Automatic Inheritance Hierarchy Restructuring and Method Refactoring

Ivan Moore

Department of Computer Science
University of Manchester, Oxford Road
Manchester M13 9PL, England

Abstract

Most object-oriented programs have imperfectly designed inheritance hierarchies and imperfectly factored methods, and these imperfections tend to increase with maintenance. Hence, even object-oriented programs are more expensive to maintain, harder to understand and larger than necessary. Automatic restructuring of inheritance hierarchies and refactoring of methods can improve the design of inheritance hierarchies, and the factoring of methods. This results in programs being smaller, having better code re-use and being more consistent. This paper describes Guru, a prototype tool for automatic inheritance hierarchy restructuring and method refactoring of Self programs. Results from realistic applications of the tool are presented.

1 Introduction

Factoring shared methods into classes and shared code into methods allows systems to be compact and improves consistency, making them more easily understood and less expensive to maintain. Manually designing inheritance hierarchies and methods which maximize factoring is very difficult. Even if a system is well designed initially, maintenance and evolution will tend to cause its design to deteriorate. Many programmers are reluctant to manually restructure a system, as this can be very difficult, particularly if the system is large and has been built by many different programmers. For large systems, no one programmer may understand the whole system at the level of detail of individual methods. Manual restructuring is also error prone and, while a system works, however badly structured it is, the temptation is to leave it alone.

Previous work on automatic and semi-automatic structuring and restructuring of object-oriented systems [Casais92] [Dicky96] [Godin93] [Hoeck93] [Lieberherr88][Lieberherr91][Moore95] [Opdyke92] [Pun90] has concentrated on areas other than refactoring expressions from methods; such as restructuring inheritance hierarchies considering methods as indivisible. Factoring common code out of methods into abstract superclasses is considered in [Opdyke92]. However, user interaction is required; in particular the user specifies which (two) classes to refactor methods and common code from, to put into a shared immediate (abstract) superclass. Work on refactoring expressions from functions or procedures in conventional (non object-oriented) programming languages [Lano93] is not entirely applicable to object-oriented languages, as conventional programming languages lack inheritance, which affects how methods can be shared, and there is no message sending polymorphism, which affects how methods can be refactored. Ideas related to factoring as a feature of good design are explored in [Wolff94].

This paper appeared in the proceedings of OOPSLA 1996.

Guru [Moore95] explores a radical new approach: restructuring inheritance hierarchies and refactoring methods simultaneously. In the resulting inheritance hierarchies, none of the methods and none of the expressions that can be factored out are duplicated. This paper describes how Guru has been extended, from automatically restructuring an inheritance hierarchy to automatically refactoring methods at the same time. As Guru is automatic it can be used frequently, for example after every design iteration. In comparison with [Opdyke92], rather than refactoring methods and common code from user chosen classes into an abstract superclass, the inheritance hierarchy is restructured by Guru to enable the maximum amount of sharing of methods and expressions.

Inheritance hierarchy structure and method factoring are only two aspects of the design of an object-oriented system. There are other aspects of object-oriented design which Guru does not address.

Guru is so named as it assists in Self improvement.

2 Automatic Inheritance Hierarchy Restructuring

This section briefly describes the inheritance hierarchy restructuring, without refactoring, performed by Guru. A more detailed description can be found in [Moore95].

Guru takes a collection of objects (or classes), which need not be related by inheritance and which do not have to be a complete inheritance hierarchy, and restructures them into a new inheritance hierarchy in which there are no duplicated methods, and the behavior of objects (or classes) is preserved.

The existing inheritance hierarchy is removed, and a new one inferred using a simple but efficient algorithm [Moore96][1], which infers a hierarchy that reflects the structure inherent in its objects. The algorithm is briefly described in Appendix A. Only the features of concrete objects and classes, such as methods and instance variables, are used to infer the hierarchy. The structure of the original hierarchy is irrelevant to Guru; any hierarchy defining the same features of concrete objects and classes will be restructured into the same hierarchy by Guru.

A simple example from [Moore95] is presented below to illustrate the inheritance hierarchy restructuring performed by Guru. Consider the classes shown in Figure 1. Inheritance is shown as arrows from subclasses to superclasses. The labels represent the methods defined by each class, and methods of the same name are equivalent in this example. The existing inheritance hierarchy is removed, resulting in the classes shown in Figure 2. A new inheritance hierarchy which preserves the behavior of the concrete classes (in this case the leaves of the hierarchy), and in which no methods are duplicated, is inferred, resulting in the hierarchy shown in Figure 3.

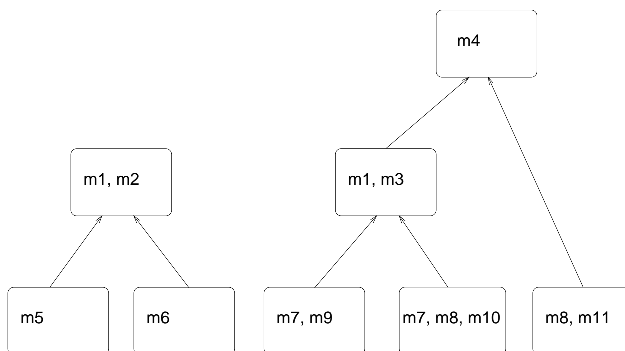

Figure 1: Example inheritance hierarchy

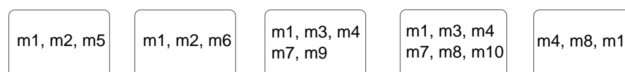

Figure 2: Example classes after removing hierarchy

[1] Discovered independently of a similar algorithm described in [Mineau95].

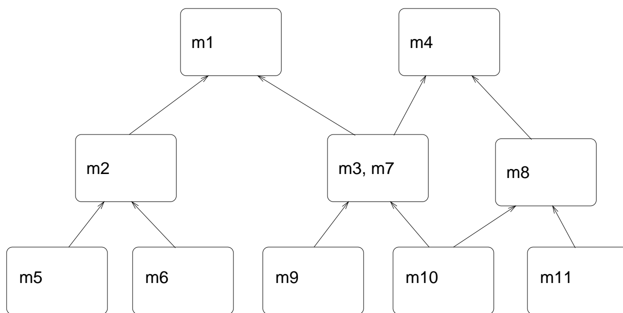

Figure 3: Restructured inheritance hierarchy

There have been some minor improvements to Guru since it was described in [Moore95], which will not be discussed in this paper as they do not affect the refactoring of methods.

3 Automatic Refactoring of Methods

This section describes the essential features of the refactoring of methods performed by Guru. Features of Self relevant to refactoring methods are introduced, followed by subsections which introduce refactoring of methods, describe limitations inherent to method refactoring, and describe the additional limitations imposed by Guru.

3.1 Self

Self [Ungar87] is a dynamically typed object-oriented language, similar in many ways to Smalltalk [Goldberg90], but with some differences which will need explaining in order for the Self code fragments to be understood by readers familiar only with Smalltalk. Self code fragments are shown in sans serif font. Messages without an explicit receiver are sent to self. Variable accesses and assignments are made using message sends. Therefore, the expression x aMessage: y means that aMessage: is sent to the object resulting from x sent to self, with the argument which is the object resulting from y sent to self. The messages x and y may refer to the Self equivalent of instance variables, or to methods. The message aMessage: may invoke either an assignment, or a method taking one argument.

In Self, methods, instance variables, data and inheritance relationships are defined in *slots*. Data slots are equivalent to non-assignable instance variables or class variables.

Self is an object-centric language; it does not have classes. However, the majority of readers will be more familiar with class-centric languages, so the word ‘class’ will be used for objects which perform the rôle equivalent to classes.

Self is dynamically typed; additional considerations which would need to be taken into account if the approach described were used for a statically typed language are beyond the scope of this paper.

3.2 Refactoring methods

Expressions can be factored out of methods by creating a new method to implement the expression, and by replacing occurrences of that expression by the appropriate message send. In this way, an expression can be shared by many methods. The terminology used in this paper is that expressions which are factored out are called ‘factored expressions’ and the methods created to share factored expressions are called ‘factoring methods’.

The behavior of a method is determined by its sequence of message sends. Provided the same messages are sent in the same order, the factoring of methods and the classes where they are located do not matter. Due to message sending polymorphism, a sequence of message sends may result in different methods being executed for different objects. If two methods (or expressions) send the same messages in the same order, then they can be factored out as one method (or expression), irrespective of the original classes (or methods) they occurred in. It does not matter which methods will execute as the result of those message sends, they will be the same irrespective of where the methods (or expressions) are located. Due to message sending polymorphism, more factoring is possible than in conventional languages which have only procedure or function calls. Note that in Self it does not matter whether a message send results in an instance variable access (or assignment) or a method being executed; this enables

more refactoring than otherwise.

The following code shows a simple example of refactoring a method:

```
methodOne = ( (x aMessage: y) something )
```

may be refactored as:

```
newMethod1 = (x aMessage: y)
methodOne = ( newMethod1 something )
```

provided that the object for which the latter version of methodOne is executed responds to the message newMethod1 by executing the implementation shown above. For the rest of the paper this condition is assumed to be valid, and Section 4 will describe how this condition is guaranteed by the refactoring system. For conciseness, some refactorings are shown for expressions occurring only once (as above).

3.3 Expressions which can be factored out

Not all expressions may be factored out of a method. For example, a block with a non-local return may not be factored out, because a non-local return is from a particular method; returning from a different method does not have the same effect. Similarly, assignments to local variables may not be factored out.

Expressions containing references to arguments and local variables may be factored out, provided the reference to the argument or variable is also given to the factoring method. For example:

```
methodOne: argument =
      ( | temporaryVariable |
            temporaryVariable: something.
            temporaryVariable result.
            (argument + aMessage) * 10)
```

may be refactored as:

```
newMethod1: temporaryVariable =
      (temporaryVariable result)
newMethod2: argument =
      ((argument + aMessage) * 10)
methodOne: argument =
      ( | temporaryVariable |
            temporaryVariable: something.
            newMethod1: temporaryVariable.
            newMethod2: argument)
```

Although references and assignments to variables *appear* identical to other message sends, it is straightforward to statically determine whether a message send is a reference or assignment to a local variable or a reference to an argument. In the above example, the expression temporaryVariable result is not worth factoring out, as its replacement newMethod1: temporaryVariable is no improvement. The system uses a simple metric based on the 'size' of an expression to determine whether to factor it out, so that cases such as this do not occur. Section 6 includes a discussion of whether this metric is adequate.

An expression which includes a block containing assignments to local variables of that block *may* be factored out, as shown in the following example.

```
methodOne = (| temporary |
            do: [ | :e. t |
                  t: e aMessage.
                  t something: temporary ].
            some other code)
```

can be refactored as:

```
newMethod1: temporary = (do: [ | :e. t |
                              t: e aMessage.
                              t something: temporary ])
methodOne = (| temporary |
            newMethod1: temporary.
            some other code)
```

However, the expression t: e aMessage inside the block cannot be factored out by itself.

A more subtle restriction is imposed by the implementation of Self used[2]. Blocks by themselves may not be factored out; they may only be factored out as part of an expression. For example, the expression [a b c] value may be factored out, but the block [a b c] by itself may not. The reason for this limitation is that blocks which execute after their enclosing method has returned (called non-lifo blocks) are not supported by the current implementation of Self. This restriction is very

[2]Sun Microsystems Laboratories, Self version 4.0

minor, as the expressions inside blocks are refactored; thus, in the example above, the expression a b c may be factored out of the block.

3.4 The expressions factored out by Guru

A limitation of the current implementation of Guru is that, with the exception of expressions containing references to arguments or local variables of their enclosing method or block, only complete expressions (i.e. a piece of code which evaluates to an object) are factored out. References to arguments or local variables are passed to factoring methods as described in the previous section. Expressions which are message sends to implicit self, explicit self or literals, or messages sends to such expressions, to any depth, can be factored out. Expressions which occur more than once will be factored out, including expressions repeated in the same method, and repeated subexpressions of the same expression. Furthermore, the system refactors expressions within factoring methods.

An example of the method of factoring that Guru uses is that the expressions a b c and a b d can be factored into newMethod1 = (a b), so that a b c would become newMethod1 c and a b d would become newMethod1 d.

There are many other different ways of refactoring. One example is refactoring the expressions a b c and x b c. Guru can only refactor these expressions in limited circumstances; but this is a limitation of Guru rather than this method of refactoring. A factoring method newMethod1: a = (a b c) could be created, and the expressions a b c and x b c would then become newMethod1: a and newMethod1: x respectively. This refactoring is performed by Guru only when a and x are references to arguments of their enclosing methods.

In situations such as the example above, the names of messages which represent references to arguments are ignored when comparing expressions, otherwise the two expressions would have to be identical. As a consequence, the names of method arguments also have to be ignored when comparing methods, otherwise unnecessary methods could be created, for example, if both newMethod1: a = (a b c) and newMethod2: x = (x b c) were created. This comparison of methods is used for all methods, not just factoring methods.

Two other ways that the expressions a b c and x b c could be factored are presented below.

If the two expressions occur in methods in different classes which had a common superclass, then a factoring method newMethod1 = (newMethod2 b c) could be created in the superclass, and the methods newMethod2 = (a) and newMethod2 = (x) could be created in the appropriate subclasses.

Alternatively, the expressions could be factored out by creating factoring method newMethod1 = (b c) in the classes of the objects which may result from the message sends a and x, and replacing the expressions with a newMethod1 and x newMethod1 respectively. If the message sends a and x always result in objects of only one class, then this would be a very good solution. However, determining the classes of the objects which may result from the message sends a and x may be extremely difficult as it requires precise type inferencing [Agesen95]. Furthermore, it may be that these message sends result in objects of many different classes. If these classes were not restructured, then the refactoring would increase the number of methods in the system, which would be counter-productive.

A consequence of Guru only factoring complete expressions is that two expressions which are the same except for a small difference are not refactored. For example, the expressions a b: (c d: e) and a b: (c d: f) cannot currently be factored as newMethod1: e = (a b: (c d: e)) with the expressions becoming newMethod1: e and newMethod1: f respectively, unless e and f are arguments of their enclosing method. Furthermore, the largest expressions which can be factored out are individual statements. If consecutive statements are shared by two or more methods, the statements are factored out as separate factoring methods, rather than as a single factoring method.

Currently, Guru only refactors in limited ways.

There are many other ways that methods and expressions could be refactored, in addition to the ways discussed above. Section 7 discusses the possibilities for extending Guru to refactor in different ways. In practice, even with the limitations described, the system performs a considerable amount of refactoring, as discussed in Section 6.

4 Simultaneous refactoring of inheritance hierarchies and methods

The refactoring of methods is performed as part of inheritance hierarchy restructuring for two reasons. Firstly, all of the methods in all of the objects and classes are refactored together, which achieves the highest possible amount of method refactoring. There is no limitation on refactoring of methods that they have to be related by inheritance (before the refactoring stage; they will necessarily be related by inheritance after refactoring and inheritance hierarchy restructuring). Secondly, the refactoring of methods can discover classes and inheritance relationships which would not necessarily otherwise exist. That is, if two classes share expressions, even if they do not share any methods, then they will be related by inheritance in the resulting inheritance hierarchy. This is further discussed in Section 6.

The algorithm used to refactor methods is described below, using an example.

Consider the three methods:

```
m1 = (((size + 1) > end)
            ifTrue: [something])
m2 = (((start + end) > 0)
            ifTrue: [size: size + 1])
m3 = (size: size + 1.
      ((start + end) > 0) ifFalse: [error])
```

A dictionary is created which relates methods to *all* of the expressions they contain which may be factored out by Guru.

```
m1  →  size + 1, (size + 1) > end
m2  →  start + end, (start + end) > 0,
       size + 1, size: size + 1
m3  →  size + 1, size: size + 1, start + end,
       (start + end) > 0
```

For brevity, the largest expressions (((size + 1) > end) ifTrue: [something], ((start + end) > 0) ifTrue: [size: size + 1] and ((start + end) > 0) ifFalse: [error]) have been omitted, and will not be shown in the rest of this example. Another dictionary is created which relates expressions to the methods which contain them.

```
size + 1            →  m1, m2, m3
(size + 1) > end    →  m1
start + end         →  m2, m3
(start + end) > 0   →  m2, m3
size: size + 1      →  m2, m3
```

From this, a dictionary is created which relates collections of methods to the expressions which they share.

```
m1, m2, m3  →  size + 1
m1          →  (size + 1) > end
m2, m3      →  (start + end) > 0,
               start + end, size: size + 1
```

In order to avoid unnecessary refactoring, subexpressions of those expressions shared by exactly the same set of methods are not factored out. Hence, in the example above, the expression start + end is not factored out, as (start + end) > 0 is also shared by the same methods. Expressions which appear only once, such as (size + 1) > end in the example, are not factored out. Having determined which expressions should be factored out, a factoring method is created for each factored expression, with a unique name. In the example above, this means that the following methods are created:

```
newMethod1 = (size + 1)
newMethod2 = (size: newMethod1)
newMethod3 = ((start + end) > 0)
```

A replacement method is made for each method which includes any factored expressions. These replacement methods are modified such that each factored expression is replaced by the appropriate message send to invoke the appropriate factoring

method.

The resulting methods are:

```
m1 = ((newMethod1 > end)
          ifTrue: [something])
m2 = (newMethod3 ifTrue: [newMethod2])
m3 = (newMethod2.
      newMethod3 ifFalse: [error])
```

These modified methods and the factoring methods for all of the factored expressions that they include effectively replace the original methods in the objects and classes to be restructured. For example, the method m1 = (((size + 1) > end) ifTrue: [something]) is replaced by m1 = ((newMethod1 > end) ifTrue: [something]) *and* newMethod1 = (size + 1).

The restructuring is then performed on the objects and classes with their refactored and factoring methods, as described in Section 2, exactly the same as if there had not been any refactoring. As the appropriate factoring methods are included in the objects and classes which include methods containing their factored expressions, the restructuring ensures that the appropriate factoring methods will be in the appropriate restructured classes. A factoring method will be located in the class from which all methods which included its factored expression inherit. In other words, if an expression is factored out of methods from only one class, then the factoring method will be in the same class as those methods. Alternatively, if an expression is factored out of methods from several different classes, then it will be in a class from which all those classes inherit, directly or indirectly. Therefore, the appropriate factoring method will definitely be executed by a message send of its name, because the names of such methods are unique, and factoring methods are inherited by every class or object which includes a method which contains such a message send.

Objects and concrete classes will be slightly changed by the refactoring version of Guru, because they will understand the messages implemented by the factoring methods they inherit. However, the introduction of factoring methods does not cause any problems, as the behavior of programs will not be changed. It is straightforward to check, within the limitations discussed below, that the system does not contain any message sends which would execute factoring methods inappropriately and does not implement any methods with the same names as any of the factoring methods. This check needs to be performed on the complete Self system, as an object which is not included in the collection of objects refactored may send a message to a refactored object which it previously could not understand, but after refactoring it can understand it, thus changing the behavior of a program. Programs which rely on an object *not* understanding a certain message are unusual, so this check will rarely be necessary. In Self, it is possible to write code which sends a message which cannot be determined statically, thereby defeating any attempts to statically check whether a certain message is sent, but this programming style is unusual.

5 Results

The results of applying Guru to three inheritance hierarchies are presented in this section. The three inheritance hierarchies, which will be called the indexables, orderedOddballs and sendishNodes hierarchies respectively, were restructured using Guru, both with and without refactoring of methods. In all except one restructuring, Guru was used fully automatically. Restructuring of inheritance hierarchies is abbreviated to *restructuring*, and refactoring of methods is abbreviated to *refactoring*.

Two of the hierarchies, the indexables and orderedOddballs, were chosen because it was expected that they would already be well designed and well factored, and so would provide a good benchmark for evaluating the performance of Guru. Both of these hierarchies were designed before Guru existed (by someone other than the author), and hence their design could not have been influenced by the existence of Guru.

Both the indexables and orderedOddballs hierarchies are used extensively during the running of the Self system, as the programming environment is written in Self. Some of the objects and

classes are fundamental to the running of nearly all Self code. In particular, vectors, byteVectors, canonicalStrings (the Self equivalent of Symbols in Smalltalk), smallInts, floats, true and false are *extensively* used. The restructured hierarchies, with and without refactoring, were used to replace the original hierarchies, with no change in the behavior of the system. While the successful replacement of such fundamental classes is not a formal proof that the restructurings are correct, it gives *substantial* evidence.

The sendishNodes hierarchy was chosen because it is known to be imperfect, and as it is part of the Guru system itself, its design has been affected by the existence of Guru, which is what we should expect for hierarchies which are developed when a system such as Guru is available in a programming system. This point is further discussed in Section 6.

In the figures, restructured objects and classes are labeled either with the name of the object or class they replace, or with the name of the class they *most closely* replace. What is meant by this is that, for example, the class labeled traits string in Figure 5 does not necessarily define the same behavior as the original class traits string. Only the behavior of objects and concrete classes is preserved, and traits string is not a concrete class. The label traits string in the restructured hierarchy is used only for convenience, and reflects the fact that this class is inherited by the equivalent objects and concrete classes in the original hierarchy. Restructured objects and classes which cannot be labeled in this way are not named. All objects and classes are shown with the number of non-inheritance slots they define. Hierarchies restructured with and without refactoring which have the same structure are shown together. In those figures, the first number is the number of slots without refactoring, and the second is the number with refactoring.

Tables of simple metrics are presented. The entries labeled 'Message sends' refer to the total number of potential message sends in all the methods in all the classes in the appropriate hierarchy. This metric is used as an indication of the total code size of the hierarchies. Counting the number of methods, the number of statements, or the 'lines of code' does not really measure the amount of code. Such measures are misleading for large, badly factored methods or statements, or code written in very long lines. Measuring the number of potential message sends gives a more accurate indication of the amount of code.

The entries labeled 'Overriding methods' are the number of methods which override other methods both from inside and outside the classes restructured. Having too much overriding, or methods overridden too often, is an indication of poor design [Johnson88].

5.1 The indexables hierarchy

This hierarchy includes strings, vectors and sequences. A very similar hierarchy was restructured by the previous version of Guru, and the results are described in [Moore95]. The differences between the hierarchy restructured by Guru, shown in Figure 4, and the hierarchy restructured in [Moore95], are because a newer version of the Self system has been used.

The result of restructuring without refactoring is shown in Figure 5. This hierarchy is not exactly the same as in [Moore95], because of the differences in the original hierarchy and some minor improvements to Guru since it was described in [Moore95].

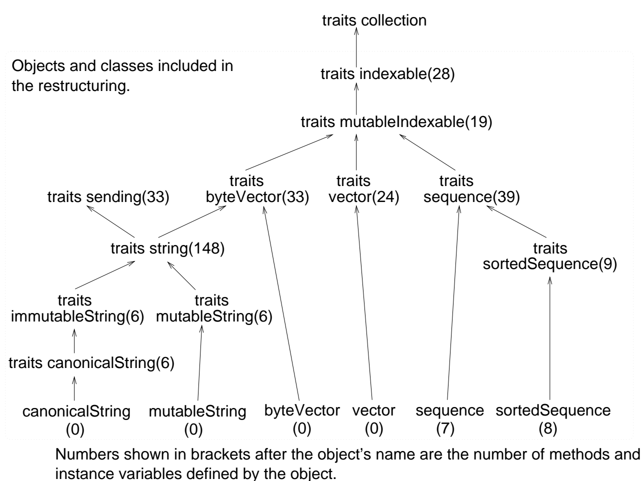

Figure 4: The original indexables hierarchy

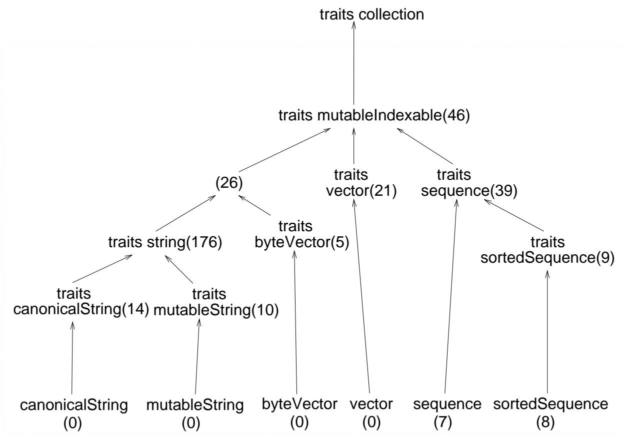

Figure 5: The restructured indexables hierarchy

This restructuring included a very small amount of programmer intervention, in that one method was manually moved higher in the inheritance hierarchy. The original hierarchy was then restructured with refactoring, producing the hierarchy shown in Figure 6.

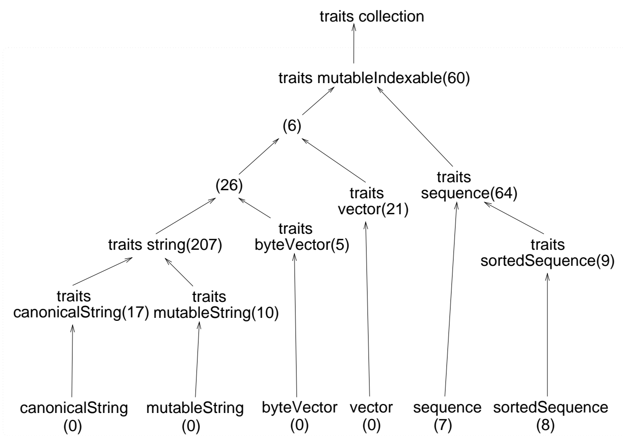

Figure 6: The restructured and refactored indexables hierarchy

In this case, there was no manual intervention. It is interesting to observe that the hierarchy is slightly different to the one produced by the restructuring without refactoring. An additional class has been discovered because of the factoring methods introduced. Furthermore, this additional class is very easy to understand; it is the class of vector-like objects, as opposed to sequence-like objects.

The following table gives some simple metrics concerning the refactoring and restructuring results. (Ori = Original, Res = Restructuring without refactoring, RwR = Restructuring with refactoring)

	Ori	Res	RwR
Classes and Objects	17	15	16
Methods	316	311	390
Message sends	3681	3662	3622
Overriding methods	86	72	69

A small problem with replacing the original objects with the restructured ones was that, because vector, byteVector, mutableString and canonicalString have the equivalent of indexed instance variables, only their class objects were actually modified. This has exactly the same effect as if these objects are modified, as they each have only one slot defining inheritance from their class.

5.2 The orderedOddballs hierarchy

This hierarchy includes the numbers and boolean classes. Although Guru was designed to restructure hierarchies including concrete objects, it can equally be used when given only the Self equivalent of classes, as in this case. Only classes have been used, because number objects cannot be modified; furthermore, nothing would have been gained by including number objects in the restructuring as they define only one slot for inheritance.

The original hierarchy is shown in Figure 7. The results of restructuring with and without refactoring methods are both shown in Figure 8.

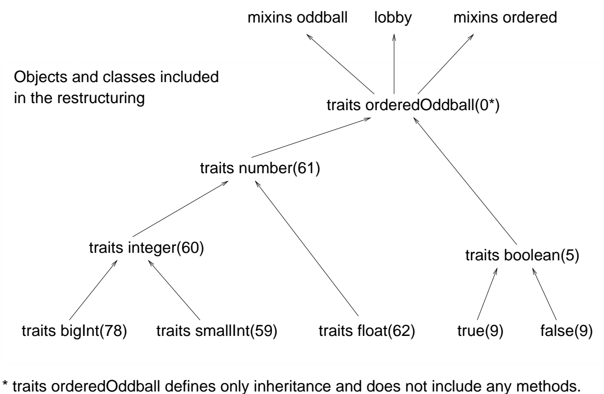

Figure 7: The original orderedOddballs hierarchy

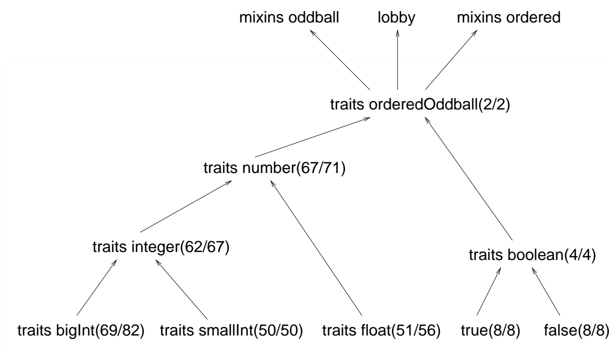

Figure 8: The restructured orderedOddballs hierarchy, with and without refactoring of methods

Notice that there is no difference in the structure of the hierarchies, only in the details of where methods are located and how they are factored. This is not generally true of hierarchies restructured using Guru, but in this case the hierarchy can be assumed to have been very well designed initially, as such hierarchies are well understood, and it is a small hierarchy, in terms of the number of classes.

An example of the detailed difference between the hierarchy restructured without refactoring and the original is that there were several methods which had been defined identically in **traits bigInt**, **traits smallInt** and **traits float**, for which in the restructured inheritance hierarchy a single implementation has been put in the equivalent of **traits number**.

In the hierarchy restructured with refactoring there are more detailed differences. For example, in the equivalent of **traits number**, one expression was factored out from 14 methods, and shared between them using a factoring method. Some methods in the replacement for **traits float** had the same implementation but different names, and so their code was shared using factoring methods.

The following table provides some simple metrics about the original hierarchy, and the restructured hierarchies with and without refactoring.

	Ori	Res	RwR
Classes and Objects	9	9	9
Methods	323	301	328
Message sends	2539	2493	2464
Overriding methods	35	29	29

5.3 The sendishNodes hierarchy

This hierarchy is part of the parse tree nodes hierarchy used in the implementation of Guru.

The original hierarchy is shown in Figure 9, the restructured hierarchies with and without refactoring of methods are shown in Figure 10.

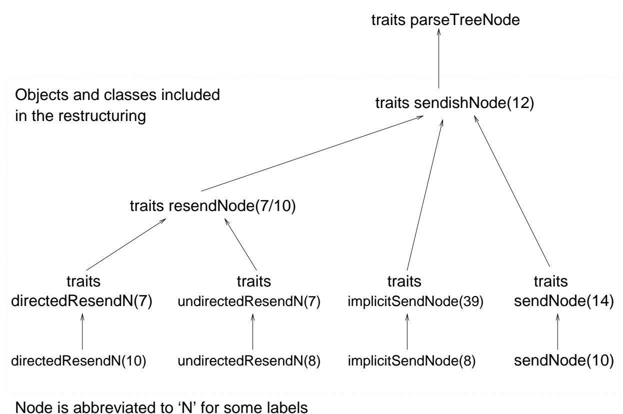

Figure 9: The original sendishNodes hierarchy

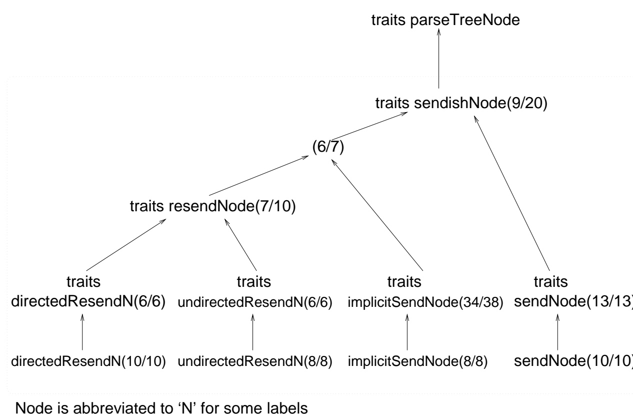

Figure 10: The restructured sendishNodes hierarchy, with and without refactoring of methods

The restructured hierarchies are similar to the original hierarchy, with the difference in structure due to the discovery of a new class. This class can easily be understood as the class of all parse tree

nodes which represent expressions that have self as the implicit receiver (including resends).

The following table provides some simple metrics about the original classes, and the classes restructured with and without refactoring.

	Ori	Res	RwR
Classes and Objects	10	11	11
Methods	77	70	89
Message sends	490	461	429
Overriding methods	37	29	29

6 Discussion of results

The most important feature of the results is that the structure of the inheritance hierarchies produced by Guru are exactly as expected for well designed hierarchies. It is important to remember that Guru does not use the structure of the original hierarchies to guide the creation of the restructured inheritance hierarchies. The hierarchies produced are based only on maximizing sharing and minimizing duplication of the features (mostly methods) of objects and concrete classes. Any other hierarchies which defined the same features for their objects and concrete classes, however badly structured, would have produced the same results from Guru. It should be noted that Guru will produce inheritance hierarchies with multiple inheritance when necessary; for the three examples used, single inheritance was sufficient to ensure no duplication of methods or factored expressions.

It is interesting that method refactoring has not had more effect on the structure of the inheritance hierarchies produced. While the approach taken maximizes the amount of factoring of methods (within the limitations of Guru), the benefit of inferring new classes from factoring methods would appear, on the evidence gathered so far, to be of minor importance.

A more detailed feature of the hierarchies restructured with refactoring was that a high proportion of method refactoring was found to be inside individual restructured classes, with *relatively* few factoring methods sharing expressions amongst methods of more than one class. The explanation for this, given that refactoring of methods is performed before the final restructured hierarchy is created, must be due to the fact that methods that will be restructured into in a single class will tend to have more similarities between them than methods in different classes.

The largest difference between the original and restructured hierarchies is in the details of the location and factoring of methods. The restructured and refactored hierarchies are an improvement compared to the original hierarchies. In terms of simple objective measures, the reduction in the number of potential message sends in the restructured and refactored hierarchies indicates that the total amount of code has been reduced, and an improvement in code reuse has been achieved. Furthermore, as no methods or factored expressions are now duplicated, but are defined only once, the restructured and refactored hierarchies are more consistent than the original hierarchies.

The results of the previous section show that, despite the limitations described in Section 3.4, the system performs a considerable amount of refactoring. In the case of the indexables hierarchy, 77 expressions were factored out. There is scope for improvement, particularly in allowing for more refactoring to be *possible*, while only performing refactoring which is *desirable*.

One of the problems of automatic refactoring is that it can be difficult to understand the meaning of some of the methods automatically created. The names generated for factoring methods are unique system generated names which have no inherent meaning. These methods can be renamed by the user, and Guru will rename all sends of the appropriate message. It can be difficult to invent a short and meaningful name for many of the factoring methods. To give an indication of the sort of expressions which were factored out from the orderedOddballs and indexables hierarchies, a selection of some of the factoring methods, and one method which was modified to use a factoring method, are shown below (including too many brackets, this is one of the features which could

be improved, as mentioned in Section 7):

```
newMethod651P: a P: res =
        (res sign: (sign * (a sign)))
newMethod627 = (-1 = sign)
newMethod659P: d = ((d size) - cByteSize)
mostSignificantDigit: d =
        (in d At: (newMethod659P: d))
newMethod636P: digit =
        ((digit asByte) - ('0' asByte))
newMethod661 = (size + 1)
newMethod695 = (size: (newMethod661))
```

The purpose of a factoring method may not be obvious, unless one fully understands the code. It is currently impossible for a fully automated system to determine the purpose of a fragment of code to decide whether it is worth refactoring or not, and to invent a meaningful name for the factoring method. Furthermore, the amount of refactoring that should be performed can be argued to be a subjective decision. For example, in Self some programmers use the expression x + 1, where others use the equivalent expression x succ. A possible argument for limiting the amount of factoring is that some programmers may understand the meaning of, for example, x + 1 more easily than x succ, and in some cases may have to find the factoring method that will be executed in order to understand the code. Furthermore, it may be very difficult to think of a name for a factoring method which is understandable, and at the same time more abstract and preferably also more compact than the original expression. For example, the expression size + 1 could be factored out as a factoring method called sizePlusOne. This is slightly more characters to type, and is not very abstract. The ideal refactoring should discover meaningful method abstractions from expressions, so that the system is easier to understand, reuse and modify. An approach that may satisfy both those who prefer as much factoring as possible and those who do not, would be for the code to be as highly factored as possible, but for the system to allow expressions to be shown as if they were inlined in the code, as much as each programmer requires. To implement such a facility would, in general, require precise type inferencing [Agesen95].

The metric used by Guru to decide whether to factor out an expression is based simply on the 'size' of the expression, measured as the number of message sends, not including references to arguments or local variables. The minimum 'size' of expression that will be factored out has been chosen as the smallest that ensures that replacement expressions will be smaller than the expressions they factor out (in terms of number of message sends); hence the total number of message sends will be reduced by the refactoring. Detailed analysis of the results of refactoring several hierarchies indicates that this metric is not ideal, as many expressions are factored out which have no easily understood meaning as method abstractions. A metric which could be used, that may give a better indication of which expressions to factor out, could be the number of times an expression occurs. Currently, any expression which occurs more than once is factored out, but it may be better to factor out certain expressions, for example the smallest possible expressions, only if they occur frequently.

A possible criticism of increasing factoring is that it will degrade the performance of the code. However, this is a weak argument, as sophisticated compilers, such as the Self system used for this work, are able to automatically inline code so that the amount of factoring at the source level does not affect the performance of compiled code.

In the sendishNodes hierarchy, some methods were written in a particular way in order to maximize their possibility of refactoring. In particular, consistency between methods in the order of expressions, in cases where the ordering does not matter for the meaning of the expression, enabled some subexpressions to be factored out which would not have otherwise been possible. The existence of a tool such as Guru can influence the way that code is written. There is possibly a counter-productive psychological consequence of the existence of a system such as Guru, that programmers may become lazy about creating good code and inheritance hierarchies, because they believe that the system will 'tidy everything up' for

them.

7 Future research

More work is required to reduce the limitations of Guru's refactoring. If the system is made capable of factoring methods or expressions in different ways, then it will have to be able to decide which refactoring to use. Similarly, as more factoring becomes possible, the system will either have to decide whether a potential refactoring is worth applying, given that some programmers will not appreciate too much refactoring, or it should provide the ability to display source code as if inlining has been performed, as described in the previous section. Also, there may be a limit to the amount of refactoring that can be performed in an acceptable amount of time.

A drawback of Guru which needs improvement is that the source code of factoring methods and methods which have been modified because they included a factored expression is generated from parse trees, ignoring the original source code. This results in the original layout and comments being lost, and most programmers would prefer as little disruption to their layout and comments as possible. In the present system, the source code generated is not very well formatted; for example it uses too many brackets.

The speed of Guru for restructuring including refactoring of large systems is poor in the current implementation. Restructuring including refactoring of the indexables hierarchy took approximately 8 hours on a Sun Sparc 2. This is mostly because of details of the current implementation of Guru, but partly due to the fact that all methods in a restructuring are refactored together. While this approach ensures the maximum amount of refactoring is possible, it is computationally expensive. One way of reducing the computational expense would be to restructure a hierarchy without refactoring, and then perform refactoring on subgraphs of the restructured hierarchy. This approach would not achieve the maximum amount of factoring possible, and would not discover any new classes or inheritance relationships in performing the refactoring. The results discussed in Section 6 suggest that if the time taken to refactor a large system were important then the loss of these benefits might be a reasonable compromise. However, the limit on the size of hierarchy which can reasonably be restructured including refactoring may be reached before the limit imposed by computational considerations. This is because of the complexity of understanding a large hierarchy well enough to understand the effects of the restructuring.

More refactoring could be possible by using a more sophisticated comparison of expressions than simply examining their sequence of message sends. For example, by inlining expressions, superficially different expressions or methods which have the same effect could be refactored [Ungar94]. Furthermore, if it can be determined which methods are private (only executed due to messages sent to self) then these methods could be removed if all message sends which cause them to execute are removed through inlining. This would allow better refactoring of the public methods. Similarly, it might be possible to determine that expressions are equivalent even if the order of message sends is different. Sophisticated analysis of code is increasingly performed by optimizing compilers; a refactoring system could benefit from reusing the analysis of such compilers.

The successful application of Guru to objects and classes fundamental to the Self system is a practical demonstration that the restructuring and refactoring performed do not alter the behavior of a system. However, proofs should be constructed in order to verify that the algorithms used by Guru preserve the behavior of a system.

8 Conclusions

This paper has shown that automatic restructuring and refactoring can improve the inheritance hierarchy structure and method factoring of realistic examples.

Hierarchies created by restructuring realistic examples are exactly the structures that should

be expected of good designs, and eliminate duplication of methods. The refactoring of methods improves hierarchies even further by eliminating duplication of the expressions which it factors out. Eliminating duplication of methods and factored expressions reduces the total amount of code (measured as the number of potential message sends), improves consistency and increases code reuse. Further work is required to increase the amount of refactoring possible, and to produce more easily understood factoring methods.

Guru does not directly approach the related problems of (user directed) program maintenance or program understanding. However, program understanding may be assisted due to reducing the size and increasing the consistency of a system. Similarly, Guru provides an approach to perfective maintenance which may simplify user directed maintenance due to this improved consistency, which makes alterations easier and safer, as it ensures that a change needs to be made in only one method, rather than having to make the same change in many methods.

The structure of the hierarchies produced by Guru is either the same, or better, than the original programmer designed hierarchies, providing objective evidence that maximizing factoring is a good design principle. The comparison of an original hierarchy with its restructured hierarchy is suggested as a quality metric.

As the system uses only details of code in the system, rather than knowledge of the problem domain, the hierarchies and refactoring created by Guru reflect what *actually exists* in a system, which may not be the same as what *should exist* in a system.

Acknowledgements

I would like to thank the EPSRC for funding this work. I am very grateful to my ex-supervisors, Trevor Hopkins, Mario Wolczko, Jon Taylor and Tim Clement and my current supervisor Chris Kirkham, for their encouragement and assistance during my studies at Manchester University. I am also indebted to the Self group, for advice and informative discussions about the Self system.

References

[Agesen95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings of ECOOP 95.* (LNCS 952, pages 2-26) Springer-Verlag, 1995.

[Casais90] Eduardo Casais. Managing Class Evolution in Object-Oriented Systems. In *Object Management.* Centre Universitaire d'Informatique, Geneve, 1990.

[Casais92] Eduardo Casais. An Incremental Class Reorganization Approach. In *Proceedings of ECOOP 92.* (LNCS 615, pages 114-132) Springer-Verlag, 1992.

[Dicky96] Herve Dicky, Christophe Dony, Marianne Huchard and Therese Libourel. On Automatic Class Insertion with Overloading. In *Proceedings of OOPSLA*, 1996.

[Godin93] Robert Godin and Hafedh Mili. *Building and Maintaining Analysis-Level Class Hierarchies using Galois Lattices.* In *Proceedings of OOPSLA*, (SIGPLAN Notices 28(10), pages 394-410) 1993.

[Goldberg90] Adele Goldberg and David Robson. *Smalltalk-80 : The Language.* Addison-Wesley, 1990.

[Hoeck93] Bernd H. Hoeck. *A Framework for Semi-Automatic Reorganisation of Object-Oriented Design and Code.* MSc thesis, University of Manchester, 1993.

[Johnson88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming* 1(2), pages 22–35, 1988.

[Lano93] Kevin Lano and Howard Haughton. *Reverse Engineering and Software*

Maintenance: A Practical Approach. McGraw-Hill, International Series in Software Engineering, 1993.

[Lieberherr88] Karl J. Lieberherr, Ian Holland, and Arthur J. Riel. Object-Oriented Programming: An Objective Sense of Style. In *Proceedings of OOPSLA*, (SIGPLAN Notices 23(11), pages 323-334) 1988.

[Lieberherr91] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. From Objects to Classes: Algorithms for Optimal Object-Oriented Design. *Software Engineering Journal* 6(4), pages 205-228, 1991.

[Mineau95] Guy W. Mineau and Robert Godin. Automatic Structuring of Knowledge Bases by Conceptual Clustering. In *IEEE Transactions on Knowledge and Data Engineering* 7(5), pages 824-829, 1995.

[Moore95] Ivan R. Moore. Guru - a Tool for Automatic Restructuring of Self Inheritance Hierarchies. In *TOOLS USA 1995.* (TOOLS 17, pages 267-275) Prentice-Hall, 1995.

[Moore96] Ivan R. Moore and Tim P. Clement. A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies. In *TOOLS Europe 1996.* (TOOLS 19, pages 173-184) Prentice-Hall, 1996.

[Opdyke92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Opdyke93] William F. Opdyke and Ralph E. Johnson. *Creating Abstract Superclasses by Refactoring.* In *Proceedings of CSC '93: The ACM 1993 Computer Science Conference.* ACM, 1993.

[Pun89] Winnie W. Y. Pun and Russel L. Winder. Automating Class Hierarchy Graph Construction. Technical report, University College London, 1989.

[Pun90] Winnie W. Y. Pun. *A Design Method for Object-Oriented Programming.* PhD thesis, University College London, 1990.

[Ungar87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA*, (SIGPLAN Notices 22(12), pages 227-241) 1987.

[Ungar94] David Ungar. Private communication. Sun Microsystems Laboratories Inc, Mountain View, California 1994.

[Wolff94] J. Gerard Wolff. Towards a New Concept of Software. In *Software Engineering Journal* 9(1), pages 27-38, 1994.

A The hierarchy restructuring algorithm

This section briefly describes the important features of the algorithm used for inferring an inheritance hierarchy, called the IHI algorithm, for a set of objects/classes. It is more fully described in [Moore96]. To simplify the text, the word 'class' is used to mean 'object or class'.

Given a set of classes, each class defining a set of features (such as instance variables and methods, without any inheritance links), the IHI algorithm infers a hierarchy with no duplication of features, which includes replacement classes defining or inheriting exactly the same sets of features as defined by the original classes. Let O be the set of classes for which a hierarchy is to be inferred. Let R be the set of classes inferred.

In order to ensure no duplication of features, a relation is built mapping each set of all the classes in O which share a feature, to the set of features they (alone) share. This requires that features define equivalence. A set of features may contain only one feature. Similarly, a set of features defined by only one class requires a mapping from a set containing that class only, to the set of features defined. Such single element sets need to be

constructed for classes even if there is no feature that they alone define (in which case the set of features defined will be empty). Each mapping from a set of classes to a set of features defines a class in R (with that set of features), but does not define the necessary inheritance links. Each mapping from a set containing a single class to a set of features defines a class in R which directly replaces a class in O (the class which is the single element of the set).

To ensure that replacement classes inherit all the features they need, inheritance links are added from each class in R which shares features between classes C, to every class in R which shares features between a proper superset of C. In order to remove transitively unnecessary inheritance, for each class C in R, for each class D which has an inheritance link to C (before any inheritance links have been removed), and each class E which has an inheritance link to D, remove any inheritance links from E to C.

The classes in R, and their inheritance links, are now defined.