Traits: Experience with a Language Feature

Emerson R. Murphy-Hill
The Evergreen State College
2700 Evergreen Parkway NW
Olympia, WA 98505
mureme12@evergreen.edu

Andrew P. Black
OGI School of Science & Engineering,
Oregon Health and Science University
20000 NW Walker Rd
Beaverton, OR 97006
black@cse.ogi.edu

ABSTRACT

This paper reports our experiences using traits, collections of pure methods designed to promote reuse and understandability in object-oriented programs. Traits had previously been used to refactor the Smalltalk collection hierarchy, but only by the creators of traits themselves. This experience report represents the first independent test of these language features. Murphy-Hill implemented a substantial multi-class data structure called ropes that makes significant use of traits. We found that traits improved understandability and reduced the number of methods that needed to be written by 46%.

Categories and Subject Descriptors

D.2.3 [**Programming Languages**]: Coding Tools and Techniques - *object-oriented programming*

D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, inheritance*

General Terms

Design, Languages

Keywords

Traits, Ropes, Reuse, Smalltalk, Inheritance

1. INTRODUCTION

Reusability is a very important property of object-oriented programs. With no reuse, all the methods that a class requires would need to be defined in the class itself. When two or more classes define the same method, code is duplicated. In addition to being wasteful by taking up memory, duplicated code lowers programmer productivity in two ways. Initially, the programmer must take the time to make a copy of a certain method. Later, if the desired semantics of that method changes, or if a bug is found, the programmer must track down and fix every copy. By reusing a method, behavior can be defined and maintained in one place.

The research described in this experience paper was conducted at the OGI School of Science & Engineering at OHSU by an undergraduate student from The Evergreen State College. It was funded by NSF award CCR 0098323.

In object-oriented programming, inheritance is the normal way of reusing methods—classes inherit methods from other classes. Single inheritance is the most basic and most widespread type of inheritance. It allows methods to be shared among classes in an elegant and efficient way, but does not always allow for maximum reuse.

Consider a small example. In Squeak [7], a dialect of Smalltalk, the class Collection is the superclass of all the classes that implement collection data structures, including Array, Heap, and Set. The property of being empty is common to many objects—it simply requires that the object have a *size* method, and that the method returns zero. Since all collections have a size, it makes sense that the method *isEmpty* should be defined for all collection classes. Because Squeak uses single inheritance, by defining *isEmpty* in the class Collection, all subclasses of Collection inherit the *isEmpty* method.

However, a problem arises with classes outside the collection hierarchy. Take the class CRDictionary, a part of the "Genie" handwriting recognition system in Squeak. This class is not a subclass of Collection, but it does have a *size* and an *isEmpty* method. The method *isEmpty* was written in exactly the same way twice; once in Collection and once in CRDictionary.

This may seem like a small problem—one repeated method is not a lot of repeated code. But this problem is compounded by several factors. First, the method *isEmpty* occurs in 24 classes, often defined in a similar manner. Second, a number of similar methods that rely exclusively on *isEmpty* are common, including *notEmpty* and *ifEmpty:*. Third, it is quite common to find duplicated methods in unrelated classes in the class hierarchy. For example, the classes Rectangle and RectangleMorph, have much common code that they cannot share because they are unrelated: their only common superclass is Object [3]. How can we reuse code more effectively?

One workaround is simply to tolerate the missing methods, rather than dupliciating them. For example, the user of a CRDictionary could be required to convert it into a Dictionary, CRDictionary could be required to convert it into a Dictionary, which is a subclass of Collection, and then send the *isEmpty* message to the Dictionary, but this would be grossly inefficient and exceedingly verbose. Another workaround would be to push the method *isEmpty* up the hierarchy into the Object class, but *isEmpty* is inappropriate in subclasses of Object that do not have a *size* method. Multiple inheritance is another solution, but it presents its own set of problems, such as the possibility of conflicting variable and method names. No alternative to single inheritance has gained widespread acceptance [8].

2. TRAITS

Traits are a new solution to the problem of code duplication across unrelated classes [5]. A trait is a reusable building block from which programmers construct classes. Traits are collections of pure methods, much like classes, except that they have no superclass and no state. Traits have a set of methods that define behavior, called the "provides set", and another set of methods that have yet to be defined, called the "requires set."

Suppose that we want to create a trait to solve the *isEmpty* problem. We define the method *isEmpty* in terms of *size*, but we cannot yet define *size*, because it is often dependent on implementation. Furthermore, the methods *notEmpty, ifEmpty:*, and *ifNotEmpty:* can be defined in terms of *isEmpty*. So *isEmpty, notEmpty, ifEmpty:*, and *ifNotEmpty:* are in the provides set and *size* is in the requires set. We now have a reusable building block that we will call TEmptyness (see Figure 1).

TEmptyness can be incorporated into the Collection class, into the CRDictionary class, and into any other class that defines *size*. To incorporate a trait into a class, the code in this class has to be refactored so that old, repeated methods are removed. Then, a trait should be made which defines these methods and the trait itself is used in the class. Each class that duplicates behavior defined by the trait should be refactored in

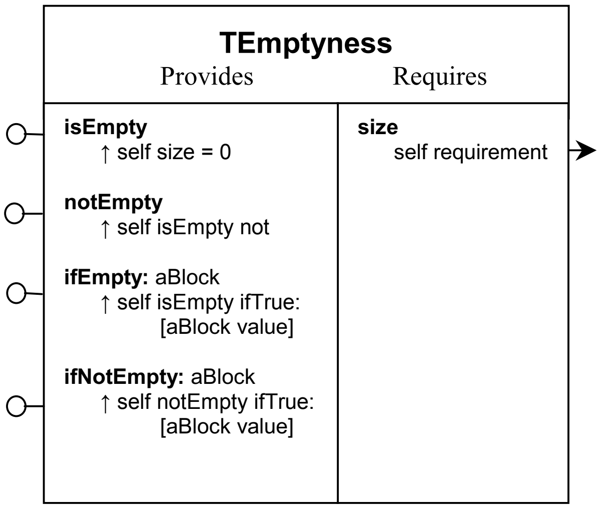


Figure 1. The trait TEmptyness with the provided methods in the left column and the required methods in the right column.

this manner. Using traits, we can define behavior in one place and have that behavior utilized in multiple places. In addition, the protocols of the classes that use traits tend to become more uniform. For example, while 20 Squeak classes define *isEmpty*, only 2 also define *notEmpty*. However, if the methods had been reused from the TEmptyness trait, both methods would be defined in all the classes. Readers wishing to learn more about traits and the comparison between traits and other language features are referred to previous publications [3,5].

Small examples are acceptable illustrations of the possibilities of traits, but little is known about how useful traits are in a realistic programming situation. Traits were tested by refactoring the entire collections hierarchy [3], but the refactoring was done by two of the creators of traits themselves—Andrew Black and Nathanael Schärli. Emerson Murphy-Hill, a programmer familiar with object-oriented programming but unfamiliar with traits, was assigned to evaluate traits in an independent manner, under the gentle direction of Andrew Black. Murphy-Hill spent a summer at OGI School of Science and Engineering with a Research Experience for Undergraduates grant from the National Science Foundation. At the beginning of the project, Murphy-Hill was unfamiliar with both Smalltalk and traits. While Black assisted him in learning Smalltalk, Black did *not* provide guidance on how to use traits. Murphy-Hill learned about traits from some of the available papers [3,5,6], and utilized this knowledge to write a significant, multi-class data structure called *ropes*. The entire design and implementation process took about one month, but certainly could have progressed faster if the programmer had experience with Smalltalk prior to this experience.

3. ROPES

Ropes are an alternative to strings originally developed by Boehm, Atkinson, and Plass at Xerox PARC [4]. Strings are almost always implemented as fixed length arrays of characters, which are not ideal for some of the most common string operations. In particular, concatenation and substring selection are common operations that strings perform slowly in traditional implementations because they require that large portions of the array be copied. Ropes make both concatenation and substring selection faster by introducing alternative data structures. Although these data structures require ropes to be immutable, immutability can be considered an advantage in that it greatly simplifies sharing of ropes between cooperating processes.

4. THE EXPERIMENT

To see the effect of traits on the development process, we implemented ropes twice: once with traits, and once without. We based both versions on a Java implementation of ropes by Andrew Black, which utilized Aspects [2]. Black chose ropes as an application of traits for Murphy-Hill and let him develop ropes as he saw fit.

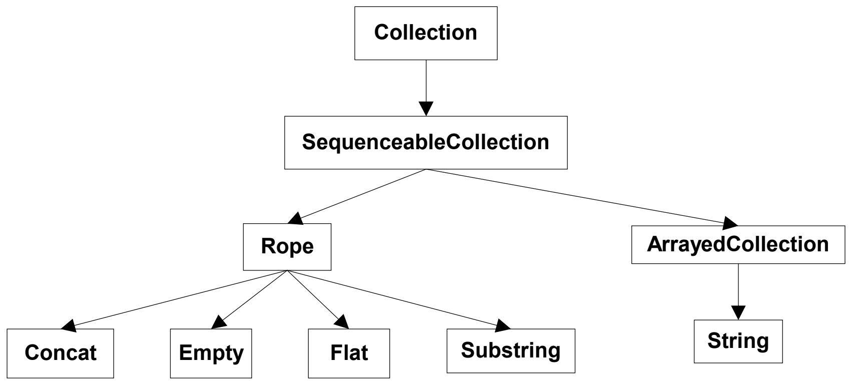


Figure 2. The partial Collection hierarchy, showing the String class and the Rope class.

4.1. First Implementation: Ropes without Traits

The first issue was where to put a Rope class so that it could reuse as much behavior from String as possible. To gain the greatest amount of reuse, the Rope class should be a subclass of String. However, this is both inelegant conceptually and difficult in practice. It is inelegant in that a single-inheritance hierarchy, the parent-child relationship is an "is-a" relationship. A rope "isn't-a" String nor an ArrayedCollection physically, and to make it a subclass of either would be to imply the contrary. It is difficult in practice because, in Squeak, any class occupying a variable amount of storage, such as String, cannot have named instance variables; nor can any of its subclasses. The superclass SequenceableCollection is an appropriate place to share some collection behavior with String, without acquiring inappropriate implementation-specific behavior because SequenceableCollection is an abstract class that encompasses collections whose elements are ordered (Figure 2). In order to ensure that it had an interface similar to that of String, Rope was initially just a copy of String without any underlying data structure and without any mutating methods. In this way, the Rope made no assumptions about instance variables used by its subclasses.

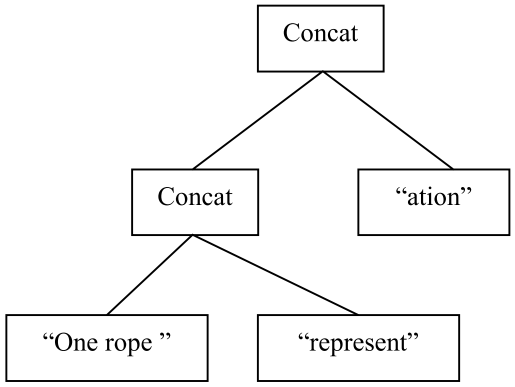


Figure 3. A series of concatenations representing a rope.

To implement ropes, Murphy-Hill wrote four subclasses of Rope: Empty, Flat, Substring, and Concat, each with a different underlying data representation. Empty implements the singleton pattern and therefore behaves the same across all instantiations. Flat has an array structure, like String. Substring takes any other Rope, and stores the index of the start of the requested substring and its length. Concat stores references to the Ropes being concatenated and, for efficiency, caches their lengths. A series of concatenations leads to a tree of Concats (see Figure 3) with each leaf being a Flat or a Substring and each internal node meaning "concatenated with". All methods that are data-structure dependent were not implemented in Rope, but were flagged as abstract so that they would be required in each of the four subclasses.

Empty needs no instance variables. This class allows ropes of size zero to respond quickly to messages, since they often have little or no behavior. For example, the method do: iterates over a collection and the method capitalized returns the receiver with the first letter capitalized. For an Empty, do: does nothing and capitalized returns the receiver, unmodified.

```
capitalized
    ↑ self
```

Flat was intended to be a copy of String without the mutating methods. To accomplish this, Murphy-Hill made Flat use the same array data structure as String, and each implementation-specific method was simply copied from either String or its superclass ArrayedCollection. With the exception of the mutating methods, this gave Flat the same interface as String. The Java interface would be a good choice, but no such enforcement mechanism exists in Smalltalk. The similarity was simply a convention dictated by the programmer and hopefully observed by future maintainers.

Substring was fairly straightforward to implement. A substring selection takes three arguments which are stored in a new Substring object: a base Rope from which the substring is requested, the offset of the head of the substring in the base, and the length of the substring. For most methods on a Substring, the method is passed to the base, accounting for the position and the length of the substring. For example, the *at:* method, which indexes into a string, needs only to add the offset of the head to the requested index, and then send the message to the base rope:

```
at: index
    ↑ base at: (baseOffset + index – 1).
```

Concatenation of two Ropes is easy—a new Concat object is built with two variables pointing to the arguments. Methods performed on Concats often are defined by passing the method to the left rope and then to the right rope. For example, the method *do:* in the Concat class is defined as follows.

```
do: aBlock
    left do: aBlock.
    right do: aBlock.
```

The method *at:* is a constant-time operation on an array-based data structure, but takes logarithmic time on a Concat tree. For this reason, methods such as *do:* which relied on a sequence of *at:* operations in class String, had to be reimplemented in Concat.

In addition to implementing methods in the protocol of String, methods were needed across the Rope hierarchy to balance Concat trees. Although concatenation and substring selection are highly efficient with ropes, other methods can become inefficient if a Rope contains an unbalanced concatenation tree. For example, the *at:* method becomes very inefficient when it has to traverse a very deep tree. For this reason, the ability to rebalance themselves is a very important property of ropes. Some methods necessary for balancing, such as *isBalanced,* needed to be implemented in each subclass of Rope because Substring and Flat ropes are always balanced, whereas Concat objects may or may not be balanced, depending on their depth and length. Other methods, such as those that do the majority of the actual balancing, needed to be implemented only in the Rope class.

4.2. Second Implementation: Ropes with Traits

Once the first implementation was complete, Murphy-Hill considered how to implement ropes with traits. It was advantageous to place our Rope class in the refactored collections hierarchy (see Figure 4). This refactoring had been designed previously using traits and reduced the number of methods in the hierarchy by about 10% [3]. By placing the Rope class and its subclasses in this hierarchy, we immediately gain some of the savings introduced by traits and the refactored collections hierarchy. Additionally, two traits useful for ropes already existed: TStringI and TStringImpl. TstringI, the T- prefix meaning that it is a trait rather than a class and the -I postfix indicating that it is immutable, is a large trait composed of smaller traits that provide implementation details for strings. In conjunction with TStringI, TStringImpl provides almost all of the remaining methods of String, with the exception of the methods requiring mutability.

It would have been acceptable to make Rope a subclass of SequencedExplicitly, just like String in the refactored collections hierarchy. However, there already existed an abstract class called SequencedImmutable with an interface with exactly the same methods. SequencedImmutable was a more natural choice because, unlike SequencedExplicitly, it does not require the mutating method *at:put:*, a method which writes a value at a specified index in a sequenced collection.

Unlike the trait-free implementation described in Section 4.1, it was not necessary to copy methods in String into Rope in order to make the interface of Rope similar to that of String. Instead, the trait TStringI provided most of the methods in the String class. The classes Substring and Concat were written in much

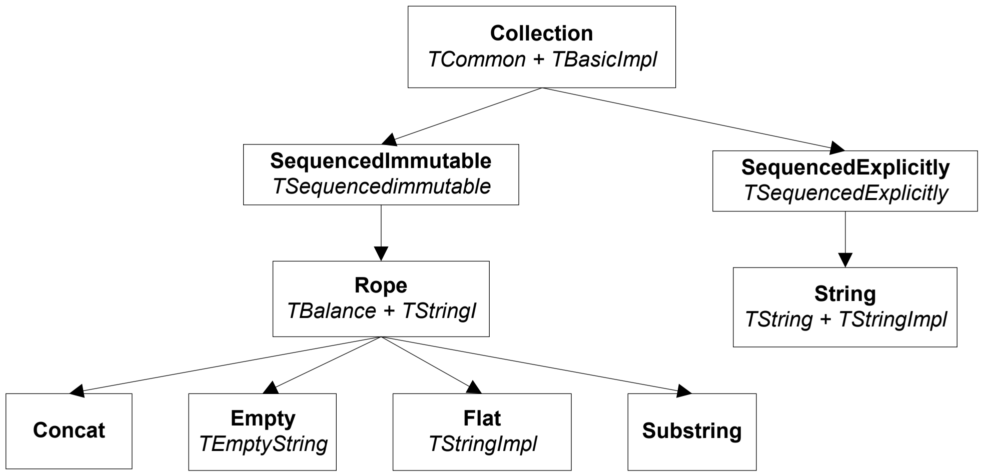


Figure 4. The refactored Collections hierarchy with traits. Each trait is shown in italics.

in the same way as the trait-free implementation; methods whose efficiency was implementation-dependent were overridden or implemented only in the appropriate subclass.

Of all the classes that Murphy-Hill wrote, the Flat class benefits most from reusability. As with the trait-free version, the programmer made its interface identical to String, minus the mutating methods. Because Flat uses TStringImpl and its superclass uses TStringI, Flat has exactly the interface desired—the interface of String without the mutating methods.

Instead of putting the balancing methods directly in the Rope classes, the programmer instead created a new trait called TBalance that included the methods necessary to balance Concat trees. Not all balancing methods could be implemented in TBalance, however. For example, as with the trait-free version, *isBalanced* is best defined separately in Flat, Empty, and Rope. Thus, *isBalanced* becomes a required method of the trait TBalance.

Another trait that the programmer created was TEmptyString. This trait provided almost all the methods for the Empty class. Although there was no code savings and understandability of the code was not changed, creating and using this trait made sense for two reasons. First, since both traits and Empty have no state, all the behavior of an Empty can be put into a trait. For example, it is easy to define *length* for an Empty rope inside of a trait:

```
length
    ↑ 0.
```

Second, if in the future anyone needs to write a class like String with static behavior, then this trait would be very easily reusable.

5. FINDINGS

From our experience implementing ropes with and without traits, we were able to make a number of observations about traits in general.

5.1. Traits Are Easy to Learn

Although Murphy-Hill had to learn about Smalltalk, ropes, and traits, he found that traits were the least difficult concept to grasp. The programmer was able to easily grasp the concept of traits, not because the previous writing on traits was exceptionally clear, but because the use of traits was intuitive to an object-oriented programmer. Almost without exception, issues associated with traits, such as composition, nesting, conflict resolution, overriding, and precedence, were easy to understand during development.

5.2. Traits Allow Methods to be Reused

Figure 5 shows the composition of classes in the version of ropes with traits. Each class is made up of three components: "own" methods of the class, methods reused from traits, and methods implemented with traits. Own methods are those that have been implemented as usual, in the class itself. Methods reused from traits are those that exist in traits that had been previously created and used in the refactored Collection hierarchy. An example is TStringImpl, which is also used in the class String. Methods implemented with traits are those that exist in traits that were written expressly for this application, such as TBalance.

Figure 6 shows the number of methods in each class written in the version of ropes without traits. All methods were the "own" methods of the class, that is, all were written in the class itself.

In comparing Figures 5 and 6, we can see that each has roughly

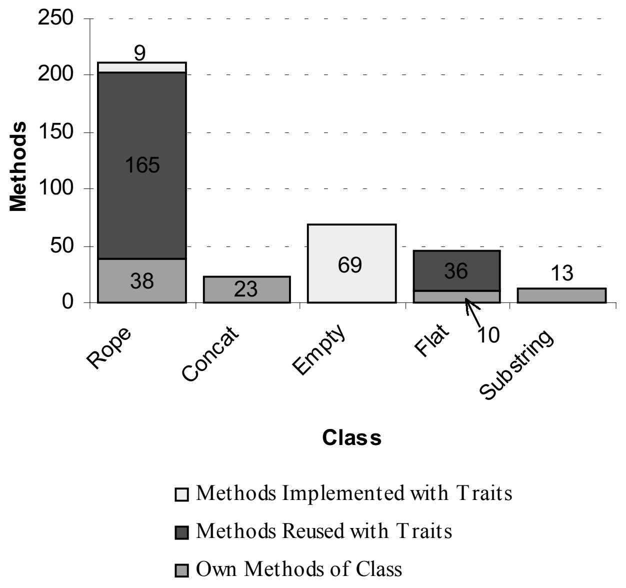


Figure 5. Methods in Traits Version

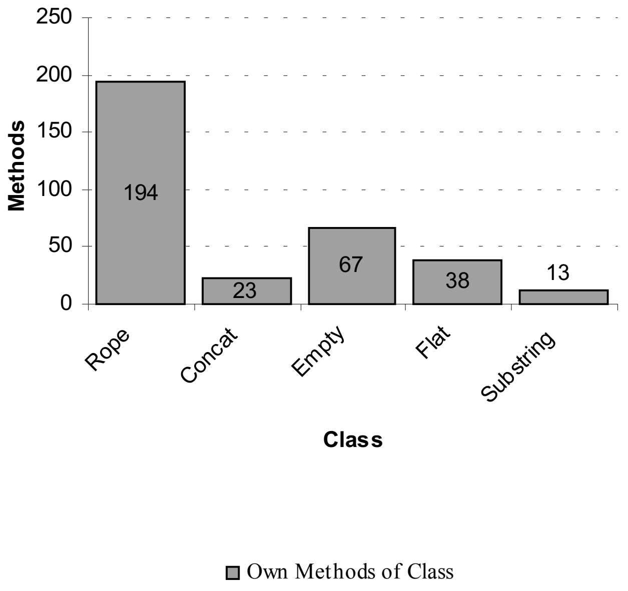
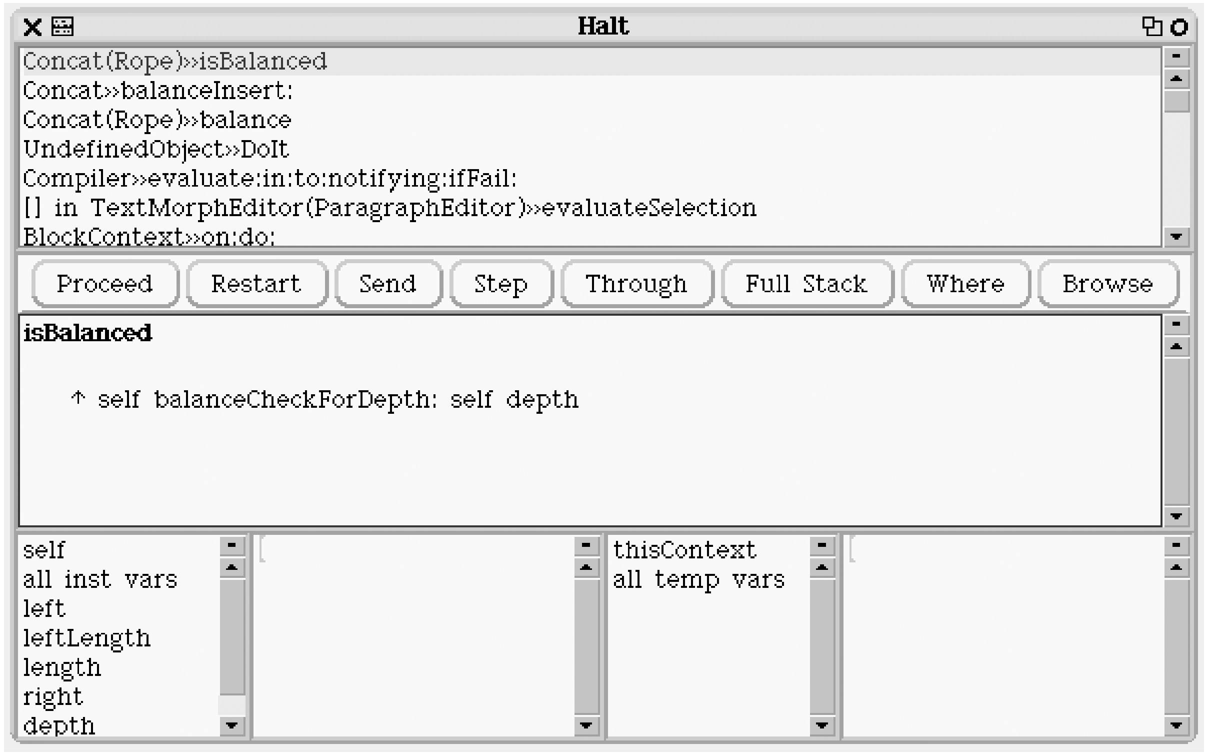


Figure 6. Methods in Trait-less Version



```
Concat(Rope)»isBalanced
Concat»balanceInsert:
Concat(Rope)»balance
UndefinedObject»DoIt
Compiler»evaluate:in:to:notifying:ifFail:
[] in TextMorphEditor(ParagraphEditor)»evaluateSelection
BlockContext»on:do:
```

```
isBalanced

    ↑ self balanceCheckForDepth: self depth
```

Figure 7. The Squeak debugger. Although the method isBalanced is defined in the trait TBalance, the debugger shows it as part of the class Rope (as shown in the top pane, on the top line).

the same number of methods per class, and roughly the same number of methods overall. The small differences in the number of methods implemented across the versions can be accounted for by the previous refactoring of the collection hierarchy and by the organization of the traits. Some methods were refactored into a superclass, while others were left for individual implementations. Howsoever, each version was designed to be identical in use. In the traits version, fewer methods had to be written — 46 per cent of the methods were reused from previously defined traits. In all, 77 per cent of the code exists in traits, and could therefore be reused without subclassing the existing data structure. In the version of ropes without traits, none of the data structure is reusable, except when creating a subclass.

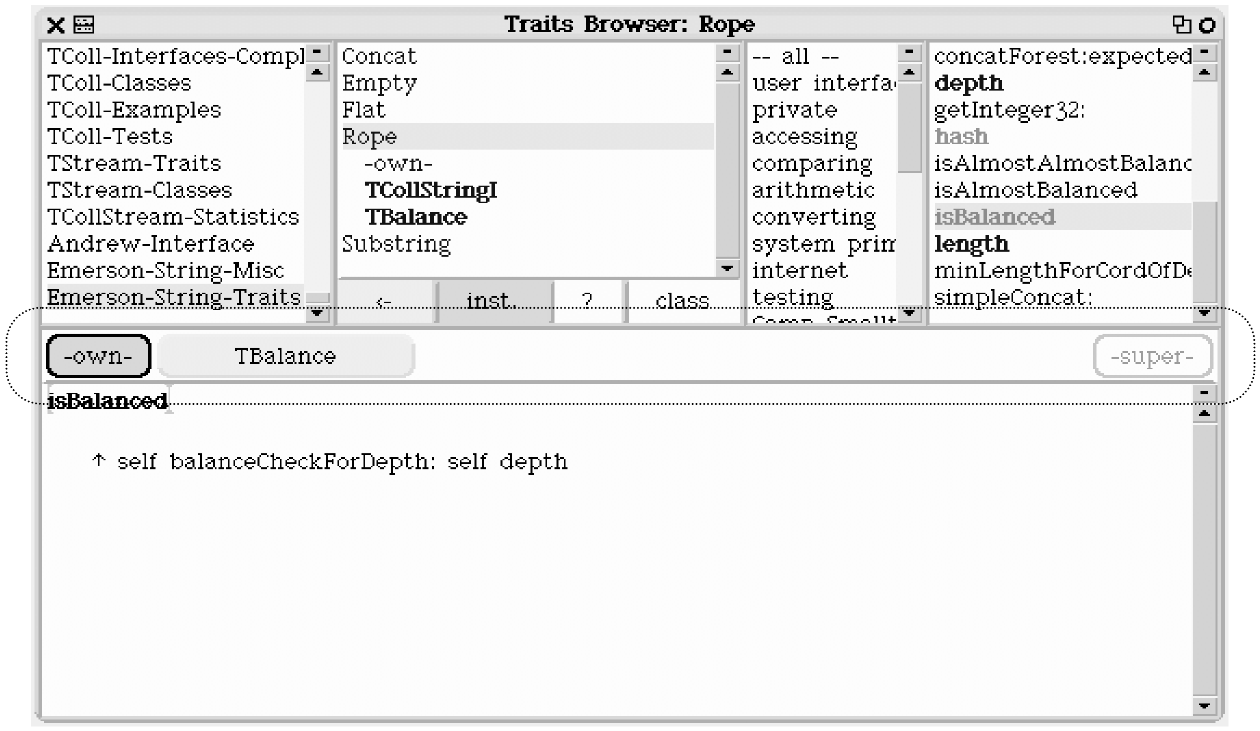
The Flat class had the greatest potential for reuse, since it needed to be behaviorally very similar to the String class. If you consider the bulk of the behavior in the Flat interface as defined in the Flat and Rope class, 78% of the behavior was obtained from preexisting traits. We feel this is a very significant saving in both code size and programmer time.

To say that 78 per cent of the methods were obtained "for free" would be to understate the impact of traits. Although a variety of traits already exist for Collections in Squeak, traits are useful because they give one the ability to create reusable components, rather than because a huge library of traits already exists. For this project, although it was helpful to have traits on hand that captured the interface of String, a set of traits could have been constructed from scratch. This is not a difficult task if a class already exists which exhibits the desired behavior. The Traits Browser [6], a tool for working with classes, methods, and traits in the Squeak environment, includes a feature that allows one to make a trait from a class, even if the class contains no traits to start with. Essentially, the tool automatically generates a copy of the class in trait form by replacing all references to instance and class variables by message sends to accessor methods. These accessor methods become requirements of the new trait, and need to be defined in every concrete class that uses the trait. This process is an easy way create traits from preexisting classes. Even if a set of traits had not been conveniently available for reuse, traits would have still allowed us to create our own reusable behavior.

5.3. Traits Have Unanticipated Side Effects

We found that using preexisting traits or making traits from preexisting classes had a negative side effect. In the version of ropes without traits, although it was tedious to copy and paste methods from the String class, at least this copying forced us to look briefly at each method. In each case, we could determine whether the method was written efficiently for our implementation. If it was not, we could reimplement it in a more efficient manner.

We found that using methods provided by traits had a "black-box" effect. The ease of using traits encourages the programmer to assume that they work as-is, as long as all the required methods are defined. Indeed, methods implemented in traits often do work in a class, but not necessarily in the most efficient manner.



```
isBalanced

    ↑ self balanceCheckForDepth: self depth
```

Figure 8. The middle bar of the Traits Browser (outlined in the center). The darkened "-own-" button indicates the method isBalanced belongs to the class Rope. This method overrides a method of the same name in the trait TBalance.

The desire to have a trait with the smallest set of required methods can conflict with the desire to have methods that are efficient across many implementations.

5.4. Traits Demand Reliable Tools

This experiment highlighted the need for reliable tools when working with traits. Squeak, in addition to being a flavor of Smalltalk, is also a development environment. All code is packaged into a monolithic image, so Squeak and the development tools packed therein are the only access to the code itself. Although traits can be created and used without any special tools, we found that the standard tool for working with traits, the Trait Browser [6], is an important ingredient in developing traits and classes that use traits. Emerson Murphy-Hill came to rely on many features of the browser, such as the automatic and instantaneous calculation of required methods and the ability to make traits from classes. Even in the trait-free version of ropes, the programmer used the Traits Browser as the central development tool. For example, after structuring the classes and defining the bulk of the Rope interface, the programmer relied on the Traits Browser's display of the required methods to tell him which methods still needed to be defined in each subclass.

Furthermore, we found that modifying the methods of a trait can be counterintuitive at times. After a trait is used in a class, it can be viewed in the Traits Browser in two ways: as part of the trait or as part of the class that uses the trait. However, the Squeak debugger always forced us to view such a method as though it were implemented in the class (see Figure 7). This is true of other tools in Squeak that are not yet aware of traits. The effect is that after editing a method in the debugger, a new version of the method is committed to the *class*, rather than modifying the method in the trait. Inadvertently and frequently, the programmer overrode the trait method. In essence, the programmer was making changes to the class when he thought he was making changes to the trait. This led to bugs that were difficult to track down, since introduced problems were subtle in effect. This is similar to what Allen calls "The Rogue Tile" [1]: in this case, a bug is fixed in the class but still exists in the trait.

A remedy for this inadvertent overriding exists in the Traits Browser. The browser has a bar that indicates whether the current method is implemented in the class, a trait that the class uses, or in one of the superclasses (see Figure 8). However, the programmer hadn't noticed the function of the bar for the first half of the project! Moreover, a similar indicator had yet to be added to other tools such as the debugger.

5.5. Traits Enhance Understanding

In addition to providing reusability, we found traits were also useful for understandability. For example, since concatenation trees need to be balanced to maintain their efficiency, the programmer factored-out a balancing trait, TBalance, that captures the bulk of the balancing operation. When the programmer wanted to modify how a concatenation tree is balanced, he simply modified the trait.

However, the ability to divide the semantic portions of ropes was limited. Although most of the methods relating to balancing were contained in the trait TBalance, some portions were impossible to factor out using traits. Ideally, we would like to simply use TBalance, then sit back and expect our tree to balance. Despite having a *balance* method that does the balancing for us, that method still has to be called explicitly whenever a concatenation tree becomes unbalanced. The version of ropes written in Java [2] utilizes Aspects to factor out balancing, successfully making a complete semantic separation. In this Java version of ropes, a snippet of code was written that calls the balancing method. This snippet is later woven into the head of a set of specified methods. The programmer was unable to accomplish this complete separation because traits are unable to achieve a finer level of granularity for code reuse than methods. However, the programmer still achieved a large degree of reuse in balancing behavior by adding the "glue" code by hand.

While the Aspects version of traits achieved a complete separation of core rope behavior from balancing behavior, we feel that traits had a significant advantage over Aspects in terms of code reuse. Consider the example of a programmer in the future wanting to reuse the balancing code, either with the Aspects version or the traits version. The Aspects version could not be reused directly, because the Balance Aspect contains explicit references to the classes it inserts code into. The traits version could be reused directly because the Balance trait contains no references to any of the classes that use it. Since it would be necessary for the programmer to use a modified version of the Balance Aspect but could use the Balance trait as-is, traits have a greater degree of reuse than Aspects in our experience.

6. CONCLUSION

While both versions of ropes were functionally equivalent, we found that the development of ropes was more enjoyable with traits than without. We found the ability to use traits as reusable units of code across the single-inheritance hierarchy a liberating experience. It took the programmer less time to develop ropes by reusing code, via traits, than it took to copy code, via cut-and-paste. By reusing 46% of the methods, we cut down on code duplication significantly. Furthermore, we were able to understand our code more thoroughly by separating semantic aspects of our code using traits. Although we ran across minor technical problems, we feel that traits are a beneficial addition to software development.

7. REFERENCES

[1] E. Allen, *Bug Patterns in Java*. Berkeley: Apress, 2002.

[2] A. P. Black, "Cords," 1.0 ed. Beaverton, Oregon, USA: OGI School of Science & Engineering, 1998, http://www.cse.ogi.edu/~black/3AspectExamples/cords.html

[3] A. P. Black, N. Schärli, and S. Ducasse, "Applying Traits to the Smalltalk Collection Hierarchy," *ACM Conference on Object Oriented Systems, Languages and Applications (OOPSLA 2003)*, Anaheim, California, USA, 2003.

[4] H.-J. Boehm, R. R. Atkinson, and M. F. Plass, "Ropes: an Alternative to Strings," *Software Practice & Experience*, vol. 25, pp. 1315-1330, 1995.

[5] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable Units of Behavior," *European Conference on Object-Oriented Programming (ECOOP)*, Springer LNCS 2743, Darmstadt, Germany, June 2003.

[6] A. P. Black and N. Schärli. "Traits: Tools and Methodology". *International Conference on Software Engineering (ICSE)* Edinburgh, Scotland, May 2004, pp 676-686.

[7] Squeak, "Squeak, Home Page" accessed June 2003: Squeak Foundation, 2000. http://www.squeak.org

[8] A. Taivalsaari, "On the notion of inheritance," ACM Computing Surveys, vol. 28, pp. 438–479, 1996.