Applying Traits to the Smalltalk Collection Hierarchy

Andrew Black, Nathanael Schärli and Stéphane Ducasse

Department of Computer Science and Engineering
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Road
Beaverton, OR 97006-8921 USA

Technical Report Number CSE 02-014

also

Universität Bern
Institut für Informatik und angewandte Mathematik
Neubrückstrasse 10
CH-3012, Bern, Switzerland

Technical Report Number IAM-02-007

25th November 2002

A version of this paper has been submitted to the 2003
European Conference on Object-Oriented Programming (ECOOP).
If accepted, the paper will appear in the proceedings of ECOOP 2003,
and subsequent bibliographic citations should
refer to the conference proceeding

— This page is blank —

Applying Traits to the Smalltalk Collection Hierarchy[⋆]

Andrew Black, Nathanael Schärli, and Stéphane Ducasse

OGI School of Science & Engineering, Oregon Health and Science University, U.S.A
Software Composition Group, University of Bern, Switzerland
black@cse.ogi.edu, {schaerli, ducasse}@iam.unibe.ch

Abstract. Traits are a programming language technology modeled after mixins but avoiding their problems. In this paper we refactor the Smalltalk collections hierarchy using traits. We observed that the original hierarchy contained much duplication of code; traits let us remove all of it. Traits also make possible much more general reuse of collection code *outside* of the existing hierarchy; for example, they make it easy to convert other collection-like things into true collections. Our refactoring reduced the size of the collection hierarchy by approximately 12 per cent, with no measurable impact on execution efficiency. More importantly, understandability and reusability of the code was significantly improved, and the path was paved for a more intensive refactoring.

Keywords: Reuse, Mixins, Traits, Smalltalk, Collection, Hierarchy, Refactoring, Inheritance, Multiple Inheritance

1 Introduction

We have long believed that classes have too many responsibilities in object-oriented programming. In many languages, classes are used for conceptual classification. They also provide for reuse in two different ways: as factories, they can be used to instantiate many similar objects, while as superclasses, they can be incorporated into new subclasses. These two kinds of reuse often have conflicting requirements. As factories, classes must be complete, while when creating new subclasses, it is more convenient to be able to incorporate small fragments of behavior.

We have developed a new programming construct, which we call a *trait*, to address this problem. Traits are intended as fine-grained units of code reuse. In essence, traits are first class collections of methods that can be reused by classes anywhere in the inheritance hierarchy.

The contributions of this paper are:

- a study of the internal structure of the existing Smalltalk collections classes, with particular attention to code duplication, unnecessary inheritance, the use of **super** and method redefinition in inheritance chains;
- a report of our experience using traits to help us remove these problems; and
- a description of the new collection hierarchy that resulted from our refactoring.

[⋆] This research was partially supported by the National Science Foundation of the United States under award CCR-0098323, and by the Swiss National Foundation.

2 What is the Problem?

Single inheritance is a very popular programming technology, and has been adopted widely since its introduction in Simula 67 [BDMN73]. Inheritance is also very powerful, and the basis for several major success stories, including Smalltalk and Java. But this success and power should not blind us to the fact that sometimes inheritance is just not up to the task of supporting the wide range of abstractions that we expect to find in a modern object-oriented framework.

Let us illustrate this point with a small example. The class RectangleMorph in Squeak Smalltalk represents a rectangular block of color that can be viewed on the display. As such, it is a subclass of BorderedMorph, from which it inherits and reuses many methods; BorderedMorph in turn is a subclass of Morph. Squeak also defines a class Rectangle, which inherits from Object. However, RectangleMorph is *not* a Rectangle; that is, RectangleMorph does not implement all of the protocol understood by Rectangle objects.

It happens that Rectangle adds 83 messages to the protocol of Object. Of these, only 13 messages are also understood by a RectangleMorph; the other 70 are missing from RectangleMorph's protocol. We say "missing" because, to the client, a RectangleMorph clearly *is* a Rectangle; moreover, the state of a RectangleMorph includes a field bounds that defines the Rectangle that it occupies.

A programmer who wishes to fix this problem is faced with a number of unpleasant alternatives. One option is to copy the 70 missing methods from Rectangle and paste them into RectangleMorph. This is a clear violation of the DRY (Don't Repeat Yourself) principle [HT00].

Another option is to provide a conversion method RectangleMorph>>asRectangle[1], and expect the client to remember to use this conversion method whenever there is a need to send a message that a RectangleMorph does not understand. For example, instead of saying myMorph area, the client must say myMorph asRectangle area. This moves a burden onto the morph's client that we feel should be borne by the morph itself.

A third option is to delegate the 70 missing methods. The simplest way of doing this is to implement each of them as a one line method that converts the receiver to a rectangle and resends the message. So the area method would actually be implemented in RectangleMorph as follows.

```
RectangleMorph>>area
    ↑ self asRectangle area
```

This seems like the best choice, but it is hard to achieve without tool support, burdens the code for RectangleMorph with a lot of "noise", making it harder to understand, is inefficient, and increases the size of the object code.

Multiple inheritance has been proposed as another solution to this problem; multiple inheritance would allow RectangleMorph to have Rectangle *and* BorderedMorph as its superclasses and to inherit methods from both. But multiple inheritance is complex, and may introduce more problems than it solves. For example, with multiple inheritance, RectangleMorph would inherit two sets of state variables that represent the same

[1] The notation c>>name refers to the method on name in class c.

information (the position of the rectangle), and two sets of methods that access and change these variables. A whole literature has developed on how to resolve these problems; Taivalsaari [Tai96] provides a good starting point.

Traits provide a solution to the problem of giving RectangleMorph the behavior of a Rectangle while retaining the simplicity of single inheritance. The trait solution avoids duplication of both source and object code, eliminates indirection, and improves modularity, thus making the classes easier to understand.

3 What are Traits?

Stripped to its essentials, a trait is a first-class collection of named methods. Methods in a trait must be "pure behavior"; they cannot directly reference any instance variables, although they can do so indirectly. The purpose of a trait is to be composed into other traits and eventually into classes. A trait itself has no superclass; if the keyword **super** is used in a Trait, it is treated as a parameter that becomes bound when the trait is eventually used in a class.

Associated with each trait is a set of methods called the *requires* set. Any class that uses a trait will have to provide all of these required methods. For example, if the methods in a trait use the expression **self** size but the trait itself does not define a method size, then size will be in the requires set of the trait. When this trait is eventually incorporated into a class, size will have to be defined, perhaps as a method that fetches the value of an instance variable.

Two traits be combined using the symmetric *sum* operation $+$. The sum $T_1 \ + \ T_2$ contains all of the non-conflicting methods of T_1 and T_2. However, if there is a *conflict*, that is, if T_1 and T_2 both define a method with the same name, the two methods annihilate and *neither* is present in the sum.

Traits require an extended form of inheritance in which a new class is constructed from a superclass, a trait, and some local definitions of instance variables and methods. Locally defined methods replace methods obtained from the trait. As one would expect, methods defined locally and in the trait override those inherited from the superclass, but these methods can access the superclass method using **super**. In practice, the trait used to construct a new class is often the symmetric sum of several more primitive traits. If the new class is intended to be concrete, the required methods of the trait should all be supplied either by inheritance or in the local definitions.

In addition to serving as components of a class, traits can be used as components of other traits. A trait T can be constructed by composing some local definitions with another trait S. S becomes a subtrait of T; methods defined directly in T itself replace replace methods of the same name defined in S.

Because of the way that we define trait composition and inheritance, the semantics of a method is independent of whether it is defined in a trait T, or in a class or trait that uses T as a component. Consequently, it is always possible to *flatten* a nested trait structure at any level without changing any of the methods. We believe that this flattening property is crucial in making traits easy to use; it is one of the critical differences between traits and mixins.

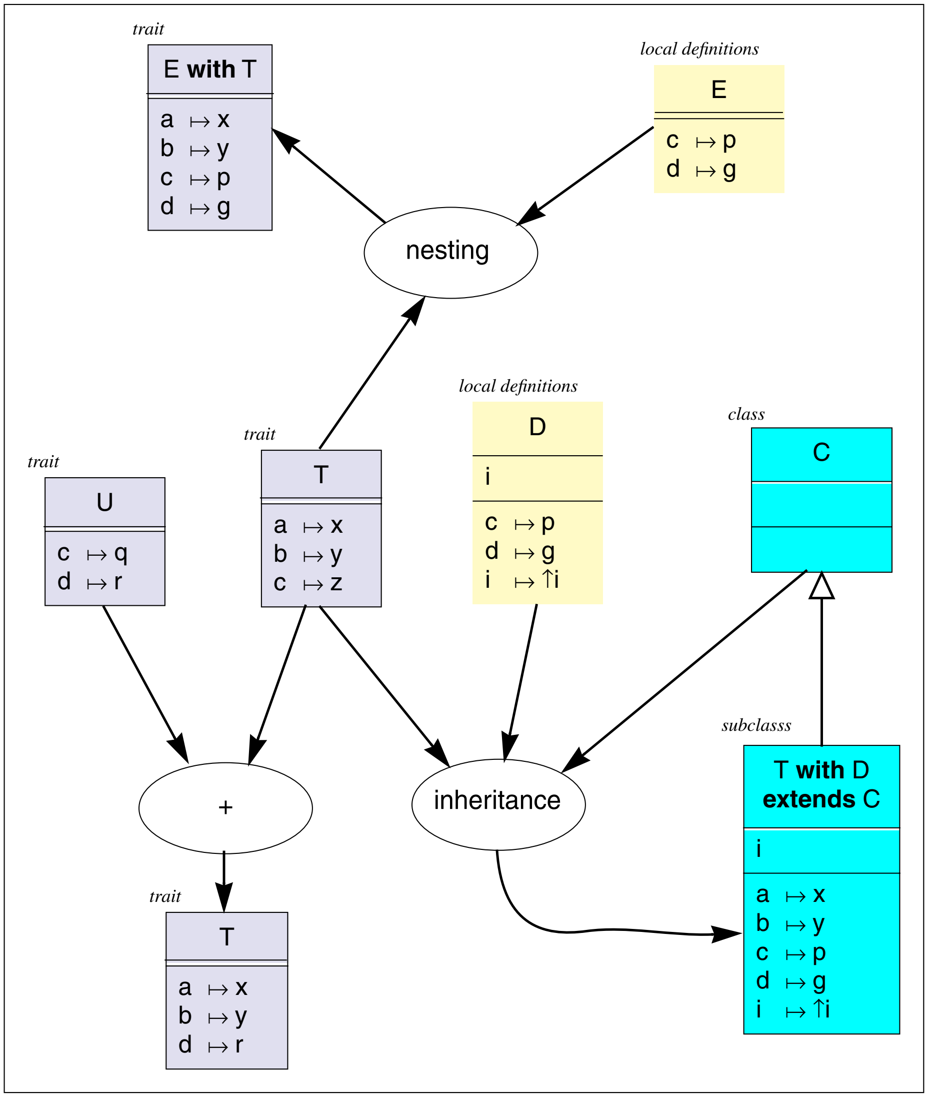


Fig. 1. This figure illustrates the major operations on traits: composition (nesting), inheritance, and sum. Note that local definitions and classes can contain instance variables and methods that refer to them, whereas traits cannot. local definitions are shown without an enclosing box because they are not reusable, in contrast to traits and classes.

A trait can also be derived from another trait by providing *aliases* for some of the methods. Aliases are useful to make conflicting methods available under another name, or to match the requirements of some other trait. Finally, a trait can be constructed by *excluding* methods from an existing trait; we use − as the exclusion operator. Exclusion is useful to avoid conflicts or if someone else has defined a trait that is "too big" for your application. We did not need to use aliasing or exclusion in refactoring the collections hierarchy, so they will not be described further.

The reader interested in a deeper understanding of traits and their composition operators, and in how traits avoid the difficulties that have beset multiple inheritance and mixins, is referred to companion papers [SDNB02a,SDNB02b].

4 Applying Traits to RectangleMorph

Now that the reader has at least a superficial understanding of what traits are, we can return to the example of RectangleMorph and show how traits can be used to make a RectangleMorph understand the 70 Rectangle methods missing from its protocol. Besides illustrating one way to use traits, this example illustrates the process used to transform a class into traits, the difficulties that may be encountered when recomposing the traits, and gives a glimpse at the tool support that we have built.

The first step is to construct a trait that contains the missing methods. This is easy to do, because the appropriate code already exists in class Rectangle. In the traits browser, an extension of the standard Smalltalk browser that understands traits, we use the "new trait from class" menu item to create a new trait from class Rectangle. We call the new trait TRectangle; the initial T in the name of a trait is a convention that we follow throughout this paper. TRectangle contains all of the methods of Rectangle, except that the *abstract variable* refactoring [2] is first applied to any method that accesses an instance or class variable directly.

For example, a Rectangle has two instance variables, origin and corner, which represent its top left and bottom right coordinates. So this method

```
Rectangle>>width
    "Answer the width of the receiver."

    ↑ corner x − origin x
```

is converted into

```
TRectangle>>width
    "Answer the width of the receiver."

    ↑ self corner x − self origin x
```

[2] This refactoring is given different names by different authors. Opdyke [Opd92] calls it "abstract access to member variable", Fowler [FBB^{+}99] calls it "encapsulate field", and the Refactoring Browser [RBJ97] calls it "abstract variable".

Once this refactoring is completed, the new trait has all of Rectangle's methods, but those that depended on the instance variables of Rectangle now depend instead on the existence of methods origin and corner. The traits browser lets us examine all of the methods TRectangle, and also shows its requires set, which contains the three messages origin, corner and species.

We can now use the browser to construct a new class RectangularMorph as a subclass of RectangleMorph. This is very much like the process of creating a new subclass in the ordinary Smalltalk browser. However, in addition to specifying the superclass, we are given the opportunity to write an expression specifying which trait (if any) we wish to *use*, *i.e.*, to nest inside the new subclass.

As a result, the new class RectangularMorph immediately has the 70 methods it needs from Rectangle. But it is incomplete; the browser shows us that it still requires definitions for methods origin and corner. We define these directly in the new class.

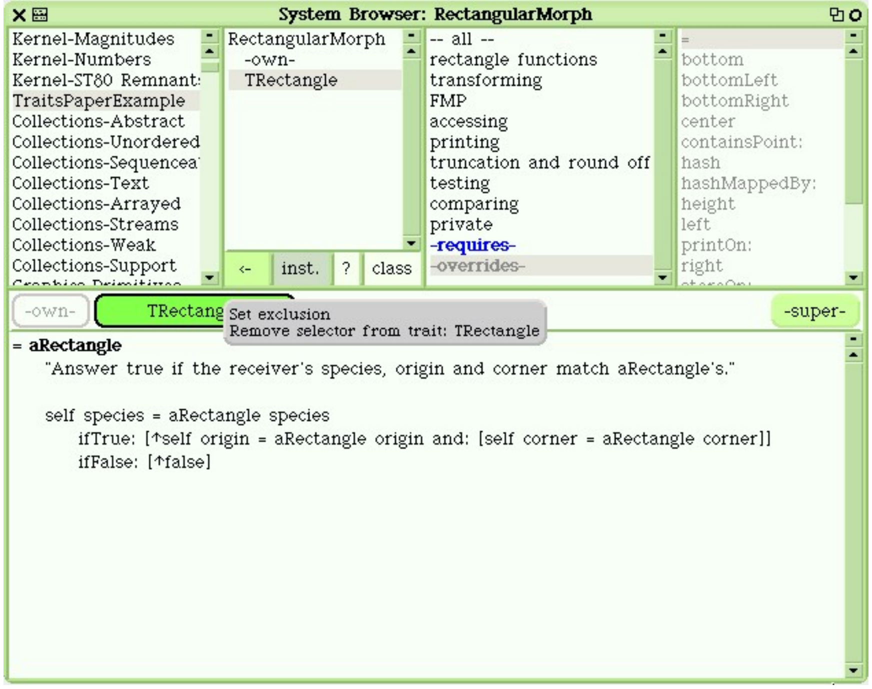
```
RectangularMorph>>origin
    ↑ self bounds origin

RectangularMorph>>corner
    ↑ self bounds corner
```

The requirement species has already been satisfied; our superclass RectangleMorph inherits this method from class Object, and thus we do too.

The browser also shows us a list of overridden methods, that is, methods where RectangularMorph redefines a method inherited from RectangleMorph. This is because some methods, like center, that are in the trait TRectangle, were also defined in RectangleMorph. We need to look at each of these methods, and decide whether we wish to keep the version from TRectangle, keep the version from the superclass, or write a new method. Buttons in the browser let us examine both of the alternatives and make our choice quickly.Throughout this process, the traits browser helps to focus the programmer on just those methods that require her attention. Once all of the overrides have been examined and all of the requirements met, our task is complete: we have created a new class that has the functionality of RectangleMorph and Rectangle. The only methods that we needed to write were the two "glue methods" origin and corner, which express how the abstract state of a rectangle is extracted from a RectangleMorph.

While looking at the overrides, we notice that the methods for =, hash and printOn: are inappropriate. We use a browser menu to exclude these methods from the composite RectangularMorph. We choose to do this by *setting an exclusion* (see figure 2), the effect of which is to modify the command that is used to build RectangularMorph so that the = and hash methods from TRectangle are never used:

```
RectangleMorph subclass: #RectangluarMorph
    uses: {TRectangle – {#=. #hash. #printOn:}}
    instanceVariableNames: ”
    classVariableNames: ”
    poolDictionaries: ”
    category: ’TraitsPaperExample’
```



```
= aRectangle
	"Answer true if the receiver's species, origin and corner match aRectangle's."

	self species = aRectangle species
		ifTrue: [↑self origin = aRectangle origin and: [self corner = aRectangle corner]]
		ifFalse: [↑false]
```

Fig. 2. The programmer examines the list of overrides created when RectangularMorph uses the trait TRectangle as a component. The = method from TRectangle is inappropriate, and is excluded.

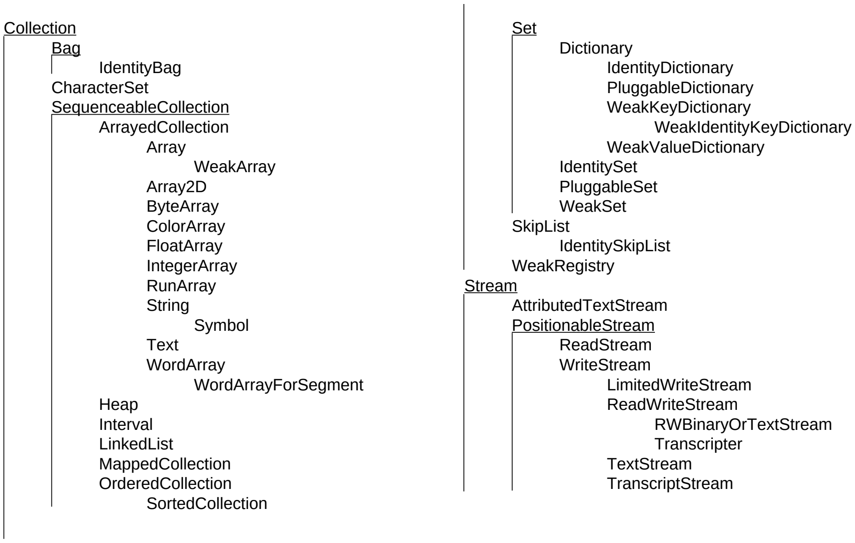
Small examples like this are fun, but they are not by themselves a compelling test of a tool and a language extension that have as their goal the understanding and reuse of large class libraries. To see how well traits perform in such a situation, they must be applied to a large framework. We chose the Smalltalk Collections Hierarchy for this experiment.

5 The Smalltalk Collection Hierarchy

The “collection hierarchy” is a loosely defined group of general purpose subclasses of Collection and Stream. The version of the hierarchy that appears in the“Blue Book”[GR83] contains 17 sub-classes of collection and 9 sub-classes of Stream, for a total of 28 classes, and had already been redesigned several times before the Smalltalk-80 system was released. This group of classes is often considered to be a paradigmatic example of object-oriented design.

In Squeak, the abstract class Collection has 98 subclasses, and the abstract class Stream has 39 subclasses, but many of these (like Bitmap, FileStream and CompiledMethod) are special purpose classes crafted for use in other parts of the system or in applications, and hence not categorized as Collections by the system organization. However, the system category Collections also includes various other classes, such as Character, Link and Association, that are related to collections but do not themselves exhibit collection behavior.

For the purposes of this study, we use the term “Collection Hierarchy” to mean the 49 classes that are subclasses of Collection or Stream and are *also* in the system category Collections. This is still a large group; the full list is shown in Figure 3. These 49 classes respond to 794 messages and define a total of 1236 methods.



- Collection
  - Bag
    - IdentityBag
  - CharacterSet
  - SequenceableCollection
    - ArrayedCollection
      - Array
        - WeakArray
      - Array2D
      - ByteArray
      - ColorArray
      - FloatArray
      - IntegerArray
      - RunArray
      - String
        - Symbol
      - Text
      - WordArray
        - WordArrayForSegment
    - Heap
    - Interval
    - LinkedList
    - MappedCollection
    - OrderedCollection
      - SortedCollection
  - Set
    - Dictionary
      - IdentityDictionary
      - PluggableDictionary
      - WeakKeyDictionary
        - WeakIdentityKeyDictionary
      - WeakValueDictionary
    - IdentitySet
    - PluggableSet
    - WeakSet
  - SkipList
    - IdentitySkipList
  - WeakRegistry
- Stream
  - AttributedTextStream
  - PositionableStream
    - ReadStream
    - WriteStream
      - LimitedWriteStream
      - ReadWriteStream
        - RWBinaryOrTextStream
        - Transcripter
      - TextStream
      - TranscriptStream

Fig. 3. The Collection Hierarchy in Squeak. Indentation indicates subclassing; for clarity, the scopes of the upper-level classes are indicated by lines.

5.1 The Varieties of Collection

To understand the challenge of refactoring the collection hierarchy, the reader needs at least a superficial knowledge of the wide variety of collections in these classes, their commonalities and their differences. The reader who is familiar with the Smalltalk collection hierarchy may safely skip this section.

Programming with aggregates rather than individual elements is an important way of raising the level of abstraction of a program. The Lisp function map, which applies an argument function to every element of a list, returning a new list, is an early example of this style, but Smalltalk-80 adopted aggregate-based programming as a central tenet. Modern functional programming languages such as ML and Haskell have followed Smalltalk’s lead.

Suppose you have a data structure containing a collection of student records, and wish to perform some action on all of the students that meet some criterion. Programmers raised to use an imperative language will immediately reach for a loop. But the Smalltalk programmer will write

```
students select: [ :each | each gpa < threshold ]
```

which evaluates to a new collection containing precisely those elements of students for which the bracketed function returns **true**. [3]

It is important to note that the message collect: is understood by *all* collections in Smalltalk. There was no need to find out if the student data structure was an array or a linked list: the collect: message is understood by both. Note that this is quite different from using a loop, where one must know whether students is an array or a linked list before the loop can be set up.

In Smalltalk, when one speaks of a collection without being more specific about the kind of collection, one means an object that supports well-defined protocols for testing membership and enumerating the elements. *All* collections understand the testing messages includes:, isEmpty and occurrencesOf:. *All* collections understand the enumeration messages do:, select:, reject: (which is the opposite of select:), collect: (which is like lisp's map), detect:ifNone: inject:into: (which performs a left fold) and many more. It is the ubiquity of this protocol, as well as its variety, that makes it so powerful.

Sequenceable		**Not Sequenceable**	
Accessible by Index	**Not Indexable**	**Accessible by Key**	**Not Keyed**
Interval SortedCollection Array ByteArray OrderedCollection	LinkedList	Dictionary IdentityDictionary PluggableDictionary	Set IdentitySet PluggableSet Bag IdentityBag

Fig. 4. Collections can be categorized according to whether or not they are sequenceable, *i.e.*, whether there are clearly defined first and last elements. All of the sequenceable collections except linked lists can also be indexed by an integer key. Of the non-sequenceable collections, dictionaries can be accessed by an arbitrary key, such as a string, while sets and bags cannot.

Beyond this basic uniformity, there are many different kinds of collection. Figure 4 shows a categorization of the collection classes according to whether they are sequenceable, that is, whether an enumeration of the collection starts from a first element and proceeds in a well-defined order to a last element. Array is the familiar indexable data structure with a fixed size; arrays are initialized to **nil** and can hold arbitrary objects. anArray at: n retrieves the nth element of anArray, and anArray at: n put: v changes the nth element to v. LinkedLists are sequenceable but not indexable, that is, they understand first and last, but not at:. The class OrderedCollection is more general than array;

[3] The expression in brackets can be thought of as a λ-expression defining an anonymous function $\lambda x.x$ gpa < threshold.

the size of an OrderedCollection grows on demand, and it has methods for addFirst: and addLast: as well as at: and at:put:. An Interval is an immutable collection defined by a computational rule when it is created. For example, 5 to: 16 by: 2 is an interval that contains the elements 5, 7, 9, 11, 13 and 15. It is indexable with at:, but cannot be changed with at:put:.

The differences between the various kinds of sequenceable collection manifest themselves in several different dimensions.

1. How is the order established? Sorted collections use a supplied total ordering function, intervals are implicitly ordered, while arrays and ordered collections are ordered explicitly when elements are inserted.
2. Is the size fixed (intervals and arrays) or variable (sorted collections, ordered collections, and linked lists)?
3. Is the collection immutable (Interval and Symbol, not shown in the figure) or mutable (the others).
4. Is the collection constrained to hold a particular kind of object, or is it completely general? ByteArrays are constrained to hold small integers, and LinkedLists are constrained to hold elements that conform to the Link protocol. IntegerArrays, CharacterArrays, WordArrays, Strings and Symbols, not shown in the chart, are also constrained.

The unordered collections (sets, bags and dictionaries) can be categorized in a different set of dimensions.

1. Are duplicates allowed (dictionary and bag) or disallowed (set)?
2. Can the elements be accessed by a key (dictionaries), or not (sets and bags)?
3. How are the keys (in a dictionary) or the values (in a set or a bag) compared? In other words: what test is used to ascertain whether two elements added to a set are "equal"? Dictionary, Set and Bag use the = method provide by the elements; the Identity variants of these classes use the == method, which tests whether the arguments are the same object, and the Pluggable variants use an arbitrary equivalence relation supplied by the user when the collection is created.

In addition to these categorizations by functionality, as re-implementors of the collection hierarchy we must also be aware of how the collection classes are implemented. As shown in figure 5, five main implementation techniques are employed.

1. Arrays store their elements in the (indexable) instance variables of the collection object itself; as a consequence, arrays must be of a fixed size, but can be created with a single memory allocation.
2. OrderedCollections and SortedCollections store their elements in an array that is referenced by one of the instance variables of the collection object. This means that the internal array can be replaced with a larger one if the collection grows beyond its storage capacity.
3. The various kinds of set and dictionary also reference a subsidiary array for storage, but use the array as a hash table. Bags use a subsidiary Dictionary, with the elements of the bag as keys and the number of occurrences as value.
4. LinkedLists use a standard singly-linked implementation.

Arrayed Implementation	Ordered Implementation	Hashed Implementation	Linked Implementation	Interval Implementation
Array	OrderedCollection SortedCollection	Set IdentitySet PluggableSet Bag IdentityBag Dictionary IdentityDictionary PluggableDictionary	LinkedList	Interval

Fig. 5. Some collection classes categorized by implementation technique.

5. Intervals are represented by three integers that record the bounds and the step size.

Readers interested in learning more about the Smalltalk collections are referred to LaLonde and Pugh's excellent book [LP90]

6 Analysis of the Smalltalk Collection Hierarchy

This section presents the results of an analysis of the collection hierarchy as it existed before our refactoring. It shows that the collection hierarchy contains unnecessary inheritance, duplicated code, and other shortcomings.

Given the many dimensions on which the Smalltalk collection classes can be categorized, it is inevitable that any attempt to organize them into a single inheritance hierarchy will run into severe difficulties. As Cook[Coo92] showed, the hierarchy attempts to maximize reuse at the expense of conceptual categorization, with the consequence that, for example, Dictionary is a subclass of Set because it shares much of the same implementation, even though it does not share the same interface.

Another way that the designers of the hierarchy attempted to maximize reuse was to move methods high up, so that all possible client classes have a chance to inherit them. For example, collect: is implemented in Collection, but the implementation is appropriate only for those collections that understand add:. Consequently, this implementation is overridden by the abstract class SequenceableCollection in favor of an implementation using at:put:. This second implementation is inherited by all of the ArrayedCollections, but also by OrderedCollection and SortedCollection, for which it is not appropriate, and which override it again. All told, there are 10 implementations of collect: in the collections hierarchy.

The following subsections examine these effects more systematically.

6.1 Unnecessary Inheritance in the Collection Hierarchy

Inheritance is used quite heavily in the collection classes, mostly for sharing implementation, but also for classification[Coo92]. As a measure of the complexity of the inheritance relationships, we counted the number of inheritance chains in which a message has three or more methods defined on it. Figure 6 illustrates two examples: the methods at:put: and add: in the inheritance chains terminating in the class WeakValueDictionary. We found 79 such inheritance chains.

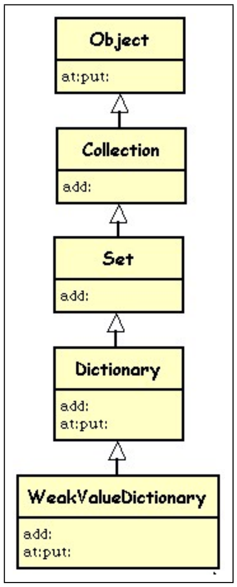


Fig. 6. Redefinition of at:put: and add: in the superclasses of WeakValueDictionary

There is nothing intrinsically wrong with re-defining a method inherited from one's superclass. On the contrary, the ability to use **super** to call the inherited definition from within the new method gives inheritance much of its power, and many people consider that adding behavior before or after a super-send is the epitome of inheritance-oriented programming.

However, for the most part, redefinition using **super** is *not* what is going on here. A total of 258 methods are involved in the 79 method redefinition chains mentioned above. Since 79 methods are at the top of a chain, $258 - 79 = 179$ methods have the opportunity of sending to **super**; only 15 actually do so. Neither are these redefinitions examples of "hook" methods that are being used to parameterize the behavior of a template method [ABW98]: all of the redefined methods are part of the functional interface. We deduce that for the most part these redefinitions are *correcting*, rather than augmenting, the behavior of the inherited method so that it is appropriate for the new subclass. In other words, we have identified 164 places where a method was inherited unnecessarily.

What is the problem with unnecessary inheritance? The cost is not in execution time nor in code space but in lost development time. The task of understanding a class that inherits several methods but does not use them is more complicated than necessary. Inheritance is often considered to be an aid to understanding a complex class, since the programmer can work down the inheritance chain, comprehending only the *differences* between a subclass and its superclass, rather than having to comprehend the entirety of the final subclass in a single step. To the extent that methods are inherited unnecessarily, this process is made more difficult, and inheritance begins to hinder rather than assist in understanding legacy code.

6.2 Code Duplication in the Collections Hierarchy

When a new subclass *does* want to re-use a method from an existing class, it may nevertheless be unable to do so because of the restrictions implied by the arrangement of the classes in a hierarchy. For example, PluggableSet and PluggableDictionary share some methods, but there is no place from which they could inherit them. PluggableDictionary is a subclass of Dictionary, and PluggableSet is a subclass of Set; there is no appropriate common superclass in which methods shared by the two pluggable classes can be placed. There is an *inappropriate* superclass: Set. The programmer is left with the choice of placing a method too high in the hierarchy, or duplicating it. This point is discussed further in section 7.2.

There is no easy way to ascertain how much duplication is caused by the fact that methods can be inherited only from a superclass. We made a superficial check by looking for methods whose decompile strings were identical. This detected as duplicates methods that differed only in formatting, comments, or the names of temporary variables. We also excluded error methods such as self shouldNotImplement, which is used to cancel an inherited method. In this way we found 28 pairs of duplicated methods, and 3 triples. In a few cases the duplication was gratuitous, that is, a subclass duplicated a method from its superclass when it could in fact have inherited it. (For example, Set and WeakSet both define size identically, even though WeakSet inherits from Set.) However, in most cases, the duplication was of a method from another part of the hierarchy, which consequently could not be inherited, or of a method defined in a superclass' superclass. For example, Dictionary and Collection both implement occurrencesOf: identically, but even though Dictionary is a subclass of collection, there is an intervening definition of occurrencesOf: in Set that prevents Dictionary from reusing the definition in Collection.

However, these duplication counts are almost certainly just the tip of the iceberg. Our primitive duplicate detection technique certainly misses many methods that differ in structure but not in semantics. For example, if two methods compare x and y for equality, but one expresses this as x = y while the other uses y = x, this duplication will not be revealed by our search. During our refactoring of the collections classes we also noticed many deeper examples of code duplication, where a method had clearly been copied from an established class into a newly created class, and then a single crucial statement had been changed to obtain a different semantics. The right way to avoid this sort of duplication is to refactor the original method so that the crucial statement is a parameter of some kind, either by adding an explicit argument to the method or, more commonly, by transforming the crucial statement into a send of a new message to **self**. The refactored version can then be reused in both the original and the new contexts, with different values for the parameter. Unfortunately, this refactoring is only possible if the newly created class is a subclass of the established class — a condition that cannot always be met.

6.3 Conceptual Shortcomings in the Collection Hierarchy

In addition to these problems in the *implementation* of the collection classes, the hierarchy also suffers from some conceptual shortcomings.

One of the reasons that there are so many collection classes is that the designers have attempted to compensate for the fact that classes are hard to reuse by providing all possible combinations of features. For example, Sets, Bags and Dictionaries must compare elements (or keys) for equality. Thus, each structure needs three variants: one that that uses = between elements, one that uses ==, and one that uses an equality function that is "plugged in" when the structure is created. Thus, we have the three classes Set, IdentitySet, and PluggableSet; the same is true for Dictionary and Bag, except that PluggableBag is missing. A similar situation exists with the "weak" variants of the collection classes, which hold onto their elements weakly, *i.e.*, in a way that does not prevent them from being garbage collected. It would be nice if these characteristics could be captured as reusable components, so that programmers could combine pluggability with, say, SkipLists, so that they could build the data structure that suits their application. This would simultaneously simplify the collection hierarchy (by eliminating the combinatorial explosion of features) *and* give the programmer the flexibility to choose from a wider range of collections.

Immutability is a "feature" not provided in the current hierarchy except in a few special cases. Symbols and Intervals are currently the only immutable collections, but immutable collections can be useful in many contexts. Strings are almost always used as if they are immutable, as are literal arrays, but this cannot be captured by the current set of classes.

The stream classes also exhibit many orthogonal features, such as read vs. write, binary vs. text, positionable (seekable) or not-positionable. The more necessary combinations are implemented by duplicating code; many other combinations are simply unavailable.

Finally, we mention that collection-like behavior is often desired for objects that are not primarily collections. For example, the class Path is a subclass of DisplayObject and thus not able to inherit from Collection. A Path represents an ordered sequence of points; arcs, curves, lines and splines are all implemented as subclasses of Path. Path implements some of the more basic kinds of collection-like behavior; for example, it has methods for at:, at:put:, and collect:. But Path does *not* attempt to implement the full range of collection behavior, for example, Path does not provide methods for select: and do:. There are simply too many such methods to make it viable to reimplement them, and the existing implementation cannot be reused. Section 7.3 discusses how traits address this problem.

7 Results

In this section we present the collection hierarchy that emerged from our refactoring efforts. We start by describing how we distributed behavior from the pre-existing abstract and classes into traits, and how those traits are used to construct a new set of classes. We then analyze the new hierarchy with respect to code duplication, possibilities for reuse and other issues.

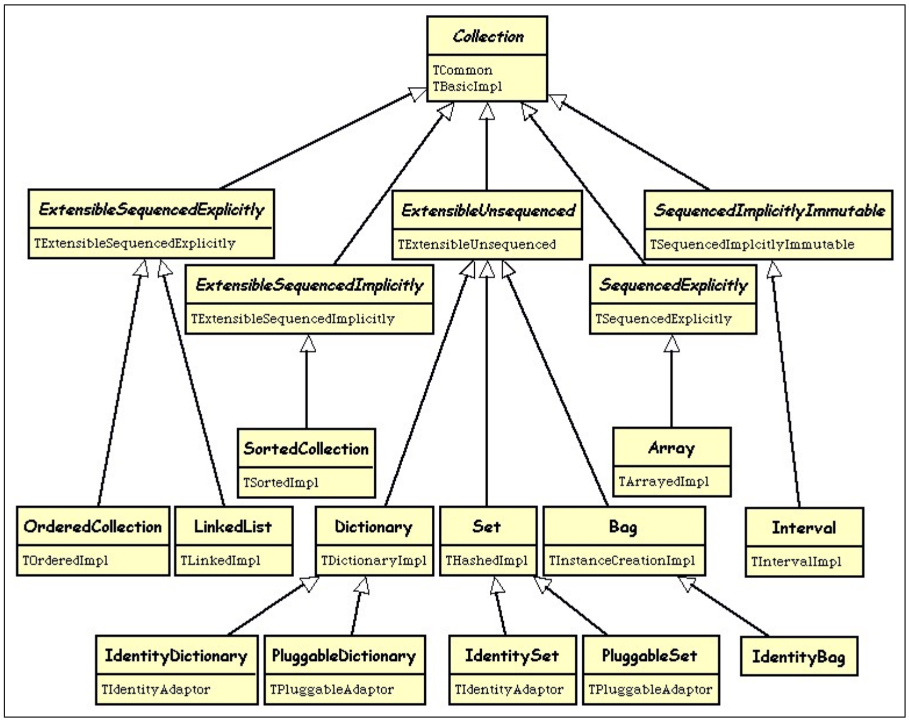


Fig. 7. The refactored collection hierarchy. Classes with italicized names are abstract; below the class name we show the top-level traits from which the class is composed. Each of these traits is in turn composed from several subtraits.

7.1 The New Collections Hierarchy

Figure 7 shows the new hierarchy for 13 common concrete collection classes and 6 abstract superclasses. This is as far as our refactoring has progressed to date. In addition to the name of each class, it also shows the traits from which the class is composed. The classes are divided in three layers. At the top of the hierarchy is the abstract class Collection, which is composed from two traits, and provides a small amount of behavior for all collections. Then we have a layer of 5 abstract classes that represent different combinations of the externally visible properties of collections. We call these properties *functional*, to distinguish them from implementation properties. At the bottom, we have 13 concrete classes that inherit from the functional classes and also use a trait that specifies an implementation.We now describe the functional properties and the implementation traits in turn.

The Functional Traits. Each kind of collection can be characterized by several different properties such as being explicitly ordered (*e.g.*, Array), implicitly ordered (*e.g.*, SortedCollection), unordered (*e.g.*, Set), extensible (*e.g.*, Bag), immutable (*e.g.*, Interval), or keyed (*e.g.*, Dictionary); see section 5.1 for more discussion.

Although single-inheritance is not expressive enough to represent such a diverse set of classes that share many different properties in various combinations, trait composition works well for this task. All that was necessary was to create a trait for each of these properties and then combine them to build the abstract classes shown in Figure 7. In order to achieve maximal flexibility, we ensured that the combinations of property traits are available in two forms: as traits that can be reused outside of the collection hierarchy, and as superclasses that can be inherited within it.

We also modularized the primitive properties more finely than would have been necessary if our only goal were to avoid code duplication. The fine structure gives us and future programmer more freedom to extend, modify and reuse parts of the new hierarchy. In addition, some of the property traits contain many methods, and creating subtraits corresponding to individual sub-properties gives them internal structure that makes them easier to understand. Because of the flattening property, there is no cost to this fine-grained structure: it is always possible to flatten it out and to work with the code in a less structured view.

Figure 8 shows how the composite property traits are built from each other and from the more primitive traits. We use the following naming convention. Some names have a suffix, which may consist of letters from the sets {S, U} and {M, I}. The letter 'S' indicates that all of the methods in the trait require the collection to be sequenced, whereas 'U' means that none of the methods in the trait requires the collection to be sequenced. Similarly, 'M' means that all the methods require the collection to be mutable, and 'I' means that no method requires the collection to be mutable. If the suffix does not contain a letter from one of these groups, the trait contains some methods with each characteristic.

As an example, the trait TEnumerationUI contains the part of the enumeration behavior that does not require sequencing ('U'), whereas EnumerationI — which uses the trait TEnumerationUI as a subtrait — contains some methods that do require and some methods that do not require sequencing. Furthermore, all of the methods in both of these traits treat the target object as immutable ('I').

The Implementation Traits. Besides the functional property traits, which are visible to a client, each collection class is also characterized by an implementation, which is hidden. The functional and implementation traits are largely independent.

The refactored hierarchy separates the traits specifying the implementation of a collection from the traits specifying the functional properties. This allowed us to freely combine different functional property traits (*e.g.*, TExtensibleSequencedExplicitly and TExtensibleSequencedImplicitly) with any of the suitable implementations (*e.g.*, the linked-list implementation and the array-based implementation). In one place we also

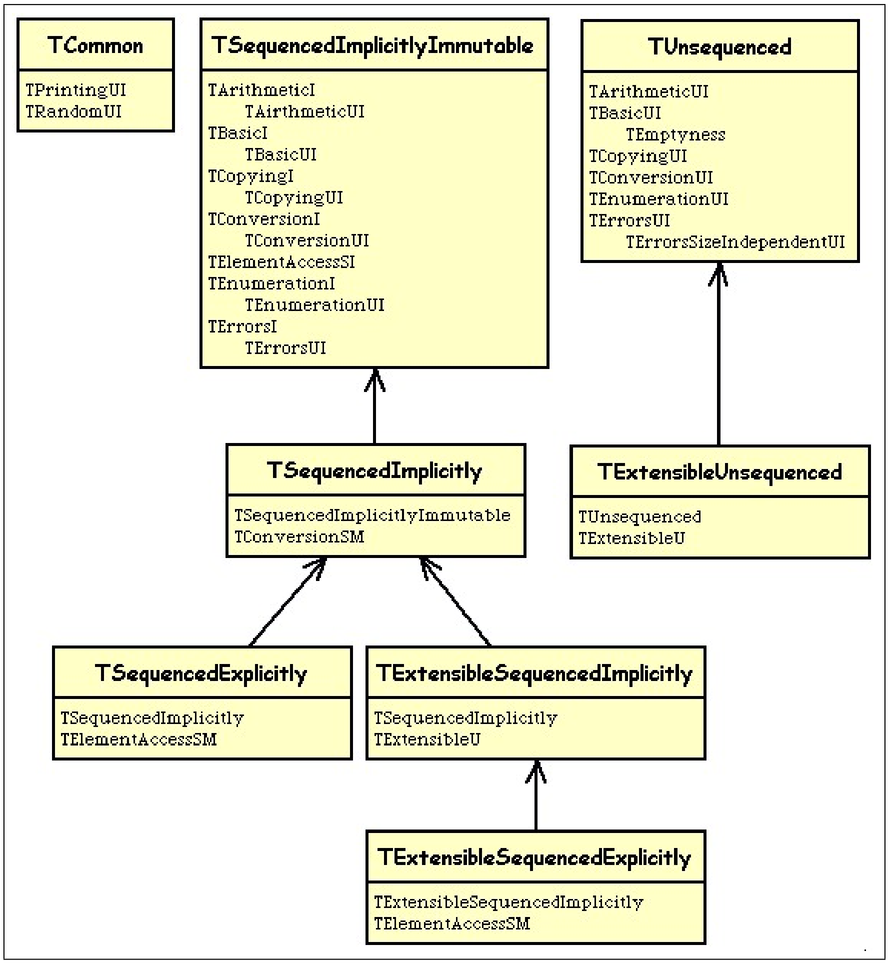


Fig. 8. The traits corresponding to composite functional properties are built from each other and from the more primitive traits. Each of the boxes represents a compound trait; the subtraits from which it is composed are listed in the bottom part of the box. Nesting of traits is indicated with indentation. If a subtrait is shown in the figure, its use is also indicated by an arrow. Thus, TExtensibleUnsequenced is composed from TUnsequenced and TExtensibleU. TUnsequenced is shown in the figure, so there is an arrow from TExtensibleUnsequenced to TUnsequenced. (TExtensibleU is primitive, and is not shown in the figure.)

extended the functionality of a concrete class: our LinkedLists are indexable, *i.e.*, they understand at:.

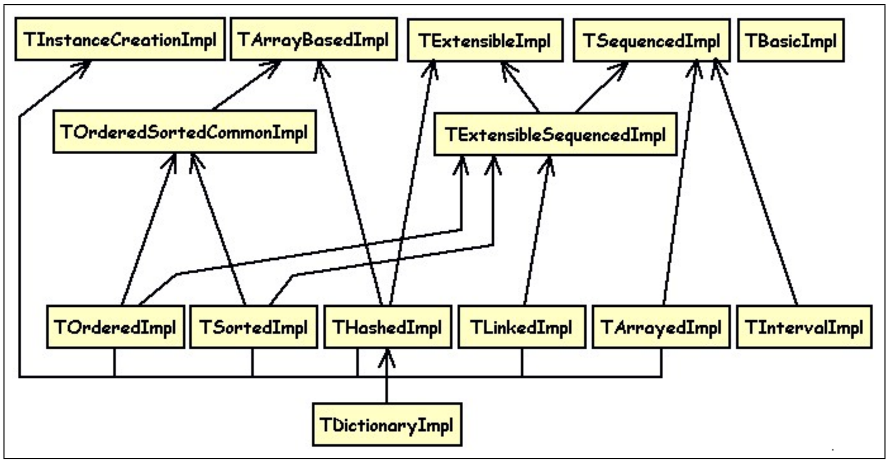


Fig. 9. Composition of the implementation traits. The seven implementation traits (TOrderedImpl, TSortedImpl, ...TDictionaryImp) at the bottom of the figure are composed from the reusable fragments shown at the top (TInstanceCreationImpl, TArrayBasedImpl,...TBasicImpl). Thus, TDictionaryImpl is constructed by adding some local definitions to the subtrait THashedImpl.

Figure 9 shows how the implementation traits used for creating the concrete classes shown in figure 7 are composed from subtraits representing more primitive implementation properties. The primitive traits are shown near the top of the figure; their main purpose is to allow the composite implementation traits shown near the bottom of the figure to share methods. As an example, we factored out the behavior for creating new instances (new, with:, withAll:, *etc.*) into the trait TInstanceCreationImpl, which is then used by TOrderedImpl and four other implementation traits.

The trait TBasicImpl contains default implementations for methods like includes: and hash. These defaults are written so as to be independent of the implementation of the underlying collection, but may be unnecessarily slow for certain implementations. For example, includes: is implemented using anySatisfy:; this is always correct, but is $O(n)$, whereas the hashed implementation should make this operation $O(1)$. Instead of using TBasicImpl as a subtrait of all the concrete implementation traits, we decided to use it in the root class of the collection hierarchy, from where its methods are inherited (and possibly overridden) by the various concrete implementations. For example, THashedImpl and TIntervalImpl have their own implementations of includes:.

7.2 Quantitative Assessment of the Refactored Hierarchy

The refactored part of the class hierarchy shown in Figure 7 contains 13 concrete classes and 6 abstract classes, which use a total of 46 traits. Each classes is composed of between 0 and 20 subtraits. The total number of methods is 509 (of which 36 are automatically generated accessor methods). This is just over 5 per cent fewer methods than in the equivalent part of the original hierarchy, which contains 540 methods. If we do not count the automatically generated accessor methods, which are not present in the original implementation because instance variables are accessed directly, our implementation contains 12.4 per cent fewer methods.

This number corresponds quite closely to the observed reduction in total size of the code. The refactored implementation is 10 per cent smaller than the original implementation if we measure bytecode, or 12 per cent if we measure source code[4]. Part of the size reduction is caused by eliminating duplicate methods. Whereas our primitive detection of code duplication identified 18 duplicate methods in the part of the original implementation that we refactored, we could not find any duplicates in new one.

The reduction in code size is even more remarkable considering that 57 (over 10 per cent) of the methods in the original implementation are placed “too high” in the hierarchy *specifically to enable code sharing*. The penalty for this is the repeated need to cancel inherited behavior. In the part of the original hierarchy that we refactored, 9 messages are explicitly disabled in subclasses by implementing methods that cause a runtime error (typically **self** shouldNotImplement). More problematic are another 48 methods that are *implicitly* disabled because they directly or indirectly call explicitly disabled methods.

This tactic does avoid the need to duplicate code, but the cost is that the whole hierarchy is very much harder to understand. For example, the method addAll: is implemented in Collection with a method that sends **self** add: for each element of the argument. Consequently, it *appears* that every collection understands addAll:, although an attempt to use this method on, say, an array, will always cause a runtime error. In the trait implementation, there is no need to resort to this tactic: each method is present in exactly the classes that need it, and in no others. This makes the classes much easier for a programmer to understand: browsing the protocol of a class tells her exactly what methods are available for use.

7.3 Subjective Assessment of the Refactored Hierarchy

Besides the quantitative improvements in the refactored part of the collection hierarchy noted above, the trait implementation has other advantages that will have impact both inside and outside the collection hierarchy.

We will certainly be able to reuse many of our traits when we extend our refactoring to other parts of the existing hierarchy. In addition, the traits that we have written will allow us to construct new kinds of collection by just composing the necessary traits and implementing a few glue methods. For example, we can build a class PluggableBag

[4] We measured source code size by decompiling the methods. This excludes comments and automatically adjusts for differences in formatting, naming of variables, and so on

by simply using the trait PluggableAdaptor in a subclass of Bag, and we can create immutable variants of collections by omitting the mutable interface traits.

Thus, as we continue this project and refactor even more of the collection hierarchy, we expect to see an *increasing* percentage reduction in code size as we apply the same reusable traits to remove code from more and more classes. In addition, the availability of these traits frees the implementors of the hierarchy from the need to ship pre-built collection classes for rarely used special cases. Instead, the main responsibilities of the implementor of the hierarchy become the implementation of (1) a basic class hierarchy that contains the more common collection classes and (2) a set of well-designed set of traits that can be composed with these classes. Using this basis, another programmer can then easily recompose the traits in order to build her own special-purpose collections.

Another advantage of the new hierarchy is that some of the traits can be used *outside* of the collection hierarchy. As an example, the trait TEmptiness, which requires size and provides isEmpty, notEmpty, ifEmpty:, isEmptyOrNil ifEmpty: and ifNotEmpty:, can be used in *any* class that defines size. Similarly, the trait TEnumerationUI can be used to provide a class with 24 methods from the enumeration protocol, provided that it implement do:, emptyCopyofSameSize, and errorNotFound. Why is this important? We believe that much of the power of the object-oriented paradigm comes from having many *different* objects understand the *same* protocol in corresponding ways. For example, it can be quite frustrating to find that a SoundBuffer, although it understands size and isEmpty, does *not* understand ifEmpty:. The availability of fine-grained traits at last makes it possible to make protocols more uniform across all of the classes in the system, with no cost in code size or maintainability, and with a *reduction* in the effort required to find ones way around the system.

7.4 Discussion

The availability of both trait composition and single inheritance gave us a lot of freedom in designing the new collection hierarchy. One possible approach would have been to use trait composition exclusively and to minimize the use of inheritance. If we had done this, all the concrete collection classes would have been built using trait composition alone, and every collection class would be a direct subclass of Object (or of an empty common superclass Collection that is a direct subclass of object).

However, we decided against this approach and used both single inheritance and trait composition in the new hierarchy. This makes the hierarchy easier to understand and extend by programmers who are familiar with single inheritance code, and especially for programmers who know the old collection hierarchy. Indeed, looking at our hierarchy in the flattened view, it exhibits a structure quite similar to the old hierarchy, although the abstract superclasses do not correspond one-to-one.

Single-inheritance also turns out to be well-suited for explicitly representing a functional property layer with abstract classes and a implementation layer with concrete classes. This is particularly true because the separation between functional methods and implementation methods is not always very clear. For example, it is sometimes the case that a particular implementation trait defines a optimized variant of a method that is generically defined in a functional trait. Because the concrete classes are *composed* from the implementation traits but *inherit* from the functional traits, we can be sure that in these situations the implementation methods override the functional methods. Thus there are no method conflicts, and therefore no need to resolve them.

8 Lessons Learned

While conducting this experiment we learned a number of things about traits and our programming tools, and also some more general things about refactoring.

Traits Simplify Refactoring. Using traits, refactoring a major hierarchy such as the Smalltalk collections is not as hard a task as one might think. We are not wizards, and we did not have a very clear plan, when we started, of where we would end up. Instead, we just started pair programming, doing the simplest thing that could possibly work, until we found that it didn't work — at which point we did something just slightly more sophisticated.

When we started dragging methods out of existing classes and dropping them into traits, it was quite easy to identity the necessary traits. We had a superficial familiarity with the Smalltalk collection classes, and had re-read Cook's 1992 study[Coo92]. So we expected to find traits related to the five different implementations and the major categories of functionality described in section 5.1. When we found a method that did not seem to fit into one of the traits that we had already defined, we simply created a new trait. Often, the hardest part was finding appropriate names for the traits; this is important and difficult, and the naming scheme used in this paper can surely be improved upon, even though it represents our third or fourth attempt.

Tools are Important. During the refactoring project, both the standard Smalltalk programming tools (which allow one to look at not just classes but also all the implementors of and senders of a particular message) and the trait specific tools (abstracting away from instance variables, viewing unsatisfied requirements, being able to move a method and have instance variable accesses automatically turn into message sends, *etc.*) turned out to be an enormous help. It was particularly useful to know that the nesting of traits does not change anything about the semantics of the methods. Thus, we could consider our refactoring task as simply grouping the existing collection behavior into coherent traits. For each of the newly constructed traits, the *requires* category in the browser always showed us which methods were missing in order to make the trait complete. Naturally, some of these missing methods belonged in other traits; we simply continued adding methods to the trait until *all* of the unsatisfied requirements belonged in other traits.

Use Fine-grained Components. As our refactoring progressed, we realized that the methods in the collection hierarchy could be grouped into traits at a much finer granularity than we had initially thought. Given good tools, traits do not impose any cost for the finer-grained structure: we didn't have to make the trade-off between the elegance of the implementation and the understandability and usability of the functional interface that characterizes both mixins and multiple inheritance.

Defer the design of the Class Hierarchy. Getting a class hierarchy "right" is known to be hard. The problem is that one is usually forced to make decisions too early, before enough is known about the implementation.

Our response was to put off making decisions for as long as possible, which turned out to be almost to the end of the refactoring. Since this was our first large project using traits, we had not yet developed a very clear feeling for the optimal balance between trait composition and inheritance. However, the theoretical properties of traits made us confident that things would turn out well in the end, provided that we collected behavior into logically coherent traits. Whether these traits would eventually be combined into complete classes or be used to build a deep hierarchy of abstract and concrete classes did not matter, because we knew that trait composition and inheritance could be freely combined.

Once we had built the first few implementation and interface traits, it became obvious how to combine them. The more we combined traits, the more important the flattening property became. However, we also realized the importance of the structured view, because it shows the traits from which a class is composed and how they are interconnected.

To summarize: we were able to put off the hard decisions until we knew enough about the system to make them correctly. This was because of the combination of

- a language technology with the right properties (flattening), and
- a set of tools that exploited those properties to provide multiple views on the program.

9 Related Work

The work presented here was inspired in part by Cook's study of interface conformance and implementation inheritance in the Smalltalk-80 collection classes [Coo92]. Cook first extracts an interface hierarchy based on conformance[BHJ$^+$87,Car88] between the sets of public methods of the various classes. Then, to solve problems raised by messages being interpreted differently in different classes, he writes formal specifications for the methods and corrects some method names. Cook's results shows that there is a wide divergence between the inheritance mechanism used to build the hierarchy and the conformance relationship between the interfaces.

Our work is complementary to Cook's. We did not attempt to merge the implementation and conformance hierarchies. Instead we moved almost all of the implementation into traits, where it can be widely reused; this frees the inheritance hierarchy to capture conformance.

Few other workers have reported measurements of the impact of mixin-like abstractions on non-trivial class hierarchies. Moore reports on the use of a "self improvement" tool called Guru, which automatically restructures inheritance hierarchies and also refactors the methods in Self programs [Moo96]. Moore applies Guru to the indexables, a fragment of the Self library that includes strings, vectors and sequences, and which contains 17 objects (most of which play the role of classes). The restructured version of the hierarchy reduced the number of methods from 316 to 311, and the number of overridden methods from 86 to 72. However, his method-level refactoring introduced 79 additional methods.

Moore's analysis finds some of the same problems with inheritance that we have described in this paper, and also notes that sometimes it is necessary to manually move a method higher in the hierarchy to obtain maximal reuse. Our work differs from Moore's in that he uses a tool to *automatically* restructure and refactor inheritance hierarchies, whereas we developed a new language concept and associated tools to *support the programmer* in writing better (*e.g.*, less duplicative) and more reusable code in the first place. Our focus is on improving understandability; Moore's approach, used by itself, may have a negative impact on understandability, because it introduces methods with generated names. However, it would be very interesting to adapt the techniques used in Guru to help the programmer identify traits by, for example, identifying duplication in an existing hierarchy.

Our work also shares some similarity with research efforts in hierarchy reorganization and refactoring. Casais [Cas91,Cas92] proposes algorithms for automatically reorganizing class hierarchies. These algorithms not only help in handling modifications to libraries of software components, but they also provide guidance for detecting and correcting improper class modelling. Dick *et al.* propose a new algorithm to insert clasess into a hierarchy that takes into account overridden and overloaded methods [DDHL96]. The key difference from the results presented here is that all the work on hierarchy reorganization focuses on transforming hierarchies using inheritance as the only tool. In contrast, we are interested in exploring other mechanisms, such as composition, in the context of mixin-like language abstractions.

Refactorings — behaviour preserving code transformations — have become an important topic in the object-oriented reengineering community [RBJ97,TB99,TDDN00]. Research on refactoring originates from the seminal work of Opdyke [Opd92] in which he defined refactorings for C++ [JO93,OJ93]. In this context, Tokuda and Batory [TB99] evaluate the impact of using a refactoring engine in C++. Fanta and Rajlich report on a reengineering experience where dedicated tools for the refactoring of C++ were developed [FR98]. However, they do not analyse the two versions of their code to compare the degree of reuse.

Some other languages do have constructs similar to traits, although they differ in some important details, which we believe limit reuse compared to traits. (A study of this issue is available in a companion paper [SDNB02a]). The language Self [US87] even uses the name "traits", although Self traits are basically objects that play the role of method dictionaries shared by prototypes. Strongtalk is a typed version of Smalltalk implemented using mixins at a deep level. However, to the best of our knowledge there has been no scientific study evaluating the level of code reuse engendered by such approaches.

10 Future work

This paper is a report of work in progress. We plan to continue the refactoring effort, including more of the collection classes and some of the stream classes, and see if the further reduction in code size that we have predicted actually materializes. We would also like to conduct a more sophisticated analysis of code duplication, perhaps using a tool such as Duploc [DRD99] or Guru (see section 9).

A thorough testing of the new collection classes is also necessary. We believe that this can be accomplished using a random test generator and the existing classes as a test oracle.

We are very pleased with how well the tools that we have built actually support the process of programming with traits. There are naturally a few missing features. For example, when examining a class in the traits browser, its subtraits are immediately visible, but the subtraits of the subtraits are not. Examining the sub-subtraits requires a double-click and a switch to another context. It would be better if the browser supported hierarchic lists that could be opened and closed at any level. In addition, some of the ordinary Smalltalk tools need to be enhanced to deal with traits. We plan to continue our tool development, and also to add "trait awareness" to some of the standard Smalltalk programming tools.

We also expect our experience with traits to feed back into the design of traits themselves. We do not consider traits as just another programming construct, but as an enabling technology for a grander vision called multi-view programming, which is gradually being realized in the Perspectives project [BJ00].

Refactorings that can be expressed in a language can be thought of as defining equivalence relations; each such equivalence has the potential to raise the level of abstraction of the programming process to that of the induced equivalence classes. A new language technology like traits, which greatly extends the range of possible refactorings, also defines a new set of equivalence classes, and thus permits programming at a more abstract and powerful level. However, a fuller discussion of these possibilities is outside of the scope of this paper.

11 Conclusion

We undertook this refactoring primarily to obtain some practical experience with the use of traits. We believed that the theoretical properties that we had given to traits — especially flattening, but also the annihilation of conflicting methods in the symmetric sum — were the right ones. But programing languages are too, and theoretical elegance is no substitute of usability. Only extensive use on a realistic codebase could validate these beliefs.

It did. We were even surprised with how well the tools and the trait concepts worked in practice. The theoretical characteristics do really seem have the practical benefits for which they were designed.

However wonderful a language technology may be to those who use it, new language features can be a real obstacle to those who have not previously met it. One of the pleasant properties of traits is that we took great care not to change the method-level syntax of Smalltalk at all. Thus, an ordinary Smalltalk programmer can open an ordinary Smalltalk browser on our new hierarchy and understand everything that she sees. All of the concrete classes will be there, with all of their methods. The methods will appear to be defined directly in a subclass or inherited from a superclass exactly as in ordinary Smalltalk, and the semantics will be exactly the same. If a method is modified in a conventional class view, and the method is actually defined in a shared trait, then the effect will be to to define a customised version of the shared method local to the class. Again, this is exactly the semantics of ordinary Smalltalk.

This property is critically important, because we believe that one of the reasons that previous technologies such as mixins and multiple inheritance have not become popular is because of the complexity that they force on every programmer. For example, the rules for linearising multiple inheritance chains must be understood by every programmer who looks at or modifies a multiple inheritance hierarchy.

While these results validated our expectations, there were also some surprises. For example, it turned out that the fine-grained nesting structure has only advantages. Not only does it allow better code reuse, but it also assists in program understanding, because it makes it easier to see how something is built up. However, this is only true as long as the flattened view is also available. We are not experts in human perception, but all the evidence that we have seen indicates that humans grasp things more quickly and more accurately if they can observe them though different views.

Another surprise was that the refactoring process turned out to be quite enjoyable and very straightforward. Trait-based refactoring seems to compatible with an extreme programming style of development, because it does not require one to do all of the design "up front", when one knows nothing about the system, but lets one start by identifying related methods and putting them into traits. The shape of the inheritance hierarchy can emerge later.

Good tool support proved to be critical: it had a tremendous impact on the efficiency of the refactoring task. It is hard to imagine undertaking this refactoring with an ordinary Smalltalk browser that does not show the *requires* and *supplies* sets and that does not support the abstract instance variable refactoring. Performing the same task with with a traditional file-based tool such as emacs is inconceivable to us. The incremental nature of the Smalltalk environment played an important role, because the current state of the composition was instantly visible at all times.

To summarise: we have successfully refactored a significant fragment of the Smalltalk collections classes. In the process we:

- removed code duplication;
- improved understandability;
- provided reusable traits that make it easier to write new collection classes; and
- made it possible to reuse collection code *outside* of the collection hierarchy.

The second claim, improved understandability, is necessarily subjective. However, we argue that two *objective* features of the refactored hierarchy support it. First, there is no discrepancy between the apparent and actual interfaces of a class. In other words, we never needed to resort to implementing a method "too high" in the hierarchy just to enable reuse. As a consequence, when browsing the hierarchy, "what you see is what you get": all of the public methods in a class are actually available. Second, the structured view (with fine grained traits) provides a lot of insight about the functional properties of the methods: which are immutable, which require sequenceability, which do enumeration and so on. Since the structured view containing this extra information is optional, there is no tradeoff to be made in supplying it: programmers who do not find it useful can simply not use it.

Acknowledgements This work was initiated during a sabbatical visit by Andrew Black to the University of Bern, and continued during a visit by Nathanael Schärli to OGI. We would like to thank Oscar Nierstrasz and the other members of the Software Composition Group in Bern for making the sabbatical possible, and for being such intellectually stimulating and congenial hosts. We also thank the National Science Foundation and the late Professor Paul Clayton, then Provost of the Oregon Graduate Institute, for the financial support that made the visits possible.

References

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.

[BDMN73] G. Birtwistle, Ole Johan Dahl, B. Myhrtag, and Kristen Nygaard. *Simula Begin*. Auerbach Press, Philadelphia, 1973.

[BHJ+87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract data types in emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[BJ00] Andrew P. Black and Mark P. Jones. Perspectives on software. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-oriented Systems*, 2000.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Cas91] Eduardo Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach*. Ph.D. thesis, Centre Universitaire d'Informatique, University of Geneva, May 1991.

[Cas92] Eduardo Casais. An incremental class reorganization approach. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92*, volume 615 of *LNCS*, pages 114–132, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[Coo92] William R. Cook. Interfaces and specifications for the smalltalk-80 collection classes. In *Proceedings OOPSLA '92*, pages 1–15, October 1992.

[DDHL96] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. In *Proceedings of OOPSLA'96*, pages 251–267, 1996.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, September 1999.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[FR98] Richard Fanta and Vaclav Rajlich. Reengineering object-oriented code. In *Proceedings of the International Conference on Software Maintenance*, 1998.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison Wesley, 2000.

[JO93] Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. Springer-Verlag, November 1993.

[LP90] Wilf LaLonde and John Pugh. *Inside Smalltalk: Volume 1*. Prentice Hall, 1990.

[Moo96] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of OOPSLA '96*, pages 235–250. ACM Press, 1996.

[OJ93] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of the 1993 ACM Conference on Computer Science*, pages 66–73. ACM Press, 1993.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[SDNB02a] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. Technical Report CSE-02-012, OGI School of Science & Engineering, Oregon Health & Science University, Beaverton, OR 97006, USA, 2002. Submitted for publication, ECOOP 2003.

[SDNB02b] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: The formal model. Technical Report CSE-02-013, OGI School of Science & Engineering, Oregon Health & Science University, Beaverton, OR 97006, USA, 2002.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[TB99] Lance Tokuda and Don Batory. Automating three modes of evolution for object-oriented software architecture. In *Proceedings COOTS'99*, May 1999.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*, pages 157–167. IEEE, 2000.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87*, volume 22, pages 227–242, December 1987.