

Interfaces and Specifications for the Smalltalk-80 Collection Classes

William R. Cook

Apple Computer
20525 Mariani Avenue
Cupertino CA 95014
William@AppleLink.Apple.Com

Abstract: The hierarchy of interfaces implicit in the Smalltalk-80 collection class library is computed and analyzed. The interface hierarchy is independent of the inheritance hierarchy because methods are frequently deleted by subclasses, and because unrelated classes sometimes implement the same messages. Specifications of the interfaces are developed, revealing subtle relationships among messages and their methods. The specifications help identify several kinds of problems in the library: inherited methods that violate the subclass invariant; methods that have the same name but unrelated behaviors; methods that have the same (or related) behavior but different names. This exercise demonstrates the utility of interfaces and specifications, and suggests improvements to the collection class library structure.

1 Introduction

The Smalltalk-80 class library [GR83] is a significant example of object-oriented design. The library is the product of at least ten years of development and evolution. It encompasses classes for data structures, graphics and window management, program development and compilation. This breadth and depth makes Smalltalk-80 an essential testing ground for new theories about object-oriented programming.

This paper analyzes the Smalltalk-80 collection class library. These classes, which support a wide variety of sophisticated data structures, are organized

in the Smalltalk system by *inheritance*. The inheritance structure is used to document the classes and is presented to users as an aid to understanding the library. Yet there is growing consensus that inheritance is a “producer’s mechanism” [Meyer91] that has little to do with client’s use of classes.

An alternative organization of the collection classes is developed by examining the *interfaces* supported by the collection objects [CCHO89, JF88]. A program for extracting *protocols*, or sets of message names, directly from the Smalltalk system is described. The algorithm is complicated by subclasses that *cancel* or *delete* methods that they would otherwise inherit, and by complex dependencies among partially implemented methods in abstract classes. If the subclass does not implement appropriate methods, then entire groups of methods defined in the abstract class are not supported. Taking these complications into account, useful interface information is extracted automatically from the Smalltalk library.

An *interface hierarchy* is a logical organization of the interfaces of each class in a library. The interface hierarchy is a partial order that factors out shared interfaces, using the notion of *conformance* (or *subtyping*) for interfaces [Cardelli84, CM89, BHJLC86]. An algorithm for computing the interface hierarchy of a Smalltalk class library is described. When applied to the Smalltalk-80 Collection classes, the program produces a descriptive picture of the sharing of messages among classes. Even for a language like Smalltalk, which supports only single inheritance, the interface hierarchy is a complex graph with multiple sharing of partial interfaces.

This is because of canceled methods, and also unrelated classes that support the same messages.

A number of errors in the implementation of the collection classes are immediately apparent from the interface hierarchy. These are easily corrected, but the diagram also raises questions that cannot be answered by examining interfaces along. To address these, the behavior of methods must be analyzed.

Behavioral specification tools based on pre and post conditions have also been adapted for use in object-oriented programming [America91, LW90]. Specifications have also been used in the design of the Eiffel libraries [Meyer91]. America's techniques are used to develop specifications for the classes in the Smalltalk library. Several interesting issues arise while discussing the specifications.

- In some cases, methods are being inherited that should be canceled because they violate subclass invariants.

- In some cases the classes are using a message for such different purposes that a generalized specification cannot be found. In this case it is suggested that the messages be given different names to recognize their disparate specifications.

- Two different messages occasionally have the same specification. In this case the names should be unified.

- It is useful to break the specifications down into individual messages, and analyze the relationships among the family of different implementations of the message. Interfaces can then be formed by selecting an appropriate message specification from each message family.

The inheritance hierarchy and specifications suggest some corrections to the collection classes. The effect of these changes are illustrated in a new interface hierarchy. This hierarchy contrasts strongly with the more traditional presentation of the collection classes through the inheritance hierarchy.

Section 2 discusses the problem of extracting protocols from class implementations and organizing it into an interface hierarchy. Section 3 presents specifications for the protocols that are identified in Section 2. Section 4 presents the conformance relationships among the class and method specifications. Section 5 contains an interface hierarchy that incorporates the corrections proposed in previous sections.

Section 6 compares the interface hierarchy with the traditional inheritance hierarchy. Section 7 presents conclusions.

2 Smalltalk Interface Hierarchy

2.1 Extracting Class Interfaces

This section develops a procedure for automatic extraction of interfaces from a Smalltalk system. An *interface* is a description of the legal operations on an object. The level of description may vary from names of supported messages to behavioral specification. For Smalltalk [GR83], a commonly used form of interface is the *protocol*, a simple set of operation names. The operation names are called *selectors*, which list the colon-terminated argument keyword names of the message. The instances of a given class share the same protocol [JF88].

A rough cut at extracting the protocol of objects in a class may be computed by forming the union of all the selectors for methods defined by the class and its superclasses. However, a subclass method may cancel a method from its superclass by redefining it to return a special error indicating that the method should not be used. Thus a simple union of method selectors is not an accurate representation of the legal operations on the object. For example, the class `Interval` inherits from `SequenceableCollection`, but intervals are constants and cannot be changed, unlike most collections. Thus it is necessary to eliminate all inherited methods that attempt to modify the collection. The `add:` and `remove:` messages of class `Interval` are a good illustration:

add: newObject

"Provide an error notification that adding to an Interval is not allowed."
self shouldNotImplement

remove: newObject

"Provide an error notification that removing an element from an Interval is not allowed."
self error: 'elements cannot be removed from an Interval'

A Smalltalk-80 program can compute the protocol of a class by examining the compiled representation of methods to determine if they are similar to the

ones listed above. These methods, and any private methods, are deleted from the protocol. The following program computes the protocol of a class. The method `isCancellingMethod` determines if a method returns an error immediately.

```
!Behavior methodsFor: 'accessing' !
```

protocol

```
p := Set new.
self isVariable ifTrue: [
  p addAll: #(at: at:put: size) ].
self collectProtocol: p.
```

collectProtocol: p

```
parent := self superclass.
parent notNil ifTrue: [
  parent collectProtocol: p ]
self selectors do: [ :s |
  ((self compiledMethodAt: s)
   isCancellingMethod
  | #private = (self organization
   categoryOfElement: s))
  ifTrue: [
    aSet remove: s ifAbsent: [] ]
  ifFalse: [
    aSet add: s ]]].
```

Some additional checks, like the one for variable classes, may be necessary in different versions of Smalltalk to handle special cases, like primitives, and operations on the class `Object`.

A final complication arises from the conventional use of abstract classes [JF88]. These classes include methods that return an error indicating they should be implemented by subclasses. These key methods are identified in the same way as the cancelling methods discussed above. But other methods typically depend upon the key subclass responsibility methods. If the subclass does not implement a key method, then all of the methods depending upon it must be removed from the protocol. The following methods taken from class `Collection` illustrate this situation.

add: newObject

```
"Include newObject as one of the receiver's
elements. Answer newObject. This mes-
sage should not be sent to instances of
subclasses of ArrayedCollection."
self subclassResponsibility
```

addAll: aCollection

```
"Include all the elements of aCollection as
the receiver's elements."
aCollection do: [ :each | self add: each ].
^aCollection
```

In the results presented here, this kind of message dependencies were resolved by manual examination. But because they are messages sent to self, references to subclass responsibility methods could be identified statically. The protocol method defined above could then be extended to eliminate from the protocol any method that sends messages to self that are not in the protocol. Dependencies on unimplemented methods occur surprisingly often in the Smalltalk collection library and are a significant source of confusion in trying to understand collection behavior.

2.2 The Protocol Hierarchy

The protocol hierarchy arises from the partial order of protocols by the *conformance*, or *subtype*, relationship [Cardelli84, CM89, BHJLC86]. Interface B conforms to interface A if every object that satisfies B also satisfies A. This means that the set of objects satisfying B is a subset of the set satisfying A. However, the interface B, viewed as a set of operations, is typically larger (more specific) than the interface A.

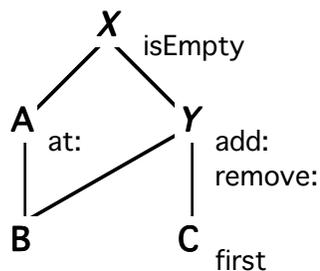
For the simple Smalltalk protocols described in Section 2.1, the conformance relationship is defined by inclusion on sets of message selectors. That is, protocol B conforms to protocol A if the set of selectors in B includes the selectors in A. For example, given

```
A = { isEmpty, at: }
B = { isEmpty, at:, add:, remove: }
```

B conforms to A because the selectors of B include the selectors of A. This means that any object supporting selectors listed in B can be used in places where selectors in A are required.

A protocol hierarchy is a graphical presentation of the conformance relation for a set of classes. The hierarchy also explicitly represents the sharing among protocols. For example, the three protocols A, B and

C = { isEmpty, add:, remove:, first }
 produce the following interface hierarchy:



For any two protocols that share some commonality, an *abstract protocol* is defined to represent the common behavior and the two protocols are identified as conforming to the new protocol. These new protocols are called abstract because they are not implemented explicitly by any class, but they are useful for describing the similarities between objects. In the diagram above, X and Y are abstract protocols.

The protocol hierarchy is computed from a set of classes. The algorithm starts with the protocol extraction method defined in the previous section:

protocol : class → set(selector)

An inverse function is then computed to map a selector to the set of classes that implement it..

inverse : selector → set(class)

for each class *c*

for each selector in *protocol(c)*

add *c* to *inverse(s)*

For example, the inverse mapping for the example given above is

isEmpty = { A, B, C }
 at: = { A, B }
 add: = { B, C }
 remove: = { A, B }
 first: = { C }

The hierarchy is then computed by collecting all the selectors that are defined by the same set of classes.

hierarchy : set(class) → set(selector)

for each selector *s*

add *s* to *hierarchy(inverse(s))*

For example, the inverse mapping for the simple example given above is:

{ A, B, C } = { isEmpty }
 { A, B } = { at: }
 { B, C } = { add:, remove: }
 { C } = { first }

A topological sort by set inclusion of the sets of classes in the domain of *hierarchy* gives the protocol hierarchy. A class name labels the node which is the minimum of all nodes in which the class appears (a minimum node may need to be added for it). In the example, the class B is a label for a new minimal node. Nodes that remain unnamed at this point are abstract and names must be invented for them.

Figure 1 illustrates this analysis for the standard Smalltalk-80 collection classes. The protocol information was extracted from ParcPlace Systems Objectworks for Smalltalk-80™, version 2.5. The nodes in the graph represent sets of classes that share a protocol extracted from the library. The selectors in the protocols are listed next to the nodes. Selector keywords in brackets are optional. The edges indicate protocol inclusion: all of the selectors listed for a node are included by all the nodes below it in the graph. The node names listed in *slanted* font are new abstract protocols that were created by the intersection of the other protocols.

Given a large library of classes cancellation and independent implementation tend to create a large number of abstract protocols and subtle relationships among them. Even when these classes are defined using only single inheritance, the protocol hierarchy that they embody can be a complex partial order, as seen in Figure 1. The hierarchy reveals a remarkably elegant structure of protocol sharing.

2.3 Design Review

A protocol hierarchy is a useful tool for reviewing the design of a class library.

The basic collection operations move upward in the graph where they are more widely shared, as abstract protocols. Several protocols have only one or two selectors. For example, the class *ExtensibleCollection* has only add: (and addAll: which depends upon it). The protocols below *ExtensibleCollection* represent those collections that can grow in size. These protocols correspond to the bounded,

or resizable, storage forms in the Eiffel data structure library [Meyer91].

Abstract protocols also arise from deletion of methods during inheritance. When messages are canceled from the subclass protocol as discussed in section 2.1, the subclass protocol does not conform to its superclass's. In this case both the subclass and superclass protocol conform to the protocol of the superclass minus those selectors canceled by the subclass.

The protocol hierarchy explicitly represents similarities between classes that are not related by inheritance. There are many examples in the Smalltalk libraries, one being the `add:` message on `LinkedList` and `Set`. This topic is examined in more detail in Section 6.

The hierarchy highlights messages that are not implemented uniformly. This is illustrated by the *SubtractableCollection* protocol, which contains only the binary minus (-) message. This message was extended for use on collections only recently in ParcPlace Smalltalk, and hence it appears to not be implemented in all the classes where it is expected. The uniformity of the class library is improved by implementing minus in `Bag` and `LinkedList`, and canceled it from `Dictionary`; *SubtractableCollection* would merge with the closely related remove messages in the *RemovableCollection* protocol.

Finally, the protocol hierarchy raises questions that cannot be answered by without a more detailed analysis of method behavior. For example, `Dictionary` cancels the removal methods it would otherwise inherit from `Set`. But it does not cancel the apparently symmetric methods for adding. Identifying this relationship raises the question of why `Dictionary` cancels the remove methods but not the add methods. Another example is the class `SortedCollection`, which supports messages for adding elements at particular places in the sequence. This is suspicious since `SortedCollections` provide an internally defined order. The next section applies specification tools to these problems.

3 Collection Class Specification

3.1 Specification Techniques

Specifications are developed for the collection class protocols to investigate further the design of the collection classes. Specification techniques available for analyzing objects include America's pre/post condition specifications with representation transfer functions [America91], Leaven's simulation model with traits [LW90], and Meyer's use of pre/post conditions and class invariants in Eiffel [Meyer91]. We use an version of America's formulation, since it is the simple and direct. The specifications for ordered collection and bag are based on America's.

Each class in the library has an abstract representation, and a set of methods defined by pre and post conditions on the representation and the method arguments. The set V represents the set of all possible Smalltalk values. In post conditions, a primed ($'$) variable indicates the value of the representation before the operation. The symbol ρ in a postcondition represents the result, or return value, of a method.

Specification conformance is determined by the existence of a transfer function that converts between abstract representations and preserves pre and post conditions. For specification B to conform to A , there must be a transfer function from the representation of B to the representation of A . In addition, for each operation in A , the precondition in A must imply the corresponding precondition in B , and the postcondition in B must imply the postcondition in A . For more explanation see America [91].

The `ifAbsent:` modifier on events is not discussed explicitly. It is understood as evaluating its exception block argument if the precondition on the basic method is violated. In addition, the "...All" variants of methods are not discussed, being repeated forms of the basic method.

The specifications are listed in an order convenient for discussion, roughly in order of increasing complexity. To deal with the problem of a selector with several incompatible specifications, new selectors, written in *slanted* font, are introduced in some cases. An attempt is made to retain the most common meaning of existing selectors.

3.2 Bags

Bags are collections that allow repeated elements but don't have any notion of ordering. Bags are represented as functions from values to natural numbers, indicating how many times the value occurs in the bag. The notation $F \equiv_{/x} G$ means that the functions F and G are equal at all points but x .

Bag

Representation $F : V \rightarrow \text{Nat}$

	isEmpty	$\rho = (\forall x F(x)=0)$
	size	$\rho = \sum_x F(x)$
	includes: x	$\rho = (F(x) > 0)$
	occurrencesOf: x	$\rho = F(x)$
	add: x	$F(x) = F'(x) + 1$ and $F \equiv_{/x} F'$
$F(x) > 0$	remove: x	$F(x) = F'(x) - 1$ and $F \equiv_{/x} F'$

3.3 Sets

Sets are represented as subsets S of the set of all values V . This specification also defines the behavior of the abstract protocols *ExtensibleCollection* and *RemovableCollection*.

Set

Representation $S \subseteq V$

	isEmpty	$\rho = (S = \emptyset)$
	size	$\rho = \#S$
	includes: x	$\rho = (x \in S)$
	include: x [add: x]	$S = S' \cup \{ x \}$
$x \in S$	removeEvery: x [remove: x]	$S = S' - \{ x \}$

This specification raises a basic problem with using message names alone as the basis for conformance and classification. The problem is that the **add:** and **remove:** on sets have different behavior than on other collections, which support multiple occurrences of elements. Since sets do not allow multiple occurrences, **add:** is weaker – ensuring only that the element is present, and **remove:** is stronger – eliminating the element completely.

If method names are to be carriers of behavioral meaning, new names are needed for these operations. Thus the two new message names, *include:* and *removeEvery:*, are listed in the specification. These messages have behaviors that are useful in any collection of elements; and they are meaningfully related to **add:** and **remove:**. By defining *include:* and *removeEvery:* on bags, they will conform to sets (although the **size** method must be omitted conformance to hold).

3.4 Indexed Collections

Indexed collections represent first-class functional mappings. These mappings need not be finite; however, they are constant because no update operations are defined. The representation is a function F with fixed domain K .

IndexedCollection

Representation $K \subseteq V$ and $F : K \rightarrow V$

$k \in K$	at: k	$\rho = F(k)$
	includesIndex: k	$\rho = (k \in K)$
$\exists k \mid v = F(k)$	indexOf: v	$F(\rho) = v$

The methods **indexOf:** from *SequenceableCollection* and **includesKey:** from *Dictionary* (renamed *includesIndex:* for uniformity) are introduced here because they apply equally well to all indexed collections. The choice of **index** to refer to the domain of the mapping is perhaps not perfect, but since indexed collections cover a wide variety of mappings (functions, arrays, hash tables, etc) no one word is likely to be natural in all cases.

3.5 Sequenceable Collections

Sequenceable collections are the base class for all collections that have a notion of ordering. They have as a representation a sequence $R : V^*$. Operations on sequences are concatenation $R_1 \bullet R_2$, length $\#R$, indexed access $R[i]$, and subsequence $R[1..u]$.

SequenceableCollection

Representation $R : V^*$

Preconditions:

$$\begin{aligned} \pi_1 &= \{ \#R > 0 \} \\ \pi_2 &= \{ 1 \leq i \leq \#R \} \\ \pi_3 &= \{ x \in R \} \end{aligned}$$

π_1	first	$\rho = R[1]$
π_1	last	$\rho = R[\#R]$

$\pi 2$	at: i	$\rho = R[i]$
	includes: x	$\rho = (\exists i . R[i] = x)$
	occurrencesOf: x	$\rho = \#\{ i \mid R[i] = x \}$
$\pi 3$	findFirst: x	$R[\rho] = x$ and $x \notin R[1 \dots \rho - 1]$
$\pi 3$	findLast: x	$R[\rho] = x$ and $x \notin R[\rho + 1 \dots \#R]$
$\pi 3$	indexOf: x	$R[\rho] = x$ and $x \notin R[1 \dots \rho - 1]$

With the introduction of a sequence order to indexed collections, the `indexOf:` message is defined to return one of the more specific index search methods, in this case `findFirst:`. This relationship between a method, e.g. `indexOf:`, on a less structured collection and method, e.g. `findFirst:`, on collections with more structure (like order or duplicate elements) is quite common; it is discussed more in Section 4.2

3.6 Linked Lists

A linked list is a particular representation for sequences of nodes. Each node can appear only once in the list, so linked lists are somewhat similar to sets. In the hierarchy it is used as a general name for a sequences from which elements can be added and removed at the ends. However, it would be better if `LinkedList` were a representational variant of a more general collection type.

`LinkedList`

Representation $R : V^*$

	addFirst: x	$R = x \bullet R'$
	addLast: x	$R = R' \bullet x$
	add: x	$R = x \bullet R'$
$\#R > 0$	removeFirst	$\rho \bullet R = R'$
$\#R > 0$	removeLast	$R \bullet \rho = R'$
$1 \leq i \leq \#R$	removeAtIndex: i	$R' = A \bullet \rho \bullet B$ and $R = A \bullet B$ and $i - 1 = \#A$
$\#R \geq n$	removeFirst: n	$R' = \rho \bullet R$ and $n = \# \rho$
$\#R \geq n$	removeLast: n	$R' = R \bullet \rho$ and $n = \# \rho$

The message `add:` is specified to be one of the order-dependent methods, in this case `addLast:`. The message `removeAtIndex:` is included here because it is directly related to the other two remove methods.

Two methods from `OrderedCollection`, `removeFirst:` and `removeLast:`, which remove a number of elements from either end of the list, are included here because they generalize the single element removal of `removeFirst` and `removeLast`. Not that although the names for these messages are similar to `addFirst:` and `addLast:`, they have very different behavior. All of these new methods can be efficiently implemented on linked lists.

3.7 Intervals

An interval is a collection of numbers between a lower and upper bound. This specification is a simplification of the Smalltalk `Interval` class, which also has a step, or increment, value. This specification assumes a step of 1. The interval is empty if the lower bound is greater than the upper bound.

`Interval`

Representation Integers L and U

	includes: x	$\rho = (L \leq x \leq U)$
	occurrencesOf: x	$\rho = 1$ if $L \leq x \leq U$ $\rho = 0$ otherwise
$U \geq L$	first	$\rho = L$
$U \geq L$	last	$\rho = U$
$i \leq U - L + 1$	at: i	$\rho = L + i - 1$
$L \leq x \leq U$	indexOf: v	$\rho = v - L + 1$
$U \geq L$	removeFirst	$\rho = L$ and $L = L' + 1$
$U \geq L$	removeLast	$\rho = U$ and $U = U' - 1$
$n \leq U - L + 1$	removeFirst: n	$\rho = \{L', \dots, L' - 1\}$ and $L = L' + n$
$n \leq U - L + 1$	removeLast: n	$\rho = \{U + 1, \dots, U'\}$ $U = U' - n$

The `indexOf:` method that was included in `SequenceableCollection` must be specified here, although it is not defined for intervals in Smalltalk. In

addition, the removal methods that work on the ends of a sequence are extended to apply to intervals.

3.8 Updatable Collections

An updatable collection is an indexed collection that can change over time. These collections tend to be finite mappings. The representation is a function and explicit domain as for indexed collections.

UpdatableCollection

Representation $K \subseteq V$ and $F : K \rightarrow V$

$k \in K$	at: k put: v	$F(k) = v$ and $F \equiv_{/k} F'$
-----------	--------------	--------------------------------------

The subtypes of updatable collections are distinguished by whether the domain is changeable (Dictionary) or fixed (Array).

3.9 Dictionaries

A dictionary is an updatable collection with a flexible domain. It is represented by an index set K together with a mapping F from indices to values. Dictionaries weaken the precondition on `at:put`: allowing new keys to be defined.

Dictionary

Representation $K \subseteq V$, $F : K \rightarrow V$

	at: k put: v	$F(k) = v$ and $F \equiv_{/k} F'$ and $K = K' \cup \{ k \}$
	add: p	$F(p.key) = p.value$ and $F \equiv_{/p.key} F'$ and $K = K' \cup \{ p.key \}$
	values	$\rho = \{ F(k) \mid k \in K \}$
$k \in K$	<code>removeIndex: k</code>	$K = K' - \{ k \}$

The `add:` method on Dictionary has a different specification from `add:` in other collections. Its argument is an *association*, which is a record with fields *key* and *value*. Adding an association to a dictionary removes a previously added association with the same key. It is unlikely that the various uses of `add:` can all be made to conform to a general specification. This is evidence that `add:` should be canceled from the Dictionary interface, and `addAssociation:` used in its place.

The message `keyAtValue:` is renamed `indexOf:` (defined in `IndexedCollection`) and `removeKey:` is renamed `removeIndex:` to be compatible with other indexed collections.

3.10 Arrays

An Array is an updatable collection with a finite, contiguous and fixed range of keys from 1 to some bound B .

Array

Representation $K = \{ 1, \dots, B \}$, $F : K \rightarrow V$

$k \in K$	at: k put: v	$F(k) = v$ and $F \equiv_{/k} F'$
	atAllPut: v	$\forall k . F(k) = v$
$1 \leq a \leq B$ $1 \leq b \leq B$	replaceFrom: a to: b with: v	$\forall a \leq k \leq b . F(k) = v$ $\forall k < a . F(k) = F'(k)$ $\forall k > b . F(k) = F'(k)$

3.11 Ordered Collections

Ordered collections are generally modifiable sequenceable collections. They are an extension of the `LinkedList` specification, represented by a sequence $R : V^*$.

OrderedCollection

Representation $R : V^*$

Preconditions:

$$\pi 1 = \{ e \in R \}$$

$$\pi 2 = \{ 1 \leq i \leq \#R \}$$

$\pi 1$	add: v before: e	$R = A \bullet v \bullet e \bullet B$ and $R' = A \bullet e \bullet B$ and $v \notin A$
$\pi 1$	add: v after: e	$R = A \bullet e \bullet v \bullet B$ and $R' = A \bullet e \bullet B$ and $v \notin A$
$\pi 2$	add: v beforeIndex: i	$R = A \bullet v \bullet B$ and $R' = A \bullet B$ and $i - 1 = \#A$
	<code>removeEvery: x</code>	$R = A_1 \bullet \dots \bullet A_n$ where $R' = A_1 \bullet x \bullet \dots \bullet x \bullet A_n$ and $x \notin A_j$
	<code>removeAll- SuchThat: c</code>	$R = A_1 \bullet \dots \bullet A_n$ where $R' = A_1 \bullet x_1 \bullet \dots \bullet x_n \bullet A_{n+1}$ and $\forall 1 \leq i \leq n . c(x_i)$ and $\forall e \in R . \neg c(e)$

3.12 Sorted Collections

A sorted collection is an ordered collection whose order is determined internally by a predicate \leq (which must be a total order). The representation is \leq together with a sequence R of values that are sorted with respect to \leq .

SortedCollection
 Representation $R : V^*$, $\leq : V \times V \rightarrow \text{Boolean}$
 Invariant: $1 \leq i \leq j \leq \#s \Leftrightarrow R[i] \leq R[j]$

add: x	$R = A \bullet x \bullet B$ where $R' = A \bullet B$ and $\forall a \in A . a \leq x$ and $\forall b \in B . x \leq b$
--------	---

The Smalltalk system allows the message `addFirst:` and `addLast:` as part of the `SortedCollection` protocol since it inherits from `OrderedCollection`. However, the specification for these messages violates the representation constraint for sorted collections: the new value cannot be guaranteed to be first in the sequence because it must be placed in sorted order. In the standard Smalltalk system the `addFirst:` message violates the representation invariant of `SortedCollections`.

There are two possibilities: either remove the messages from the sorted collection interface, or place additional constraints on them to make them valid. By adding a precondition to the messages, they can be brought into line with `SortedCollection`.

$\forall y \in R. x \leq y$	addFirst: x	$R = x \bullet R'$
$\forall y \in R. y \leq x$	addLast: x	$R = R' \bullet x$

Either canceling the methods or adding these preconditions prevents the sorted collection interface from conforming to the `OrderedCollection` interface.

It is arguable that it makes no sense to add a first element to a `SortedCollection` because the order of elements in a `SortedCollection` is determined internally: the client has no control over the order of elements. By omitting these methods, along with all the other order-specific methods, a `SortedCollection` is a `SequenceableCollection`, but not an `OrderedCollection`.

4 Specification Conformance

4.1 Class Conformance

The relationships among the classes specified in the previous section are defined in Figure 2. The diagram specifies which classes can simulate the behavior of other classes. For example, `OrderedCol-`

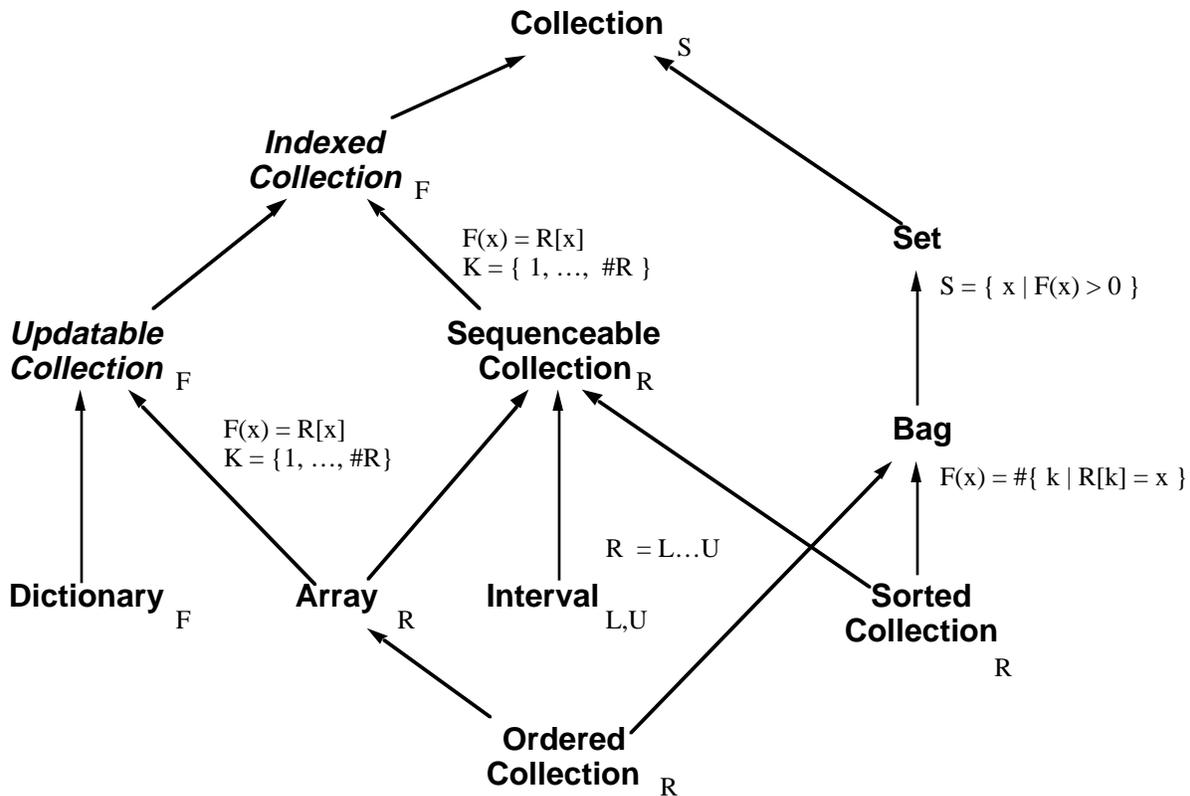


Figure 2: Class Specification Conformance

removeFirst, removeLast, removeFirst:, and removeLast: defined for Intervals.

removeFirst, removeLast, removeAtIndex: and removeAllSuchThat: defined for LinkedList

after: and before: moved to SequenceableCollection

addFirst:, addLast:, add:before:, add:after:, addAllFirst:, addAllLast:, and add:beforeIndex: canceled from SortedCollection.

5.2 Extending the Analysis

The process for analyzing class libraries can be applied to other parts of the Smalltalk. One area that would benefit from examination is the stream classes. These classes are conceptually similar to collections, but are implemented in an entirely different part of the class system.

A Stream is a destination or source of values. Streams are part of the collection classes but are not well integrated with the other collections. This section discusses how they could be unified with other collections

ReadStream
Representation $R : V^*$

	isEmpty	$\rho = (\#R = 0)$
$\#R > 0$	next	$R' = \rho \bullet R$
$\#R > 0$	peek	$\rho = R[1]$

The next method has the same specification as the removeFirst method in OrderedCollection. The fact that it removes the first element instead of the last is merely an artifact of the specification; it is not visible to the client. Similarly, peek corresponds to the first method.

WriteStream
Representation $R : V^*$

	nextPut: x	$R = R' \bullet x$
	contents	$\rho = R$

The method nextPut: has the same specification as addLast: in OrderedCollection, but is independent of the actual ordering used. Renaming the nextPut: to be add: allows for more polymorphism; WriteStream then conforms to ExtensibleCollection.

6 Interfaces Versus Inheritance

Figure 5 shows the Smalltalk inheritance hierarchy (in bold) superimposed on the protocol hierarchy of Figure 4 (dotted lines). This is a concrete illustration of the difference, even at a syntactic level, between inheritance and conformance [CHC90, Snyder86]. There are two cases where the hierarchy and protocol hierarchies are in direct conflict: Dictionary and SortedCollection. Dictionary inherits from Set but its protocol does not conform to Set's. This is because Dictionary cancels several of Set's methods. SortedCollection has a similar pattern of inheritance without conformance.

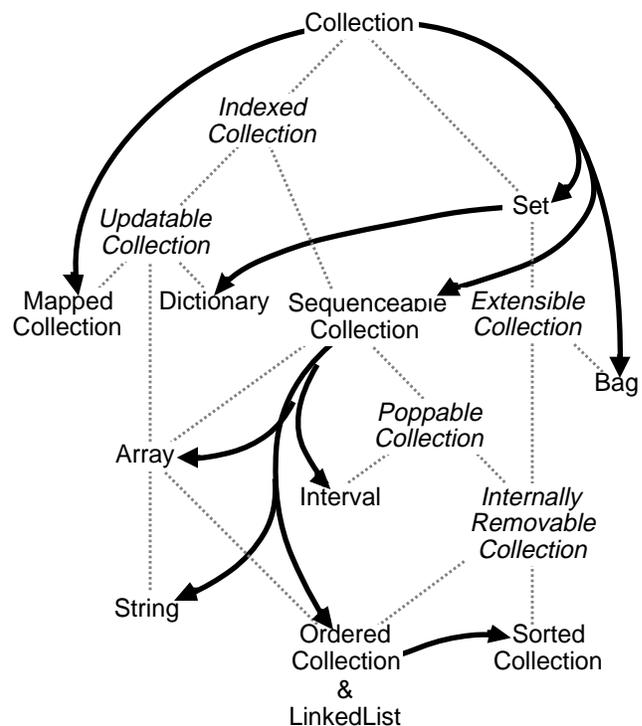


Figure 5: Interfaces versus Inheritance

Another significant deviation centers around SequenceableCollection, which has inheritors (subclasses) with various combinations of protocols unrelated to SequenceableCollection. Some of the subclasses (Array and String) are Updatable but not Extensible, since they support at:put:. Other subclasses (LinkedList and SortedCollection) are Extensible but not Updatable, since they support add:. A final one (OrderedCollection) is both Extensible and Updatable. The abstract classes in Smalltalk act as mixins for methods that depend upon a key subclass responsibility method; to express this struc-

ture more directly, Smalltalk would need multiple inheritance or mixins [BC90, Moon86, Carnese84].

7 Conclusion

Interfaces are a useful tool for analyzing class libraries. These interfaces may range in level of detail from protocols (sets of message names) to behavioral specifications. A detailed analysis of the Smalltalk collection class library demonstrates the usefulness of this approach.

Protocols have the advantage that they can be extracted automatically from classes in the Smalltalk. Two aspects of Smalltalk complicate this process. The first is the well-known problem of inheritance with exceptions, or deletion of methods. The second problem identified here is the use of abstract classes as "grab-bags" of inheritable functionality – in many cases the subclasses don't support various aspects of the abstract behavior. This use of abstract classes is an effective way to encode complex sharing of implementations in a single inheritance hierarchy, but it prevents easy determination of the messages a given class supports.

When protocols are placed into a hierarchy organized by protocol conformance, or subtyping, an alternative view of the structure of the library is revealed. This protocol hierarchy is a *clients* view of the library, in contrast to the *implementors* view provided by the inheritance hierarchy. Even though the classes are implemented using only single inheritance, protocol hierarchy of the collection class library is a complex partial order with multiple sharing of interfaces.

The protocol hierarchy is also useful for reviewing the design of a class library. Several omissions and implementation problems are immediately apparent. However, some questions are raised that cannot be answered without a more detailed analysis of method behavior.

A pre/post condition specification formalism, developed by Pierre America, is used for expressing specifications of the collection protocols. These specifications reveal the subtlety of the Smalltalk class library: each of the primary messages for adding, testing, and removing on the collection classes is a family of progressively more refined specification. The analysis also uncovers some prob-

lems in the library. Some inherited methods violate the subclass representation invariants. Some messages have incompatible specifications in different classes. There are also different messages with the same specification. For message names to be effective as carriers of behavioral meaning, care must be taken to ensure that the names are used consistently with the underlying formal specifications.

Acknowledgments

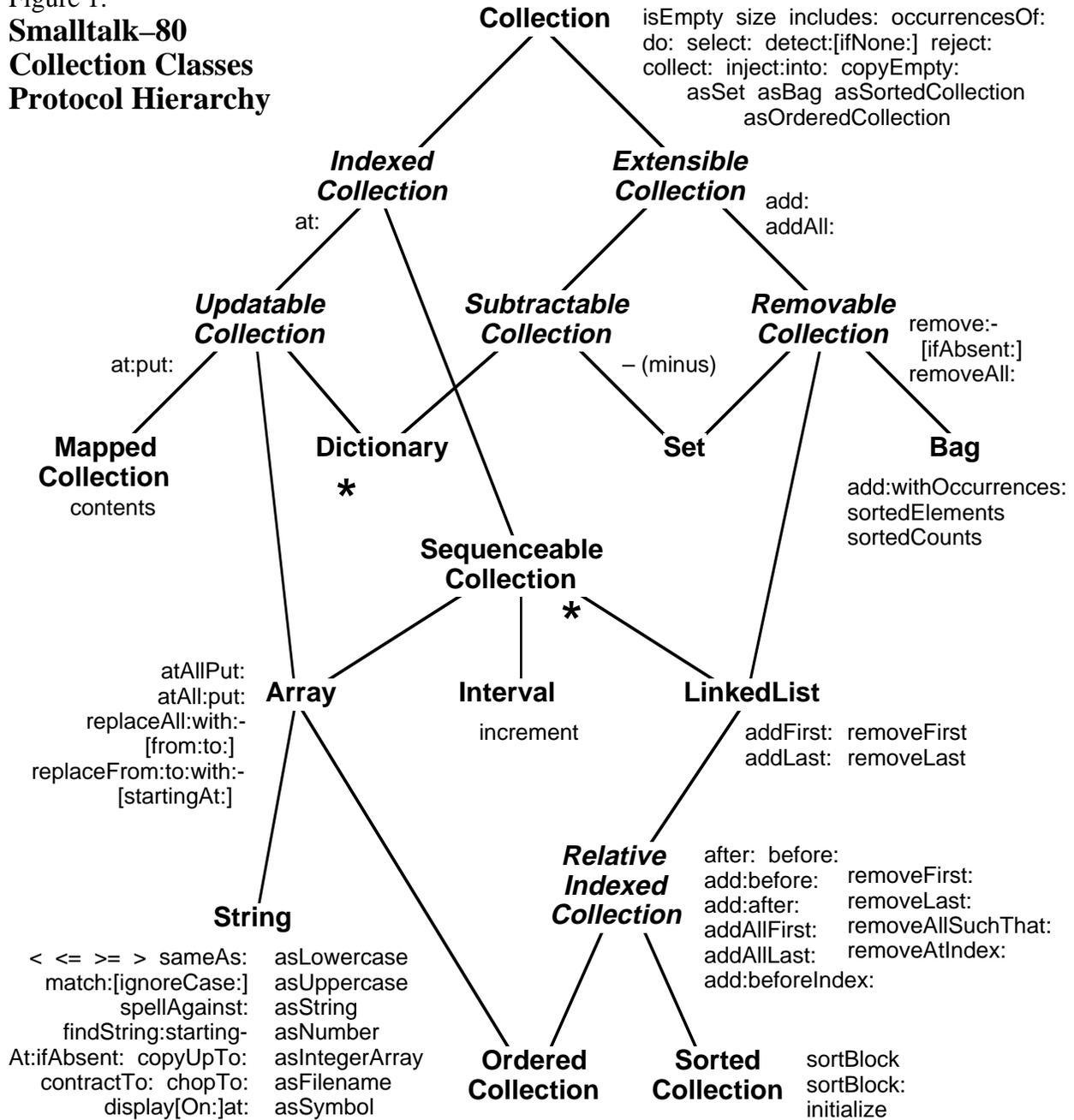
This work has its origins in the Abel project at HPLabs, where Peter Canning and Walt Hill contributed to its development. I would like to thank John Mitchell, Warren Harris, and Gary Leavens for their comments on the paper.

References

- [America91] P. America. "A behavioral approach to subtyping object-oriented programming Languages." In *Proc. of the REX School/Workshop on the Foundations of Object-Oriented Languages*, LNCS 489, Springer-Verlag, 1991.
- [BHJLC86] A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter. "Distribution and abstract types in Emerald." *IEEE Transactions on Software Engineering* SE-13:1, 1987.
- [BI8] A. H. Borning and D. H. Ingalls. "A type declaration and inference system for Smalltalk." In *Proc. of the ACM Symp. on Principles of Programming Languages*, 1982, pp. 133–141.
- [BC90] G. Bracha and W. Cook. "Mixin-based inheritance." In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1990, pp. 303–311.
- [CCHO89] P. Canning, W. Cook, W. Hill and W. Olthoff. "Interfaces for strongly-typed object-oriented programming." In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1989, pp. 457–467.
- [Cardelli84] L. Cardelli. "A semantics of multiple inheritance." *Semantics of Data Types*, LNCS 173, Springer-Verlag, 1984, pp. 51–68.
- [Carnese84] D. J. Carnese. "Multiple inheritance in contemporary programming languages." MIT Lab for Computer Science TR-328, 1984.

- [CHC90] W. Cook, W. Hill and P. Canning. "Inheritance is not subtyping." In *Proc. of the ACM Symp. on Principles of Programming Languages*, 1990, pp. 125–135.
- [LW90] G. T. Leavens and W. E. Weihl. "Reasoning about object-oriented programs that use subtypes." In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1990, pp. 212–224.
- [CM89] L. Cardelli and J. C. Mitchell. "Operations on records." DEC Systems Research Center Technical Note #48, 1989.
- [GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.
- [Graver89] J. Graver. Type-checking and type-inference for object-oriented programming languages. Ph.D. Thesis, University of Illinois, 1989.
- [JGZ88] R. Johnson, J. Graver and L. Zurawski. "TS: an optimizing compiler for Smalltalk." In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1988.
- [Johnson86] R. Johnson. "Type-checking Smalltalk." In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1986, pp. 315–321.
- [JF88] R. Johnson and B. Foote. "Designing reusable classes." *Journal of Object-Oriented Programming*, June/July 1988, pp. 22–35.
- [LTP86] W. R. LaLonde, D. A. Thomas and J. R. Pugh. "An exemplar based Smalltalk." In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1986, pp. 322–330.
- [Meyer91] B. Meyer. "Lessons from the design of the Eiffel libraries" *CACM* 33:9 (September 1991), pp. 68–84.
- [Meyer87] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1987.
- [Moon86] D. A. Moon. "Object-oriented programming with Flavors." In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1986, pp. 1–8.
- [Synder86] A. Snyder. "Encapsulation and inheritance in object-oriented programming languages." In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1986, pp. 38–45.
- [Suzuki81] N. Suzuki. "Inferring types in Smalltalk." In *Proc. of the ACM Symp. on Principles of Programming Languages*, 1981, pp. 187–199.

Figure 1:
Smalltalk-80
Collection Classes
Protocol Hierarchy



Dictionary protocol

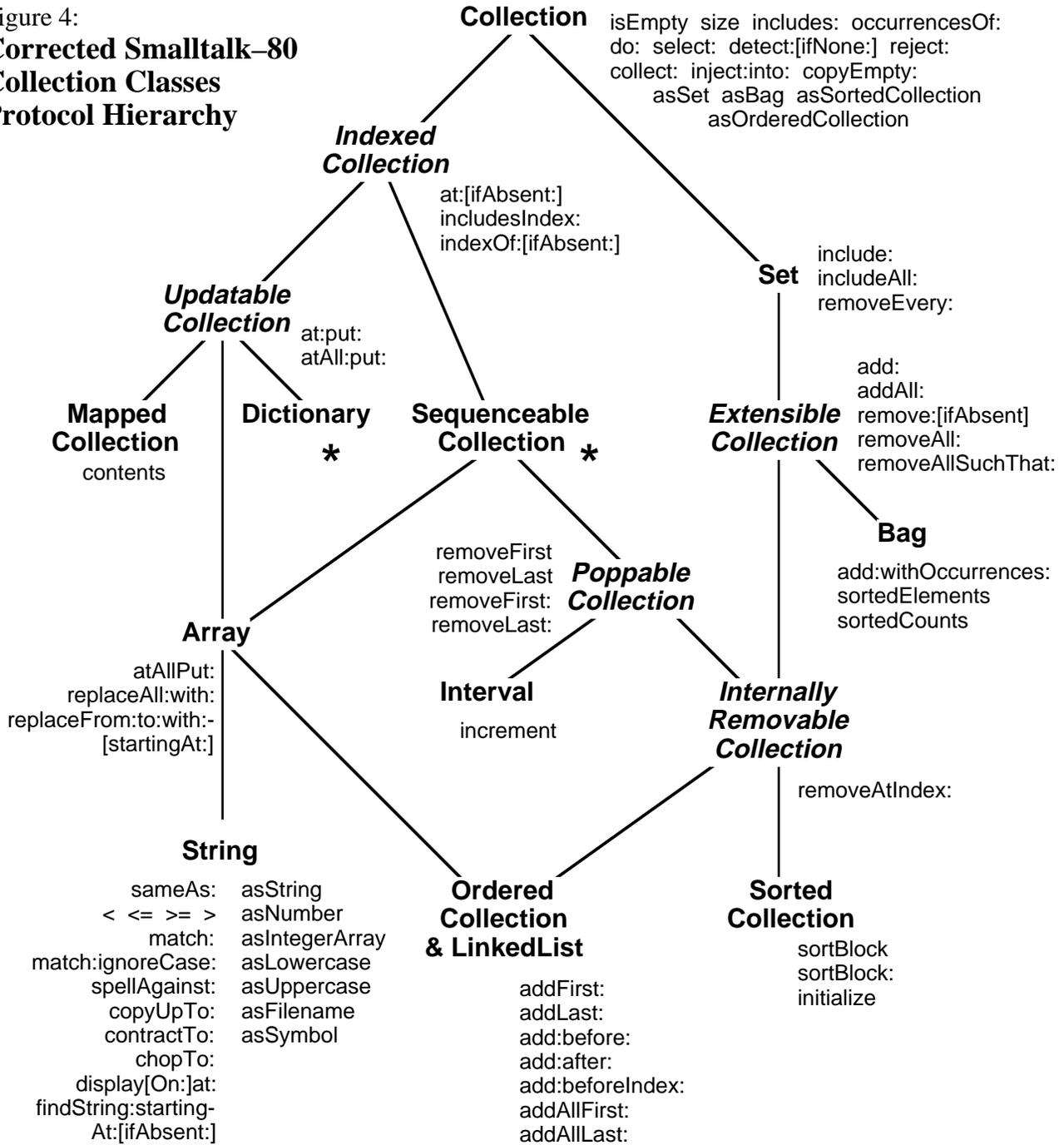
at:ifAbsent: values keys keysDo
includesKey: removeKey:[ifAbsent:]
associationAt:[ifAbsent:]
keyAtValue:[ifAbsent:]
includesAssociation: associations
associationsDo:
removeAssociation:[ifAbsent:]

SequenceableCollection protocol

first last reverse with:do: reverseDo: findLast: findFirst:
prevIndexof:from:to: nextIndexof:from:to:
copyReplaceFrom:to:with: copyReplaceAll:with: ,
copyFrom:to: copyWith: copyWithout: writeStream
readStream asArray mappedBy: indexOf:[ifAbsent:]
indexOfSubCollection:startingAt:[ifAbsent:]

Figure 4:

**Corrected Smalltalk-80
Collection Classes
Protocol Hierarchy**



Dictionary protocol

values keys keysDo
 removeKey:[ifAbsent:]
 associationAt:[ifAbsent:]
 includesAssociation: addAssociation:
 associations associationsDo:
 removeAssociation:[ifAbsent:]

SequencableCollection protocol

first last after: before: reverse with:do: reverseDo:
 findLast: findFirst: prevIndexOf:from:to:
 nextIndexOf:from:to: copyReplaceFrom:to:with:
 copyReplaceAll:with: , copyFrom:to: copyWith:
 copyWithout: writeStream readStream asArray
 mappedBy: indexOf:[ifAbsent:]
 indexOfSubCollection:startingAt:[ifAbsent:]