Applying Traits to the Smalltalk Collection Classes*

Andrew P. Black
OGI School of Science & Engineering, Oregon Health and Science University, USA
black@cse.ogi.edu

Nathanael Schärli
Software Composition Group
University of Bern
Switzerland
schaerli@iam.unibe.ch

Stéphane Ducasse
Software Composition Group
University of Bern
Switzerland
ducasse@iam.unibe.ch

ABSTRACT

Traits are a programming language technology that promote the reuse of methods between unrelated classes. This paper reports on a refactoring of the Smalltalk collections classes using traits. The original collection classes contained much duplication of code; traits let us remove all of it. We also found places where the protocols of the collections lacked uniformity; traits allowed us to correct these non-uniformities *without* code duplication. Traits also make it possible to reuse fragments 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 number of methods in the collection classes by approximately 10 per cent. More importantly, understandability maintainability and reusability of the code were significantly improved.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Classes and objects; D.2.3 [**Coding Tools and Techniques**]: Object-oriented programming

General Terms

Design, Languages

Keywords

Reuse, Mixins, Traits, Smalltalk, Collection hierarchy, Stream classes, Refactoring, Inheritance, Multiple Inheritance

1. INTRODUCTION

We have long believed that classes have too many responsibilities in object-oriented programming [3]. In many languages, classes are the prime (or only) mechanism for the conceptual classification of objects. Classes also provide for reuse in two different ways: as factories, they can be used to instantiate many similar objects, while as superclasses, they group methods so that they can be incorporated into new subclasses. These two kinds of reuse often have conflicting requirements. As factories, classes must be complete; when creating new subclasses, it is more convenient to be able to incorporate incomplete 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 the problems of code duplication, unnecessary inheritance, and method redefinition in inheritance chains;
- a report of our experience using traits to refactor the collection classes so as to remove these problems; and
- a description of the organization of the new classes that resulted from our refactoring.

2. WHAT IS THE PROBLEM?

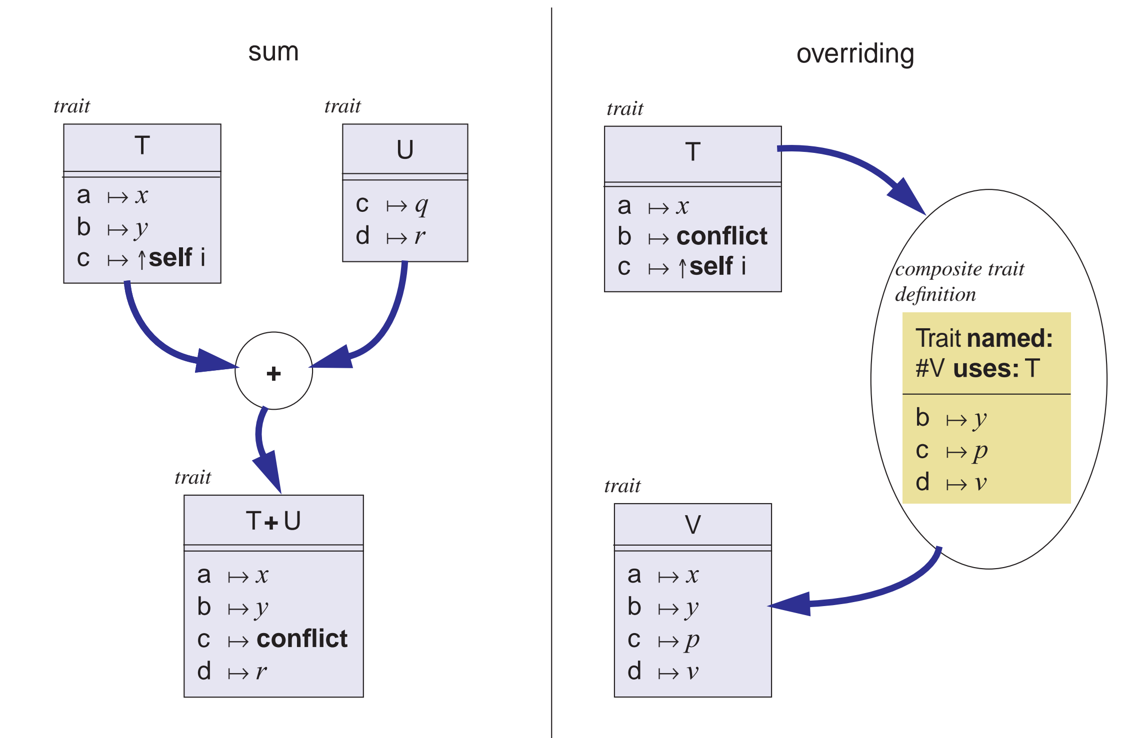
Single inheritance is a very popular programming technology, and has been adopted widely since its introduction in Simula 67 [2]. Inheritance is also very powerful, and the basis for several major success stories, including Smalltalk and Java. But success and power should not blind us to the fact that sometimes single 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.

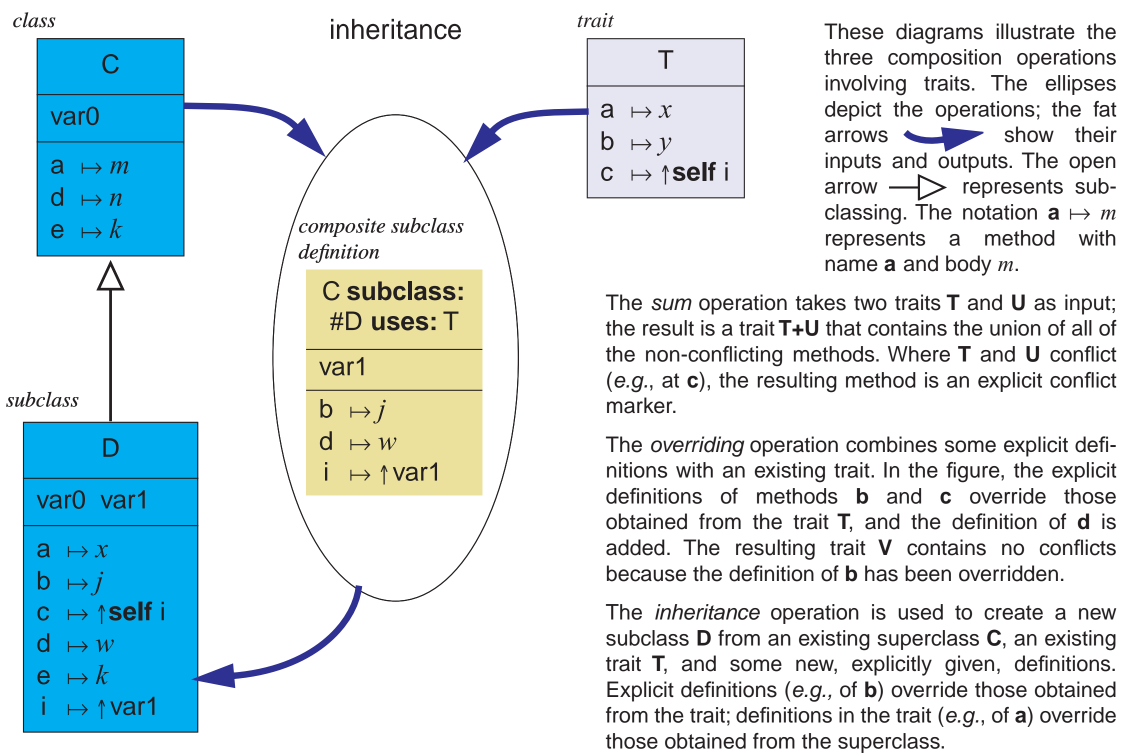
Let us illustrate this point with a small example. The class RectangleMorph in the Squeak dialect of Smalltalk represents a rectangular block of color that can be manipulated on the display. As such, it is a subclass of BorderedMorph, from which it inherits and reuses many methods; BorderedMorph in turn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

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 named bounds that represents the Rectangle that it occupies.

*This research was partially supported by the National Science Foundation of the United States under awards CDA-9703218, CCR-0098323 and CCR-0313401, and by Swiss National Foundation project 2000-067855.02.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'03, pp. 47–64, October 26–30, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-712-5/03/0010 ...$5.00.





These diagrams illustrate the three composition operations involving traits. The ellipses depict the operations; the fat arrows show their inputs and outputs. The open arrow represents subclassing. The notation **a** $\mapsto m$ represents a method with name **a** and body m.

The *sum* operation takes two traits **T** and **U** as input; the result is a trait **T+U** that contains the union of all of the non-conflicting methods. Where **T** and **U** conflict (*e.g.*, at **c**), the resulting method is an explicit conflict marker.

The *overriding* operation combines some explicit definitions with an existing trait. In the figure, the explicit definitions of methods **b** and **c** override those obtained from the trait **T**, and the definition of **d** is added. The resulting trait **V** contains no conflicts because the definition of **b** has been overridden.

The *inheritance* operation is used to create a new subclass **D** from an existing superclass **C**, an existing trait **T**, and some new, explicitly given, definitions. Explicit definitions (*e.g.*, of **b**) override those obtained from the trait; definitions in the trait (*e.g.*, of **a**) override those obtained from the superclass.

Figure 1: The major operations on traits: sum, overriding, and inheritance.

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 [20].

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 increases the size of the object code and makes RectangleMorph harder to understand because it adds a lot of conceptually uninteresting"noise". Constructing all 70 delegation methods requires tool support; using them introduces run-time overhead.

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 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 [30] 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?

A trait is a first-class collection of named methods. The methods must be "pure behavior"; they cannot directly reference any instance variables. The purpose of a trait is to be composed into other traits and eventually into classes. A trait has no superclass; if the keyword **super** is used in a trait method, it remains unbound until the trait is eventually used in a class.

Mixins are also collections of methods intended to be used as components of classes [6, 15]. The major difference between mixins and traits is that whereas mixins must be applied to classes one at a time using the inheritance operation, traits are subject to a richer set of composition operators, giving the programmer more freedom in the way that they can be combined. Figure 1 illustrates the sum, overriding and inheritance operations. The sum trait T + U contains all of the non-conflicting methods of T and U. However, if there is a *conflict*, that is, if T and U both define a method with the same name (as they they do for c in the figure), then in T + U that name is bound to a distinguished **conflict** method. The + operation is associative and commutative.

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

The overriding operation constructs a new composite Trait **named:** #V **uses:** T ... by extending an existing trait T with some explicit local definitions. The local definitions override methods of the same name in the trait T; because the local definitions are explicit, it is always clear what is being overridden. Thus, in the figure, we see that the method for c with body p overrides the one with body ↑ **self** i.

Traits are incorporated into classes by means of an extended form of inheritance. In conventional inheritance, a new subclass B is constructed from a superclass C and from some local definitions of instance variables and methods. In Smaltalk we would write C **subclass:** #B ... and then add the local definitions to B. Our extended form of inheritance is similar; a new subclass D is constructed from a superclass and a trait T in addition to the local definitions. In Smalltalk we write C **subclass:** #D **uses:** T .... We call D a *composite class*. Methods defined locally in as part of the inheritance operation (such as b in figure 1) replace methods obtained from the trait T. Methods defined locally (such as d) and methods from T (such as a) both override methods inherited from the superclass C, and can access the overridden methods using **super**. In practice, the trait that is used to build a composite class is often the sum of several more primitive traits.

The aliasing operator @ creates a new trait by providing an additional name for an existing method. For example, if U is a trait that defines methods on c and d, then U@{e → c} is a trait that defines methods for c, d *and* e. The additional method e has the same body as the old method c. Aliases are used to make conflicting methods available under another name, perhaps to meet the requirements of some other trait, or to avoid overriding. Note that because the body of the aliased method is not changed in any way, an alias to a recursive method is not recursive.

Finally, a trait can be constructed by *excluding* methods from an existing trait using the exclusion operator −. Thus, U − {c} has a single method d. Exclusion is used to avoid conflicts, or if one needs to reuse a trait that is "too big" for one's application.

Associated with each trait is a set of *required* messages. Any concrete class that uses a trait must provide methods for all of the required messages. 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 that is intended to be concrete, size will have to be defined, perhaps as a method that fetches the value of an instance variable, or perhaps as a method that calculates the size.

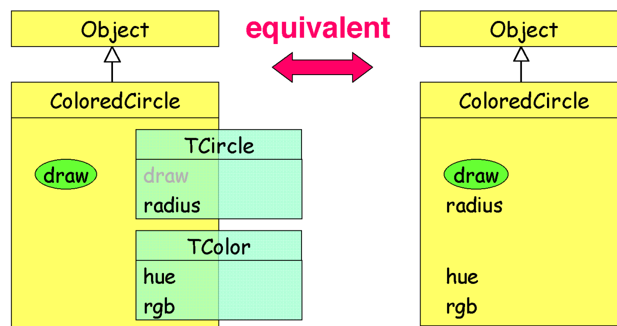


Figure 2: Flattening. The class ColoredCircle on the left supplies an overriding local definition of draw. The bodies of the corresponding methods are identical on the left and the right.

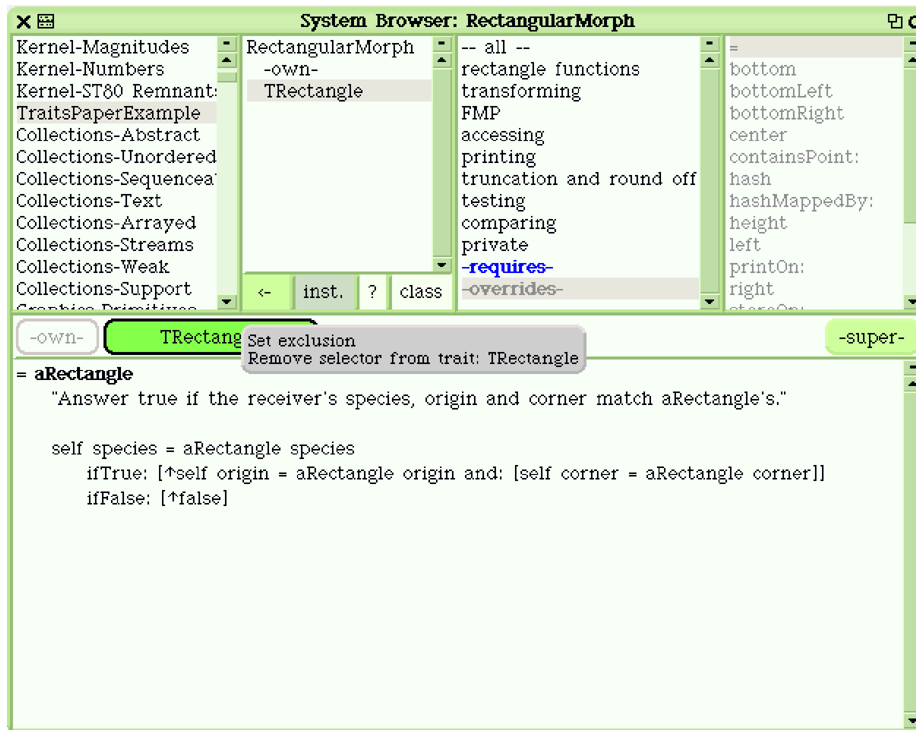


Figure 3: The programmer examines the list of overrides created when RectangularMorph uses the trait TRectangle as a component. The = method from TRectangle is inappropriate, and is excluded.

Because of the way that we define the operations on traits, the semantics of a method is independent of whether it is defined in a trait T, or in a class (or a trait) that uses T as a component. Consequently, provided that all trait conflicts have been resolved, it is always possible to convert a program that uses traits into an equivalent program that uses only ordinary classes, at the cost of possible code duplication. We call this process *flattening*; it is similar to inlining of procedures or expansion of macros. Similarly, a (conflict-free) composite trait can always be flattened into a simple trait. Flatening is illustrated in figure 2: the composite class ColoredCircle on the left, composed from two traits and a local definition for draw, is semantically equivalent to the flat class ColoredCircle on the right.

It is important that flattening never requires the bodies of the methods to be modified. Thus, a complex composite entity can be viewed as such; it can be completely flattened to remove all the internal structure; or it can be viewed at any partially structured point in between these extremes. We believe that the *flattening property* is crucial in making traits easy to use; it is another critical differences between traits and mixins.

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 [28, 29].

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 shows how a class can be transformed into a trait, illustrates the difficulties that may be encountered when using the trait, and gives a glimpse of 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, which is 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 [25] calls it “abstract access to member variable”, and Fowler [16] calls it “encapsulate field”. We follow the lead of the Refactoring Browser [27] by using the name “abstract variable”.

Once this refactoring is completed, the new trait has all of Rectangle's methods, but those methods 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 the methods in trait TRectangle, and also shows its requires set, which comprises the three messages origin, corner and species.

We can now use the browser to construct RectangularMorph as a new subclass of RectangleMorph. The process is similar to creating a new subclass in the ordinary Smalltalk browser, but uses the extended form of inheritance that lets us specify a trait (or a trait-valued expression) that will be *used* as a component. We define

```
RectangleMorph subclass: #RectangularMorph
        uses: TRectangle
        . . .
```

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

```
RectangularMorph>>origin
    ↑ self bounds origin

RectangularMorph>>corner
    ↑ self bounds corner
```

The requirement species has already been satisfied, because a species method is provided by our superclass RectangleMorph.

The browser also shows us a list of *overridden* methods, that is, places where the new class RectangularMorph defines a method that would otherwise have been inherited from RectangleMorph. An example is center, which is present in the trait TRectangle but which is also defined in RectangleMorph. Each overridden method must be examined to see whether it is appropriate to keep the version from TRectangle, to keep the version from the superclass, or to write a new method. Buttons in the browser let us examine both of the alternatives and make our choice quickly.

In the case of center and most of the other overrides, the methods from TRectangle are appropriate, because they use self origin and self corner to access the rectangle's coordinates. However, TRectangle's methods for =, hash, and printOn: are inappropriate. We use a browser menu to exclude these methods from RectangularMorph. We choose to do this by *setting an exclusion* (see figure 3), the effect of which is to modify the definition of RectangularMorph so that certain methods from TRectangle are never used; the equivalent declaration would be

```
RectangleMorph subclass: #RectangularMorph
        uses: TRectangle − {#=. #hash. #printOn:}
        . . .
```

Throughout this process, the traits browser helps us to focus on just those methods that require our attention. Once we have examined all of the overrides and satisfied all of the requirements, our task is complete: we have created a new class that has the functionality of both 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.

Small examples like this are fun, but they are not by themselves a compelling test of a language extension that is intended to improve the structure of large class libraries. To evaluate traits in a realistic setting, they must be applied to a framework of significant size. We chose the Smalltalk collections classes as the target for such an evaluation.

5. THE SMALLTALK COLLECTION CLASSES

The collection classes are a loosely defined group of general purpose subclasses of Collection and Stream. The group of classes that appears in the "Blue Book"[18] 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 and hence not categorized as "Collections" by the system organization. For the purposes of this study, we use the term "Collection Hierarchy" to mean Collection and its 37 subclasses that are *also* in the system category Collections. We use the term "Stream Hierarchy" to mean Stream and its 10 subclasses that are *also* in the system category Collections. The full list is shown in figure 4.

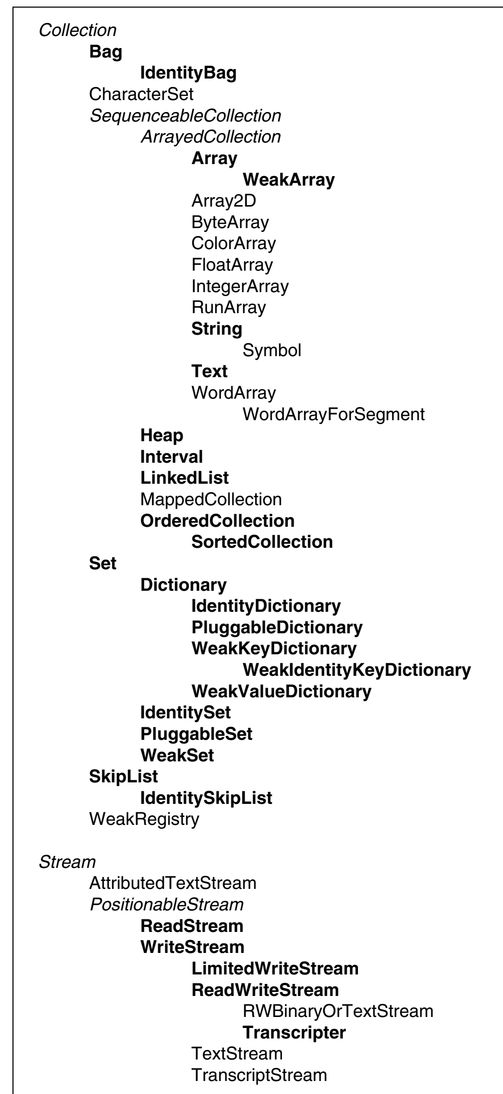


Figure 4: The collection classes in Squeak. Indentation indicates subclassing. Abstract classes are *Italicized*; the classes that we refactored are named in Bold.

These 49 classes respond to 794 messages and define a total of 1236 methods.

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. Those familiar with the Smalltalk collection classes 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. Suppose you have a data structure containing a collection of student records, and wish to perform some action on all of the students that have a particular property. 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 select: 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 select: 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.

Beyond this basic uniformity, there are many different kinds of collection. For example, some collections are sequenceable, that is, an enumeration of the collection starts from a first element and proceeds in a well-defined order to a last element. Array and LinkedList are examples (see figure 5). Array and many other collections are also indexable, that is, anArray at: n retrieves the nth element of anArray, and anArray at: n put: v changes the nth element to v. However, LinkedList, although sequenceable, is not indexable, that is, its instances they understand first and last, but not at:.

The class OrderedCollection is more general than Array; 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) or mutable (the others)?
4. Is the collection constrained to hold a particular kind of object, or is it completely general? For example, LinkedLists are constrained to hold elements that conform to the Link protocol, while CharacterArrays, Strings and Symbols must contain characters. [4]

The non-sequenceable 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, *e.g.*, what test is used to ascertain whether two elements added to a set are "equal"?

These categorizations by functionality are not our only concern; as re-implementors of the collection hierarchy we must also understand how the collection classes are implemented. As shown in Figure 6, 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. Consequently, 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 values.
4. LinkedLists use a standard singly-linked representation.
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 [22].

5.2 Streams

The collection protocol supports the storage, removal and enumeration of the elements of a collection, but does not allow these operations to be intermingled. For example, if the elements of an OrderedCollection are processed by a do: method, it is not possible to add or remove elements from inside the do block. Nor does the collection protocol allow us to perform a merge sort by sequencing through two OrderedCollections and repeatedly removing the smallest first element. Procedures like these require that a traversal index or position reference is maintained outside of the collection itself, as captured in the Iterator pattern [17]. Smalltalk Streams perform exactly this function. All stream objects are defined to

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

[4] A Symbol is a unique immutable string, used heavily in the language implementation. The unique instance property means that equality tests are particularly efficient.

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

Figure 5: 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 LinkedLists **and** SkipLists **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.**

Arrayed Implementation	Ordered Implementation	Hashed Implementation	Linked Implementation	Interval Implementation
Array String Symbol	OrderedCollection SortedCollection Text Heap	Set IdentitySet PluggableSet Bag IdentityBag Dictionary IdentityDictionary PluggableDictionary	LinkedList SkipList	Interval

Figure 6: Some collection classes categorized by implementation technique.

stream over some collection. For example:

```
r := ReadStream on: (1 to: 10).
r next.          prints 1
r next.          prints 2
r atEnd.         prints false
```

WriteStreams are analogous:

```
w := WriteStream on: (String new: 5).
w nextPut: $a
w nextPut: $b
w contents.      prints 'ab'
```

It is also possible to create ReadWriteStreams that support both the reading and the writing protocols; defining such a class without code duplication is a challenge for single inheritance. Squeak chooses to make ReadWriteStream a subclass of WriteStream, as shown in figure 4.

6. ANALYSIS OF THE COLLECTION CLASSES

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

Given the many dimensions in 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 showed[11], 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 presents a very different interface.

Another way that the designers of the hierarchy attempted to maximize reuse was to move methods high up, so that all possible 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

Inheritance is used quite heavily in the collection classes, mostly for sharing implementation, but also for classification [11]. As a measure of the complexity of the inheritance relationships, we counted the number of inheritance chains in which a method is defined three or more times. Figure 7 illustrates two examples: the methods at:put: and add: in the inheritance chains terminating in the class WeakValueDictionary. We found 79 such inheritance chains.

There is nothing intrinsically wrong with redefining 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

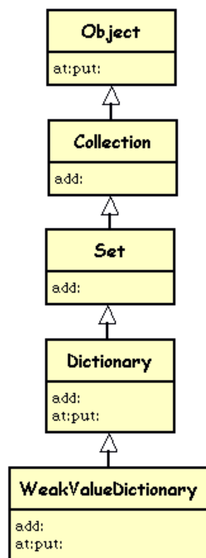


Figure 7: Redefinition of at:put: **and** add: **in the superclasses of** WeakValueDictionary

parameterize the behavior of a template method [1]: 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 to assist us in understanding legacy code.

6.2 Code Duplication

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 nature of single inheritance. For example, PluggableSet and PluggableDictionary share some methods, but there is no place from which both classes 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 (in Set), or duplicating it.

The Stream classes provide a classic example of methods being implemented too high. Conceptually, a PositionableStream is an accessor for a sequence of objects named by external indices (such as characters in a string or a file). The *positionable* protocol includes messages to set and reset the position of this index. However, PositionableStream is also the lowest common superclass of both ReadStream and WriteStream. Because ReadWriteStream is a subclass of WriteStream (see figure 4), many of the methods that support reading (for example, reader next: 4 into: anotherCollection, and reader nextDelimited: $.) are implemented in PositionableStream in terms of self next.

The method next itself is explicitly disabled in WriteStream, and re-enabled (and duplicated) in ReadWriteStream and ReadStream. The other "reading" methods in PositionableStream are *not* explicitly disabled in WriteStream, but are inherited by ReadStream and ReadWriteStream. Thus, to avoid code duplication, methods specific to reading are implemented in the superclass of WriteStream. The tactic succeeds (except in the case of next itself), but the price is high: PositionableStream is polluted by many methods that have nothing to do with positioning, and WriteStream appears to implement many reading methods, although these methods will fail if they are ever used.

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 check detected as duplicates methods those that differed only in formatting, comments, or the names of temporary variables. We excluded from our count error methods such as self shouldNotImplement, which is used to cancel an inherited method. Using this check we found 28 pairs of duplicated methods, and 3 triples. 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 method from Collection.

However, these duplication counts are 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. In addition, there is also undoubtably duplication of collection code in classes outside of the collection hierarchy, which we did not attempt to quantify.

6.3 Conceptual Shortcomings

In addition to the above implementation problems, the collection classes also suffer 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 uses equality (=) between elements, one that uses identity (==), 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 two special cases: Symbols and Intervals. Nevertheless, 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 collection classes.

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

Another problem with the collection hierarchy, also observed by Cook, is that sometimes the interfaces of the classes are not what one would expect: certain methods are missing. The classes String and Text provided an example. String adds 142 new methods to the protocol of its superclass (the abstract class ArrayedCollection). Most of these methods are related to parsing, converting to HTML, searching for substrings and regular expressions, and other operations specific to character strings, and so inevitably these methods must be defined specifically for the class String. However, Squeak also defines a class Text, for representing character strings that have been attributed with font changes, hyperlining, *etc.* All 142 String messages ought to be understood by Text objects, but in the standard Squeak system, only 15 of them actually are. The remaining 127 are missing. Why is this? Text is not a subclass of String, so the "missing" methods cannot be inherited; the situation is similar to the problem with RectangleMorph and Rectangle described in section 2. Fixing this problem in Squeak would require either code duplication or 127 delegation methods.

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 re-implement them, and the existing implementation cannot be reused. Section 7.4 discusses how traits make collection-like behavior available outside of the collection hierarchy.

7. RESULTS

In this section we explain how traits are used in the collection hierarchy that emerged from our refactoring efforts. We start by describing how we distributed behavior from the pre-existing abstract and concrete 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. The source code for the whole hierarchy and the tools that produced it are available on the web at http://www.iam.unibe.ch/~schaerli/smalltalk/traits/OOPSLACollectionRefactoring.zip

7.1 The New Collections Hierarchy

Figure 8 shows the new hierarchy for the 23 common concrete collection classes that we have re-implemented, and 6 abstract superclasses. In addition to the name of each class, the figure also shows the traits from which the class is composed. The classes are divided into three layers. At the top of the figure 8 is the abstract class Collection, which is composed from two traits, and provides a small amount of behavior for all collections. Next 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 the *implementation* properties, that is, properties that characterize the internal data structures used in the implementation rather than the external behavior. Inheriting from the *functional* classes we have 23 concrete classes, each of which also uses one or more traits that specifies its implementation. We now describe the functional and the implementation traits in turn.

The Functional Traits.

Each kind of collection can be characterized by several 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. The various combinations of these properties can be represented by combining the respective traits. All that is necessary is to create a trait for each property (see figure 9) and then combine them to build the abstract classes of figure 8. In order to allow maximal reuse, we ensured that the combinations of property traits are available in two forms: as composite traits that can be reused outside of the collection hierarchy, and as superclasses that can be inherited within it.

We modularized the primitive properties more finely than would have been necessary if our only goal were to avoid code duplication. This fine structure gives us, and future programmers, 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 9 also 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 consisting 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 sets, 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 TEnumerationI—which uses TEnumerationUI as a subtrait—contains both methods that require sequencing and methods that do not. Furthermore, none of the methods in these traits treats the target object as mutable (I).

The Implementation Traits.

Besides the functional properties, which are visible to a client, each collection class is also characterized by an implementation, which

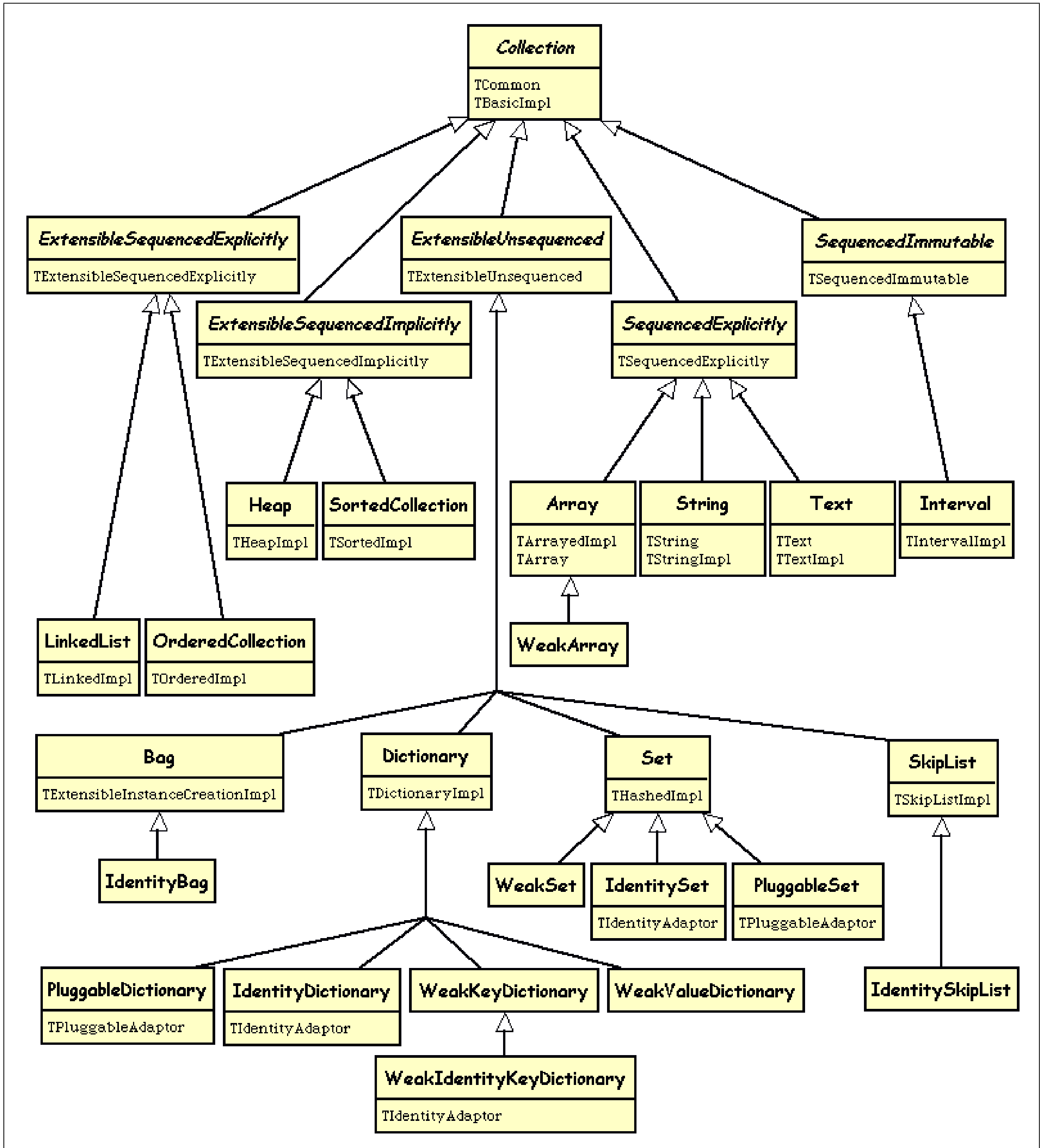


Figure 8: The refactored collection hierarchy. Classes with italicized names are abstract; below the class name we show the top-level trait(s) from which the class is composed. Each of these traits is in turn composed from several subtraits, as shown in figures 9 and 10. The names of implementation traits end in "impl". Individual methods and instance variables are not shown.

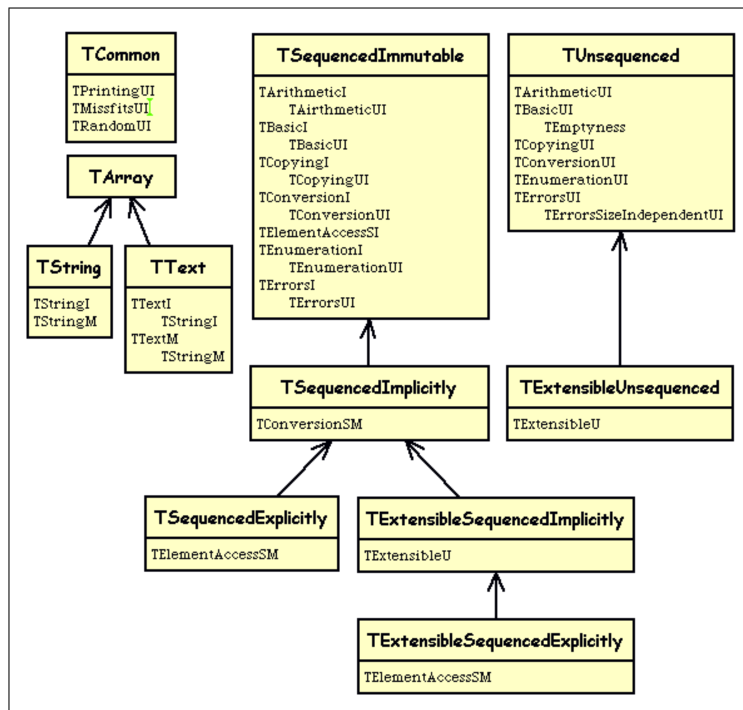


Figure 9: The traits corresponding to functional properties are built from each other and from more primitive traits. The 11 boxes represent the larger composite traits; subtrait relationships between them are shown by arrows. The remaining subtraits are listed in the bottom part of each box, using indentation to show sub-subtraits.

is normally hidden from a client. The functional and implementation traits that capture these properties are largely independent.

The refactored hierarchy separates the traits specifying the implementation of a collection from the traits specifying the functional properties. This allows us to combine the different functional property traits (*e.g.*, TExtensibleSequencedExplicitly and TExtensibleSequencedImplicitly) with any of the suitable implementations (*e.g.*, linked and array-based). In one place we also extended the functionality of a concrete class: our LinkedLists are indexable, *i.e.*, they understand at:.

Figures 10 and 11 show the structure of the implementation traits. The 9 implementation traits shown in figure 10 are "common" in the sense that they are used as components of several implementations. As an example, the behavior for creating new instances (new, with:, withAll:, *etc.*) is collected into the trait TInstanceCreationImpl, which is then used by TOrderedImpl and four other implementation traits. Each of the 12 traits in figure 11 captures the behavior of a specific concrete class, and is built from a combination of local methods and the common 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 in hashed collections includes: should be $O(1)$. Instead of using TBasicImpl as a subtrait of all the specific 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 implementations. For example, THashedImpl and TIntervalImpl have their own implementations of includes:.

7.2 The New Stream Hierarchy

The refactored version of the Stream hierarchy retains the abstract class PositionableStream from the standard Squeak hierarchy, but uses it only to capture the notion of positionability, and not as a place to put methods just so that they can be shared. Thus, the protocol of PositionableStream is reduced from 84 messages to 29. ReadWriteStream, which was formerly a subclass of WriteStream, is now a direct subclass of PositionableStream, and shares traits with both ReadStream and WriteStream, as shown in figure 12.

Seven traits are used to build the Stream classes; they have the simple structure shown in figure 13. TReadablePositionable and TWriteablePositionable actually add *different* sets of methods to TReadable and TWriteable because "positionability" means different things for readers and writers.

7.3 Measurements of the Refactored Classes

The refactored part of the collection class hierarchy shown in figure 8 contains 23 concrete classes and 6 abstract classes. These classes are built from a total of 52 traits. The Stream hierarchy shown in figure 12 contains 7 concrete classes and 2 abstract classes, which are composed from 15 traits.

The average number of traits used to build a class is more than 5; the maximum that we used in any one class is 22. Further statistics are presented in table 1.

Because String is something of an anomaly, containing many methods that do not appear elsewhere in the hierarchy (as explained in section 6.3), we initially excluded String and Text from our measurements (see the first column of table 1). Most of the numbers are self-explanatory. "Methods saved" is the difference between the number of methods in the original and the trait versions of the

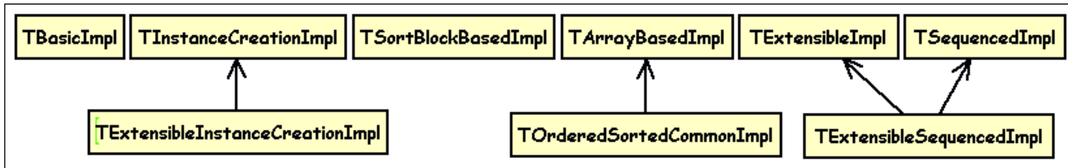


Figure 10: Common implementation traits. Each of these traits provides behavior common to more than one of the specific implementation traits shown in figure 11.

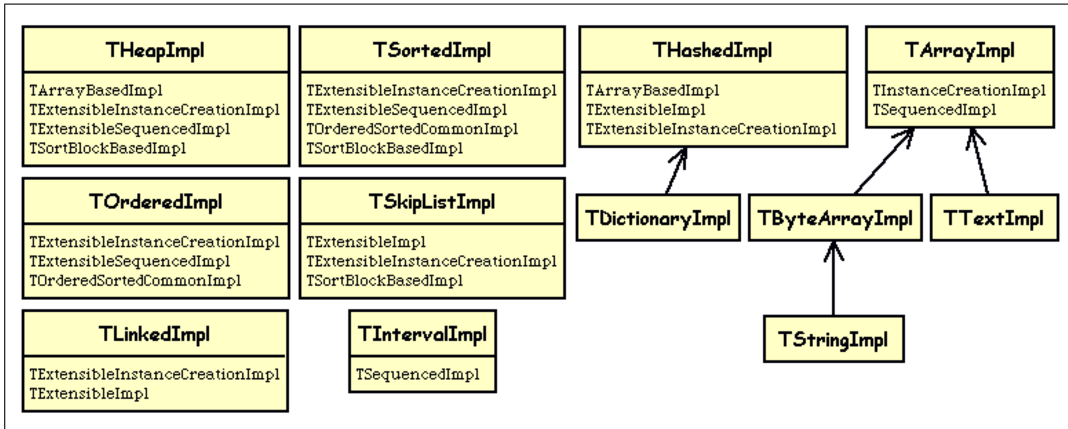


Figure 11: Specific implementation traits. Each of these traits supplies the implementation methods for a particular concrete class.

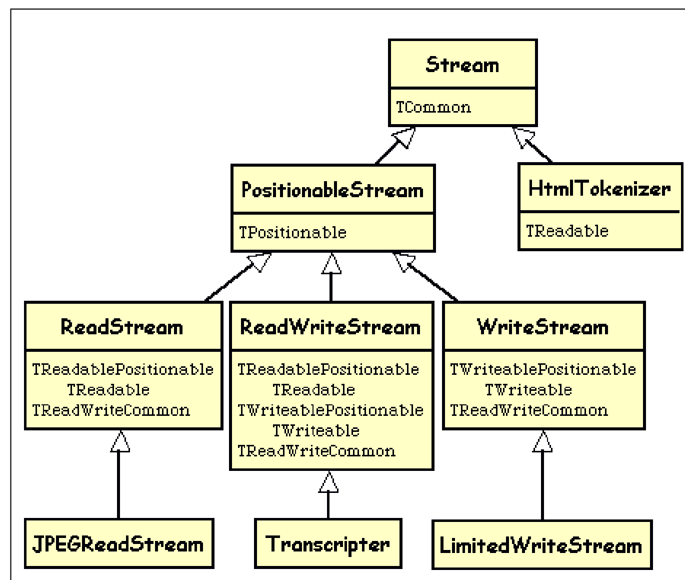


Figure 12: Classes in the refactored Stream **hierarchy.**

	Collection Classes *without* String & Text	Collection Classes *with* String & Text	Stream Classes	Totals *without* String & Text	Totals *with* String & Text
Number of concrete classes	21	23	7	28	30
Number of methods in original version	635	1 044	208	843	1 252
Number of methods in trait version	567	840	190	757	1 030
Methods saved (m)	68	204	18	86	222
Methods saved (ratio)	10.7%	19.5%	8.7%	10.2%	17.7%
Source code saved (in bytes)	9 527	14 586	1 307	10 834	15 893
Source code saved (ratio)	11.9%	10.4%	4.4%	9.9%	9.4%
Methods “too high” in original (h)	55	55	76	131	131
Methods “too high” as percentage of original	8.7%	5.3%	36.5%	15.5%	10.5%
Methods “too high” not explicitly disabled	40	40	66	106	106
Total methods saved ($m + h$)	123	259	94	217	353
Total methods saved (ratio)	19.4%	24.8%	45.2%	25.7%	28.2%

Table 1: Summary of the Refactoring

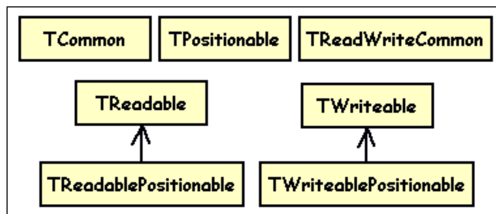


Figure 13: Traits used to build the Stream classes.

subject code. “Source code saved” is the difference in the size of the two versions, measured in bytes, and computed by decompiling the methods. This excludes comments and automatically adjusts for differences in formatting, naming of variables, and so on.

The phenomenon of methods being implemented “too high” was mentioned briefly in section 6.2. Suppose that one needs to use a method in two classes A and B that are not in a subclass relationship. In a single inheritance language, the only way to do this (other than duplicating the code) is to promote the method to the common superclass of A and B. Thus, each instance of a method being implemented “too high” represents a method that *would have had to be duplicated* if it were implemented in the logically correct place. In fact, some of these methods would have to be duplicated several times. Thus, the “Total savings” row in table 1 is simply the sum of the number of methods that were found to be implemented too high, and the number that were duplicated.

Although the evils of duplication are well-known, the problem with implementing methods too high may not be so obvious. Implementing methods too high means that inherited behavior is inappropriate and must be cancelled. For example, in the part of the collections hierarchy that we refactored, 15 messages are explicitly disabled in subclasses (typically by defining them as **self** shouldNotImplement). More problematic are the other 40 methods that are *implicitly* disabled because they directly or indirectly call explicitly disabled methods. Implementing methods too high may be better than code duplication, but it nevertheless makes the whole hierarchy very much harder to understand. For example, the method addAll: in Collection 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 one exactly which methods can be used.

The second column of the table includes the refactored String and Text classes. The comparison is between the refactored classes and a version of the collection hierarchy containing an augmented Text class that defines all of the methods found in String but missing from the standard Text class. Because regularity of the interfaces of objects is so important to object-oriented programming, we argue that this augmented Text class is the one that really ought to be provided by standard Squeak—if it were feasible to do so with existing technology. The refactored version uses traits to provide Text with all the methods of String, and thus achieves the same interfaces with far fewer methods.

The third column shows the situation with the stream classes. Note that in this hierarchy there were fewer duplicated methods, and so the reduction in the number of methods achieved with traits is slightly lower. However, the reason that there was less duplication is that an enormous number of methods were implemented “too high”, precisely so that they could be shared. Moreover, many of these methods were not explicitly disabled. This makes the Stream classes hard to understand, because it appears that these methods ought to work, but in fact they will break when they call another method that is explicitly disabled.

The final two columns present the totals for Collections and Streams, excluding and including the String and Text classes.

7.4 Assessment of the Refactored Classes

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

We undertook the refactoring in two phases. In the first phase we refactored 13 concrete collection classes, and none of the Stream classes. This phase set up the basic structure of the functional traits and the implementation traits; we developed 46 traits in all. We did not need to use exclusion or aliasing, because we were free to split any trait that was too large for our needs. As we continued our refactoring of the collection hierarchy, we found an *increasing* percentage reduction in code size as we applied the same reusable traits to remove code from more and more classes.

Returning to the refactoring three months later, we found that we were able to reuse many of our traits when we extended our refactoring to other parts of the hierarchy. Some of the traits were split into smaller pieces, but since these could be recombined without any change in semantics, the classes that had already been refactored were not affected. In some places we used exclusion to avoid disturbing an existing trait.

The traits that we have written will also allow us to construct new kinds of collection in the future, simply by composing the necessary traits and implementing a few glue methods. For example, we can build a class PluggableBag by using the trait TPluggableAdaptor in a subclass of Bag, and we can create immutable variants of collections by omitting the mutable interface traits. In addition, the availability of these traits frees the implementors of the collections framework from the need to ship pre-built collection classes for rarely used special cases. Instead, the main responsibilities of the implementor become the provision 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 special-purpose collections.

Another advantage of the new framework is that some of the traits can be used *outside* of the collection classes. As an example, the trait TEmptyness, 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 implements 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 one's way around the system.

7.5 Design Decisions

The availability of both trait composition and single inheritance gave us a lot of freedom in designing the new collection classes. Why did we choose the particular combination of trait reuse and inheritance described above? An alternative approach would have been to use trait composition exclusively and to minimize—or eliminate—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).

We decided against this approach primarily for reasons of familiarity. Using both inheritance and trait composition in the new hierarchy makes it easier for programmers who are familiar with single inheritance code, and especially for programmers who know the old collection classes, to understand and extend the new ones. Indeed, a flattened view of the new collection classes exhibits a structure quite similar to the old ones, although the abstract superclasses do not correspond one-to-one.

The combination of single-inheritance and trait composition 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 superclasses built from the functional traits, we can be sure that in these situations the implementation methods override the functional methods.

8. DISCUSSION

In this section we first discuss some of the things we learned about traits during our refactorings. Then we examine some places where the theoretical benefits of traits were of practical importance, and argue that a similarly fine-grained decomposition would be much harder to accomplish with mixins or multiple inheritance.

8.1 Lessons Learned

During this refactoring 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; when we started refactoring we did not have a very clear plan of where we would end up. 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 [11]. 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. Naming is important and difficult; 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 a layer of subtraits could be introduced or eliminated without changing the semantics of any 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 most logically 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 first phase of the refactoring. 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, particularly flattening, and
- a set of tools that exploited those properties to provide multiple views of the program.

8.2 A Comparison to Mixins

In arguing that traits are a valuable contribution to the language designer's arsenal, we must address the question of whether we could not have obtained equally impressive results using mixins or multiple inheritance. We are convinced that the answer is "no", and in this section will attempt to explain why.

We have previously presented theoretical arguments for the superiority of traits over mixins and multiple inheritance[28]. Here we will focus on experience rather than theory. We will compare traits with mixins, since multiple inheritance can be considered as a stylized application of mixins.

Our refactored collection classes are built from up to 22 traits; on average each class contains more than 5 traits. This is feasible because the sum operation lets us build a subclass from a group of traits in parallel. In contrast, mixins must be applied one at a time; this would result in huge and hard to understand inheritance chains with up to 22 levels. Even worse, there are places where we use exclusion to avoid a method conflict in a way that could not be simulated using mixins. For example, if traits TA and TB both define methods a and b, then the expression (TA - {a}) + (TB - {b}) denotes a trait containing the a method from TB and the b method from TA. If TA and TB were mixins rather than traits, there would be no way to mix both of them in and obtain this effect.

Using mixins, the solution would be to either modify TA, or to introduce a new intermediate mixin corresponding to TA - {a}. Neither choice is desirable. Modifying the components is bad because it may break other places where these components are used. Introducing intermediate components makes the inheritance chains even longer and harder to understand. This is particularly problematic because the use of **super** in mixins means that they do not support the flattening property, and thus there is no way in which these long inheritance chains can be "inlined".

With traits, these problem did not arise: sum and exclusion allowed us to obtain the right composite behavior quite easily. One example of this is the trait TSortedImpl where we had two conflicts: at:ifAbsent: and collect:, but no single subtrait that takes precedence for both of them. This is exactly the situation exclusion is designed for, and we obtained the desired behavior by excluding at:ifAbsent: from the subtrait TExtensibleSequencedImpl and collect: from TOrderedSortedCommonImpl.

Aliasing provides a way to access overridden behavior without compromising the flattening property, and thus without reducing the understandability of composite traits. We used aliasing in this way in the traits THeapImpl, TDictionaryImpl, and in the class WriteStream.

In the process of our refactoring work, we also encountered many situations where adding a new method to a component caused a conflict with another component in distant code. Thanks to the requirement of explicit conflict resolution all of these places were immediately detected, and we were able to re-establish the correct semantics by making an appropriate adjustment to the appropriate trait composition clause. It was never necessary to modify other components, so we never found ourselves in a situation where resolving one conflict created two more.

With mixins, this would not have been the case. First, we would not have detected conflicting methods so easily because the order of the mixins automatically "resolves" each conflict, although not necessarily in the way that the programmer intends! Second, even if we had noticed that a conflict had been resolved in an incorrect way, it would have been much harder to actually re-establish the correct behavior.

A comparison of our refactored collection classes to the mixin-based collection framework of Strongtalk (Animorphic Smalltalk) [7] provides more data on the effectiveness of mixins and traits. Both frameworks are based on Smalltalk-80 and are therefore quite comparable. Strongtalk has more collection classes, but uses only 10 different mixins, compared to 67 traits in our hierarchy. In particular, Strongtalk doesn't factor out characteristics such as extensible, implicitly sequenced, and explicitly sequenced; it also does not make aspects like enumeration reusable outside of the collection framework.

Of course, the fact that the designers of Strongtalk decided not to pursue a fine-grained decomposition into mixins does not mean that doing so would be impossible. However, it is an indication that the Strongtalk designers decided that the disadvantages of a finer structure outweighed the advantages. In contrast, we have found that with traits the fine-grained decomposition has only advantages.

9. RELATED WORK

The work presented here was inspired in part by Cook's study of conformance and inheritance in the Smalltalk-80 collection classes[11]. Cook first extracts an interface hierarchy based on conformance[4, 8] 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 show 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 [24]. Moore applied 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 [9, 10] 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 modeling. Dicky *et al.* propose a new algorithm to insert classes into a hierarchy that takes into account overridden and overloaded methods [12]. 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. Mirza [23] also explores the use of fine-grained composition in a collections framework, but with different goals (*e.g.*, binary deployment), and using different techniques, *e.g.*, simulated delegation.

Refactorings—behavior preserving program transformations—have become an important topic in the object-oriented reengineering community [27, 31, 32]. Research on refactoring originates with the seminal work of Opdyke [25], which defines refactorings for C++ [21, 26]. In this context, Tokuda and Batory [32] 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 [14]. However, they do not analyze the two versions of their code to compare the degree of reuse.

Some other programing languages do have constructs similar to traits, although they differ in some important details. (A study of this issue is available in a companion paper [28].) The language Self [33] even uses the name "traits", although Self traits are basically objects that play the role of method dictionaries shared by prototypes. As mentioned in section 8.2, Strongtalk is a typed version of Smalltalk that uses 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.

The Larch family of specification languages[19] is also based on a construct called a trait; the relationship turns out to be more than name deep. Larch traits are fragments of specifications that can be freely reused at fine granularity. For example, it is possible to define a Larch trait such as IsEmpty that adds a single operation to an existing container data-type. There are, of course, significant differences, since our traits are not intended to be used to prove properties of programs, and adding a trait to a class does not formally constrain the behavior of existing methods.

10. FUTURE WORK

We feel that our refactoring of the Collections classes has reached a natural conclusion; although it is not complete, we have dealt with a wide enough variety of classes that we do not believe that we will learn very much more from filling in the remaining corners. However, we would like to conduct a more sophisticated analysis of code duplication, perhaps using a tool such as Duploc [13] or Guru (see section 9). Both the Squeak Foundation and Cincom have expressed interest in incorporating traits into the main development stream of their Smalltalk systems. If this happens, it would motivate the inclusion of all of the remaining classes into the refactored framework. 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 pleased with how well the tools that we have built actually support the process of programming with traits. There are a few missing features, but the tools are quite usable by others and are available for download; we hope to hear reports of their application to other domains. We will use this widening base of experience to influence the still-evolving design of traits; we also plan to study how traits might be incorporated into typed languages and into languages in which instance variables can be part of an object's interface.

Taking a wider view, we do not see traits as just another programming construct, but as one of the enabling technologies for a grander vision called multi-view programming. This vision is gradually being realized in the Perspectives project [5]. Rather than thinking of a refactoring of a program as creating a new (equivalent) program, the idea behind Perspectives is to treat both the original and refactored programs as different *views* on the same abstract entity. Like views on a database, this would raise the level of abstraction at which the user works. A new language technology like traits, which greatly extends the range of possible refactorings, also defines a new set of views, and thus permits programming at a more abstract and powerful level. 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 retention of explicit conflicts in the sum operation—were the right ones. But programing languages are tools, and theoretical elegance is no substitute for usability. Only extensive use on a realistic codebase could validate these beliefs.

It did. Although we had designed them, we were surprised how well the tools and the trait concepts worked in practice. The theoretical characteristics do really seem to give the tools desirable practical properties.

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 them. One of the pleasant properties of traits is that they do not 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. Trait 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 the programmer modifies a method in a conventional class view, and the method is actually defined in a shared trait, then the effect will be to define a customized (unshared) version of the 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 linearizing multiple inheritance chains must be understood by every programmer who looks at or modifies a multiple inheritance hierarchy.

While our experiences with traits validated our expectations, there were also some surprises. For example, it turned out that the fine-grained trait structure has no disadvantages: 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 multiple views tailored to the task at hand.

Another surprise was that the refactoring process turned out to be quite enjoyable and very straightforward. Trait-based refactoring seems to be compatible with an extreme programming style of development: 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 a 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 summarize: we have successfully refactored a significant subset of the Smalltalk collections classes. In the process we:

- removed code duplication;
- increased uniformity;
- 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 third 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 mutate the object, 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.

Acknowledgments. 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.

12. REFERENCES

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

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

[3] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 78–86, November 1986.

[4] 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.

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

[6] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[7] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 215–230, October 1993.

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

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

[10] 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.

[11] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 1–15, October 1992.

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

[13] 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.

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

[15] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

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

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

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

[19] John V. Guttag, James J. Horning, and Jeannette M. Wing. The larch family of specification languages. *IEEE Transactions on Software Engineering*, 2(5):24–36, September 1985.

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

[21] 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.

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

[23] Yahya H. Mirza. A Compositional Component Collections Framework. Seventh International Workshop on Component-Oriented Programming (WCOP 2002) at ECOOP 2002, Malaga, Spain, June 2002.

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

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

[26] 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.

[27] 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.

[28] Nathanael Schärli and Andrew Black. A browser for incremental programming. Technical Report CSE-03-008, OGI School of Science & Engineering, Beaverton, Oregon, USA, April 2003.

[29] Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black. Traits: The formal model. Technical Report IAM-02-006, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-013, OGI School of Science & Engineering, Beaverton, Oregon, USA.

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

[31] 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.

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

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