standard Template Library, Reading, MA: Addison-Wesley, 1996.

21. J. T. Schwartz "Automatic Data Structure Choice in a Language of Very High Level", Comm. ACM, 18, 12 pp. 722-728, 1975.

22. C. Szypersky, S. Omohundro and S. Murer "Engineering a Programming Language: The Type and Class System of Sather", in Programming Languages and System Architectures, ed. J. Gutknecht, Springer-Verlag, pp. 208-227, 1993.

23. Stroustrup, B. (1989) "The C++ Programming Language", Addison-Wesley, Bonn.