Guru - A Tool for Automatic Restructuring of Self Inheritance Hierarchies

Ivan Moore

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

Abstract

This paper[1] introduces Guru, a prototype tool for restructuring inheritance hierarchies in Self, while preserving the behavior of objects. Guru reverse engineers from existing inheritance hierarchies. Unlike previous work, Guru handles resends, redefined methods and the restructuring of only part of a system. Furthermore, Guru handles dynamic and cyclical inheritance, which are more specific to Self. Guru removes duplicated methods, and can create inheritance hierarchies with no overridden methods. The results of two non-trivial tests are presented and assessed.

1 Introduction

Some object-oriented design methods encourage the designer to think of the inheritance hierarchy very early in the production of a software system. Many developers think of the inheritance hierarchy as basically static; they will add new classes, but are reluctant to restructure it. This is not surprising, as restructuring inheritance hierarchies is difficult. Guru is a prototype reverse engineering tool which can automatically restructure an inheritance hierarchy into an optimal one for the objects currently in the system, whilst preserving the behavior of programs.

The optimal inheritance hierarchy for a system depends on the current design of the system, which in turn depends on the current requirements of the system. It is impossible to predict future requirements. The software development process is iterative, whether it is planned to be or not. Therefore, as the design of the system changes to satisfy changing requirements, the inheritance hierarchy will require restructuring if it is to remain optimal. By providing a tool to automate restructuring, it is hoped that more frequent restructuring is encouraged and made feasible. Guru is designed so that it can be used on part of a system, rather than having to restructure an entire system. Guru can be used automatically but is more usefully employed as part of a broader approach.

Guru can only restructure non-reflective systems. Reflective programs depend on the structure of their objects. Changing the structure of objects involved in a reflective program can change the behavior of those reflective programs. As Guru is intended not to change the behavior of programs, it can only be used safely for non-reflective systems.

Guru has been developed in Self[Ungar 87], for restructuring Self systems. Self was chosen as the language to investigate object-oriented program restructuring because object manipulation in Self is powerful and easy to use. Furthermore, Self is a simple, consistent and powerful language. Other work[Casais 90][Hoeck 93]-[Lieberherr 91][Opdyke 92][Pun 89] uses the term 'class hierarchy', but as Self is classless, the term 'inheritance hierarchy' is used in this paper. The restructuring system has been called Guru, because it assists in Self improvement.

[1]published in TOOLS USA 1995, Prentice-Hall

This paper introduces some of the features of Self, and inheritance hierarchy restructuring (IHR). This is followed by descriptions of some of the details of how Guru restructures Self inheritance hierarchies. Finally, there are sections covering the limitations and problems of the approach taken, the results obtained, future work to improve Guru, and concluding remarks.

1.1 Self

Self[Ungar 87] is an object-centric programming language, originally developed at Stanford University and Xerox PARC (by David Ungar and Randall Smith), and more recently at Sun Microsystems Laboratories.

Self is dynamically typed, which means that the inheritance hierarchy is not used as a typing hierarchy, i.e. to constrain the compatibility of objects, but rather as a means of sharing behavior or data.

Self objects contain methods, data and their inheritance links in *slots*. Slots used for inheritance links are called parent slots. Data slots and parent slots can be either read only, or read and write (also called assignable slots). Assignable parent slots are used to implement *dynamic inheritance*.

Self is object-centric, that is, there are only objects, not classes and objects. Objects inherited from are known as 'parents' of the objects which inherit from them. Parent objects are the same as any other objects, and an object can inherit from any other object. Self even allows cycles in the inheritance hierarchy; an object can inherit from an object which inherits from itself. There are multiple roots of the inheritance hierarchy. An object does not have to inherit from any other object; it can be totally self-contained.

Objects are created by copying, rather than by making an instance of a class, thus removing any necessity for classes.

In application programs, reflection is discouraged, and separated from normal objects into meta-objects called mirrors. What is most important about an object is how it responds to messages, rather than anything to do with how it is implemented, including its structure.

1.2 Inheritance Hierarchy Restructuring

The simplest way to describe the IHR done by Guru is through an example. The inheritance hierarchy shown in Figure 1 can be restructured by Guru into the inheritance hierarchy shown in Figure 2. Other restructurings are possible, but Guru aims to produce 'optimal' inheritance hierarchies, in that no slot is duplicated and there are the minimum number of objects and inheritance relationships required for such an inheritance hierarchy. Fewer objects and fewer inheritance relationships are possible if slots are duplicated. In the figures, slots with the same name have the same value unless stated otherwise. Note that slots m1, m7 and m8 are duplicated in the example shown in Figure 1, but the restructured inheritance hierarchy is 'optimal' in that no slots are duplicated. The objects in the restructured inheritance hierarchy o1# to o5# have the same behavior as the objects o1 to o5, and so can be used to replace them. Objects which have the same behavior after restructuring are called 'preserved' objects in the rest of this paper. Section 2.1 explains why objects o1 to o5 are the only preserved objects.

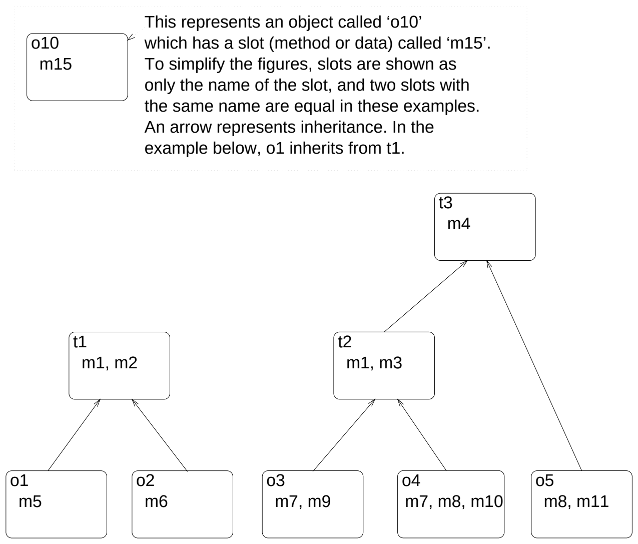

o10
m15
This represents an object called 'o10' which has a slot (method or data) called 'm15'. To simplify the figures, slots are shown as only the name of the slot, and two slots with the same name are equal in these examples. An arrow represents inheritance. In the example below, o1 inherits from t1.
t3
m4
t1
m1, m2
t2
m1, m3
o1
m5
o2
m6
o3
m7, m9
o4
m7, m8, m10
o5
m8, m11

Figure 1: Example inheritance hierarchy

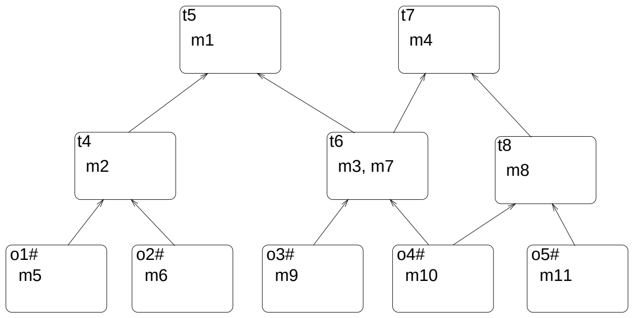


Figure 2: Example restructured inheritance hierarchy

2 Guru's restructuring process

This section describes how Guru performs IHR; by determining which objects need to be preserved, building restructured objects, and relating them back to the original objects. Details of how Guru handles dynamic inheritance and cycles in the inheritance hierarchy are beyond the scope of this paper, as these features are not common in object-oriented languages other than Self.

2.1 Identifying which objects to preserve

In order for the restructuring to preserve the behavior of the system, all of the objects which must be preserved need to be identified. In order to benefit from as much restructuring as possible, *only* those objects which *must* be preserved should be identified. Therefore, the preserved objects identified should include all necessary objects but *no more*, in order not to limit the number of objects which can be restructured and hence the effectiveness of the restructuring.

Some simple heuristics are applied by Guru to determine which objects should have their behavior preserved. These heuristics partly rely on the way that Self systems are structured. Objects which do not have children, i.e. leaves of the inheritance hierarchy, represent concrete objects, or 'instances' as they would be called in a class-centric programming language. At least these objects need to be preserved. This is the case in the example used in section 1.2 (see Figures 1 and 2).

Guru uses other heuristics to identify more objects which need to be preserved. In practice these heuristics have been adequate. However, a more accurate way to determine which objects to preserve would be to use sophisticated type inferencing[Agesen 95] to identify which objects are sent any messages.

2.1.1 The 'traits' object problem

There is a common situation which causes more preserved objects to be identified than desirable, unnecessarily limiting the amount of restructuring that can take place. Many parent objects are referred to by non-parent slots as a convenient way to refer to them while developing a system. There is an object in the standard system, called the 'traits' object, which exists for this very purpose. The name 'traits' is used to mean the shared behavior of objects, that is, their shared parent object. For example, 'traits set' is used to refer to the parent object of all 'set' objects. Such non-parent references do not affect non-reflective programs. If such parent objects are to be restructured by Guru, their non-parent references must be removed otherwise they will not be restructured. After restructuring, new non-parent references can be reintroduced by the programmer if this is required.

In general, it is not possible for the system to automatically decide which non-parent references it should remove, other than removing all of them, which may not be desirable. Furthermore, it is not possible for the system to automatically rebuild new non-parent references after restructuring, not least because the system cannot invent meaningful names for the newly created objects. The non-parent references to parent objects are considered bad style by some[Ungar 94], as not only are they reflective but they also indicate a class-centric way of thinking, contrary to the philosophy of Self.

2.2 Restructuring part of a system

Guru is designed so that it can be used on part of a system, rather than having to restructure an entire system. Restructuring only part of a system avoids changing things which one does not want to have changed. A reason for wanting only part of a system to change is to avoid the benefits of a restructured system being outweighed by the effort to learn about the restructured version of the system. Furthermore, restructuring only parts of a system is faster than restructuring the entire system, which may take too long to be feasible.

The user manually specifies what should be included in the restructuring. However, the system should check whether all the children of these objects are included, as their parents might not exist after restructuring. An example is shown in Figure 3. If there is a reason why a child should not be included, for example, if there are too many children of an object to include all of them, the parent object must be preserved so that its children still have a parent. The result of restructuring the hierarchy in Figure 3 is shown in Figure 4.

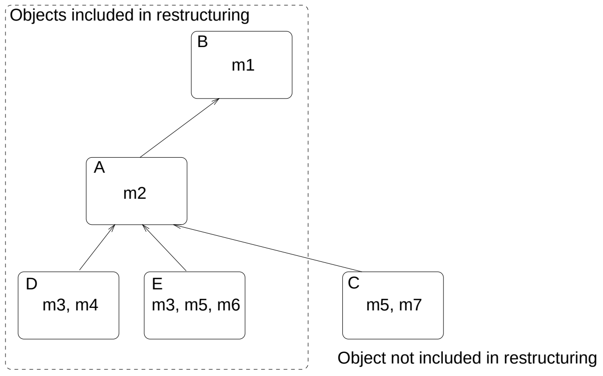
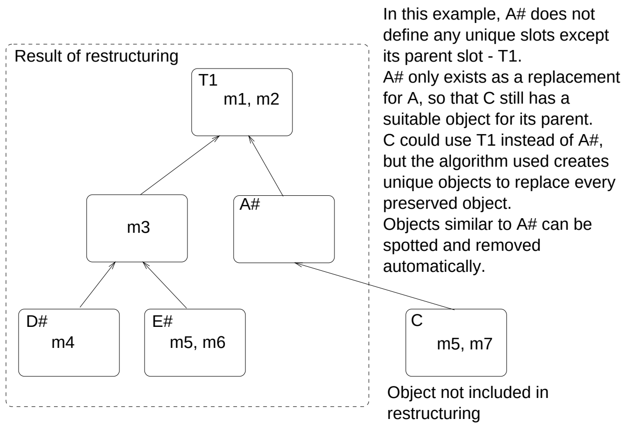

Objects included in restructuring
B
m1
A
m2
D
m3, m4
E
m3, m5, m6
C
m5, m7
Object not included in restructuring

Figure 3: Example where child is not included in restructuring



In this example, A# does not define any unique slots except its parent slot - T1.
A# only exists as a replacement for A, so that C still has a suitable object for its parent.
C could use T1 instead of A#, but the algorithm used creates unique objects to replace every preserved object.
Objects similar to A# can be spotted and removed automatically.

Figure 4: Result when A is included as a preserved object

Restructuring only part of a system can result in ambiguous message sends being introduced, as shown in Figure 5. Any ambiguities introduced can be spotted and removed by automatically adding disambiguating methods.

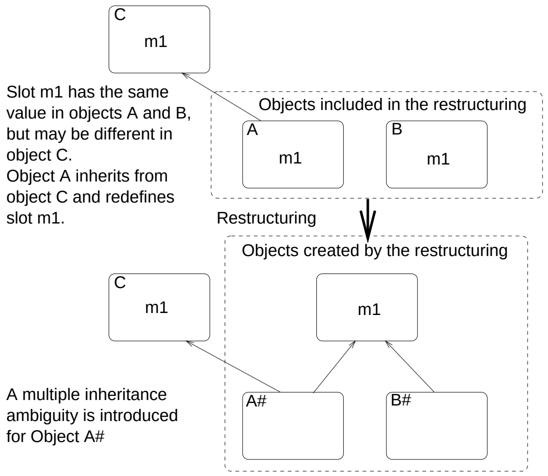

C
m1
Slot m1 has the same value in objects A and B, but may be different in object C.
Object A inherits from object C and redefines slot m1.
Objects included in the restructuring
A
m1
B
m1
Restructuring
Objects created by the restructuring
C
m1
m1
A multiple inheritance ambiguity is introduced for Object A#
A#
B#

Figure 5: Ambiguities introduced by restructuring.

2.3 Guru's approach to restructuring

Having decided which objects to include in the restructuring, and which objects will be preserved, Guru creates a restructured inheritance hierarchy. The approach used by Guru is firstly to 'flatten' the preserved objects, by copying into them all their inherited slots. For example, for the objects shown in Figure 1, the 'flattened' preserved objects are those shown in Figure 6.

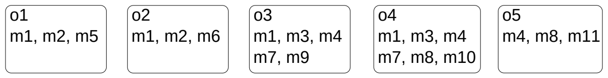


Figure 6: Example problem flattened preserved objects

An algorithm which solves the problem of creating an inheritance hierarchy for flattened objects is then applied. The algorithm ensures that equivalent slots in different objects are shared, by grouping those slots into objects, and making the objects which include those slots inherit from these grouping objects. Details of the algorithm are not included in this paper.

The way that equivalence of slots is decided has very significant implications. In Guru's restructuring, slots are only equivalent if they have the same name and the same value. The values of two methods are the same if their parsed versions are the same. This is conservative as it misses cases where methods have the same effect but are written slightly differently, which is known to be an undecidable problem. Assignable slots are only equivalent if they are identical. That is, two different assignable slots can have the same name and the same current value, but are not the same slot.

Flattening objects has some benefits but also some problems. The most significant feature of 'flattening' is that all overridden slots are thrown away. This is beneficial, because the resulting inheritance hierarchy after restructing will have no overridden slots, eliminating slots which are unused because they are always overridden and simplifying the resulting system. Having many slots overridden, or slots overridden many times, is often an indication of poor inheritance hierarchy design[Johnson 88].

Methods with resends are treated in a simple but effective way. A resend causes method lookup to start in a parent of the object where the method containing the resend is defined. Resends in methods would be meaningless if nothing was done about them, as in the 'flattened' object there is no parent for the resend to refer to. A simple solution to the problem of resends is to get rid of them. Resends to non assignable parents are statically determinable, so a resend in a method is replaced by a message send for a uniquely named method, which is created as a copy of the method which will be called and included as one of the slots of the flattened object. The renamed methods created in this way are shared by IHR amongst all the objects which call it by a resend in one of their methods. Figure 7 shows an example of how resends are flattened.

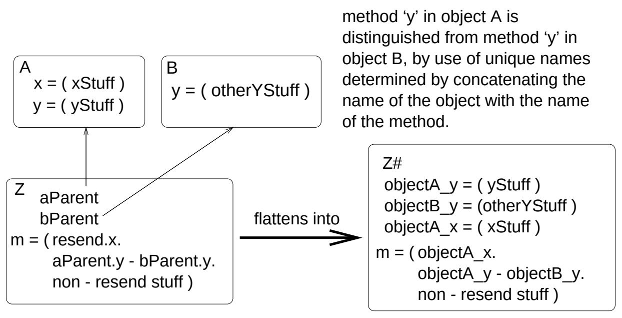


Figure 7: How resends are 'flattened'

Resends to assignable parents, which are not statically determinable, are also handled by Guru, but this is beyond the scope of this paper.

2.4 How to describe the results of a restructuring

The restructured system may be significantly differently in structure to the original system. Therefore, to make the restructured system easier to understand, a description is required of the relationship of the original system to the restructured system, and vice-versa.

Preserved objects can be mapped directly to restructured objects with the same behavior but modified inheritance hierarchies. More sophisticated mappings can be constructed, for example, relating a slot in the original system to a slot in the restructured system. The slots which were duplicated in the original hierarchy can also be found, along with their replacements. To relate a non-preserved object from the original system to the objects in the restructured system, all the slots of the original object can be related individually to the slots which replace them, but they are likely to be split over several objects. In the restructuring shown in the section 1, the system could report, for example:

- Object o1 is replaced by o1#
- Slot m1 in t1 and m1 in t2 are both replaced by m1 in t5
- Slot m2 in t1 is replaced by m2 in t4
- Object t2 is replaced by m1 in t5 and m3 in t6
- Object t3 is replaced by t7

3 Complementary techniques

The IHR described is reasonably fast to perform, but nevertheless it would not be desirable to do it every time that the system is changed. Programmers get used to the inheritance hierarchy, and so changes should not be made too often. Therefore other techniques can be employed, which are faster and incremental, but do not necessarily produce optimal hierarchies or handle all situations, such as overridden methods or resends[Hoeck 93]. For example, when a slot is added to an object, a check can be made to see if it is defined elsewhere in the system. If an equivalent slot to the one being added is inherited by a 'sibling' object then it may be possible to move the slot into a common parent, thus removing the duplication of the slot.

4 Results

Guru has been applied to some Self objects to evaluate the restructuring approach taken. In order to make a fair evaluation, two groups of objects were restructured, one group which was representative of well written Self code, and another which was written by a novice Self programmer.

4.1 Collection objects

The group of objects representing well written Self code comprised of 17 objects from the collection objects inheritance hierarchy. They are an almost self-contained inheritance hierarchy of 'vector-like' objects; the names of the objects and the number of slots they define are shown in Figure 8.

Note that **traits canonicalString**, **traits mutableString**, **traits byteVector** and **traits vector** have many children in the standard image as well as those shown in the figure. It was assumed that **traits sequence** and **traits sortedSequence** would also normally have many children other than those shown. These objects which have children, or are assumed to have children, not included in the restructuring, were included as preserved objects.

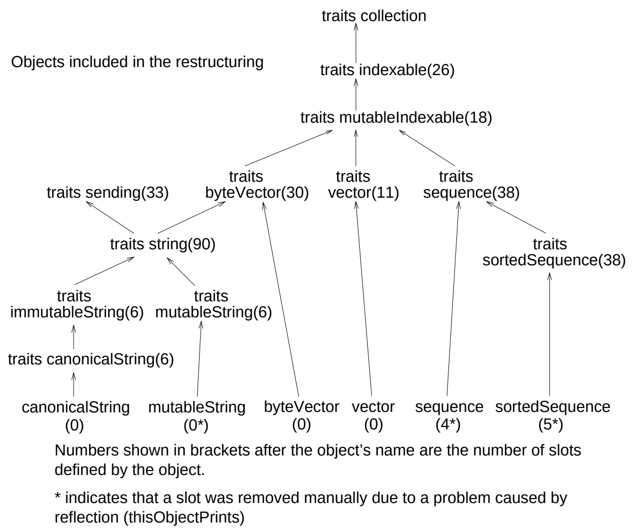

traits collection
Objects included in the restructuring
traits indexable(26)
traits mutableIndexable(18)
traits sending(33)
traits byteVector(30)
traits vector(11)
traits sequence(38)
traits string(90)
traits sortedSequence(38)
traits immutableString(6)
traits mutableString(6)
traits canonicalString(6)
canonicalString (0)
mutableString (0*)
byteVector (0)
vector (0)
sequence (4*)
sortedSequence (5*)
Numbers shown in brackets after the object's name are the number of slots defined by the object.
* indicates that a slot was removed manually due to a problem caused by reflection (thisObjectPrints)

Figure 8: The collection objects inheritance hierarchy

Guru took 45 seconds on a Sun 5 to restructure these objects into the inheritance hierarchy shown in Figure 9.

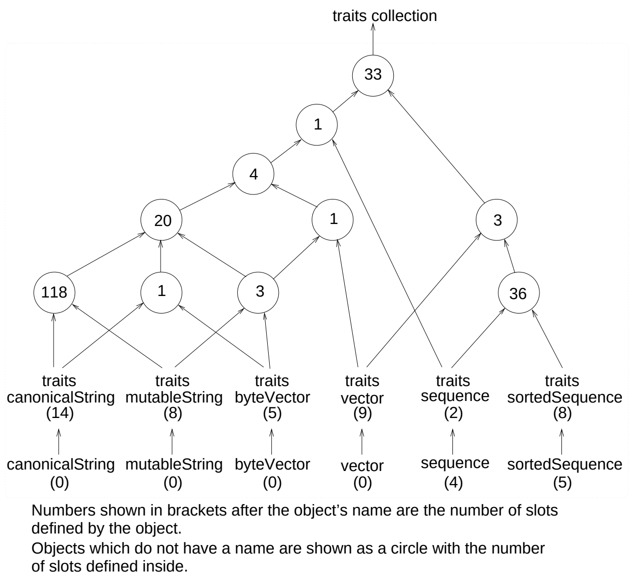

traits collection
33
1
4
20
1
3
118
1
3
36
traits canonicalString (14)
traits mutableString (8)
traits byteVector (5)
traits vector (9)
traits sequence (2)
traits sortedSequence (8)
canonicalString (0)
mutableString (0)
byteVector (0)
vector (0)
sequence (4)
sortedSequence (5)
Numbers shown in brackets after the object's name are the number of slots defined by the object.
Objects which do not have a name are shown as a circle with the number of slots defined inside.

Figure 9: Initial result of restructuring collection objects

The initial result looks over-complicated, and does not seem to be an improvement. However, further investigation revealed that some simple alterations could improve the inheritance hierarchy. It was found that there were five slots in the original objects which redefined slots unnecessarily, that is they redefined slots without making any change to the slot definition. These redefinitions were not removed by Guru because they redefined methods inherited from objects not included in the restructuring. These slots were removed manually. Code has now been added to find such slots automatically.

The original objects also included two comment slots which were essentially reflective, as they literally comment on the structure of the inheritance hierarchy. These slots were also removed, as they do not affect the behavior of the objects.

Having removed these slots, the objects were restructured again. After restructuring, one method was manually moved up the inheritance hierarchy. This method was the only slot of an object which was used for sharing it by multiple inheritance between two objects. By moving the method up the inheritance hierarchy, the two objects still inherited it, and a third object redefined it. This was a case where having a slot redefined led to a neater inheritance hierarchy than having no slots redefined. Ambiguous message sends were introduced by the restructuring, as explained in section 2.2, and were easily remedied by hand, but at the expense of adding 8 trivial disambiguating methods. If traits collection did not define slots which have to be redefined by objects included in the restructuring, then 8 slots would have been removed by the restructuring. The resulting inheritance hierarchy is shown in Figure 10.

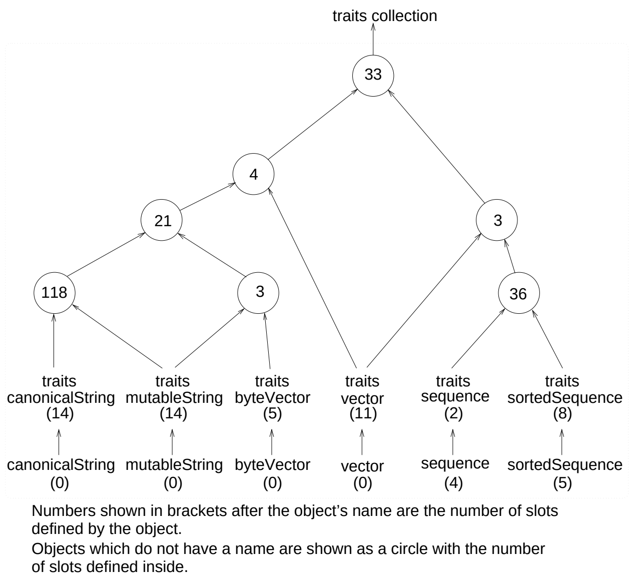

traits collection
33
4
21
3
118
3
36
traits canonicalString (14)
traits mutableString (14)
traits byteVector (5)
traits vector (11)
traits sequence (2)
traits sortedSequence (8)
canonicalString (0)
mutableString (0)
byteVector (0)
vector (0)
sequence (4)
sortedSequence (5)
Numbers shown in brackets after the object's name are the number of slots defined by the object.
Objects which do not have a name are shown as a circle with the number of slots defined inside.

Figure 10: Final result of restructuring collection objects

Relevant statistics comparing the original inheritance hierarchy to the final restructured inheritance hierarchy are given in the following table.

Number in:	Original hierarchy	Restructured hierarchy
Objects	17	19
Non-parent slots	282	282
Slots redefined within hierarchy	35	1

The restructured objects were then used to replace the original objects, and, although not fully tested, appeared to behave in the same way as the original objects.

4.2 A simple game

A group of objects in a simple game, written by a novice Self programmer, were restructured using Guru. The behavior of the game was unaffected by replacing the original objects with the restructured objects. For this program, the inheritance hierarchy was just as complicated after restructuring as before. An interesting feature of the restructured inheritance hierarchy produced was that it revealed certain faults in the original design. In particular, over-use of inheritance had led to inappropriate slots being inherited by some objects. It was apparent that the design needed a major change by an experienced Self programmer. Much of the behavior of the objects would have to change to improve the design, which is beyond the capabilities of Guru. Statistics concerning the restructuring are presented in the table below.

Number in:	Original hierarchy	Restructured hierarchy
Objects	32	32
Non-parent slots	110	103
Slots redefined within hierarchy	14	0

5 Limitations and Problems

As previously mentioned, Guru cannot restructure reflective code. This is because reflective code relies on the structure of the system, rather than only on its behavior. There appears to be no simple solution to this, except to encourage the style of avoiding reflection in application code.

The initial results using Guru appear to support the claim that restructured systems can be easier to understand. However, restructured systems and objects may bear so little resemblance either to the original system or to any concepts that are understood by the programmer or designer that the restructured system will be very difficult to understand. The inheritance hierarchy is abstracted from objects that exist in the system at the time that the restructuring is performed. This means that abstractions are made from what actually exists, rather than from the thoughts of the designer or programmer. This can be advantageous because the system as it exists contains the actual implicit design. The disadvantage is that it is only a snapshot of the design, and so may not reflect the 'long term design' but rather an unrepresentative version of the design due to the particular circumstances of that stage of the development of the system. Another disadvantage is that abstractions reverse engineered from the system may not match recognisable real world abstractions.

An optimal solution to the hierarchy restructuring problem has been shown to be NP-Hard[Lieberherr 91]. The algorithm used by Guru has performance which is adequate for problems of approximately 100 objects or less. This is not thought to be a problem, as the most satisfactory way to use Guru would seem to be for objects which together make a 'module' of some sort, that is a small collection of objects related in some way (but not necessarily by inheritance).

6 Future Work

Guru is not yet a complete system; the central IHR has been implemented but many of the support tools have not yet been implemented; for example, describing the results of restructuring, as mentioned in section 2.4.

There are improvements which can be made to the results produced by Guru. For example, resends are currently removed by the 'flattening' process described in section 2.3, leading to methods with unnatural names being introduced. Some of these methods could be renamed by reintroducing resends where appropriate.

The results in section 4.1 indicate that neater inheritance hierarchies can be achieved in some cases when slots are overridden. As Guru creates inheritance hierarchies with no overridden slots, it sometimes misses the possibility of neater inheritance hierarchies. Creating neater inheritance hierarchies by having some slots overridden is an area that requires further investigation.

Furthermore, the results in section 4.1 show that ambiguities introduced by IHR can reduce the effectiveness of Guru at minimising the total number of slots, by requiring slots to remedy the ambiguities. Including more objects in the restructuring reduces this problem, but one of the aims of Guru is to be able to restructure only part of a system. How to avoid ambiguities being introduced, and hence avoiding the addition of trivial disambiguating methods, is an area of future research.

Inappropriately inherited slots could be removed by the programmer redefining them as 'shouldNotImplement', in the style of Smalltalk. Then, before creating the restructured inheritance hierarchy, Guru could remove all 'shouldNotImplement' slots from the 'flattened objects' it creates. This would then result in an inheritance hierarchy in which there are no inappropriately inherited slots and no 'shouldNotImplement' slots.

The inheritance hierarchy is only one aspect of an object oriented program, and hence only one aspect which will require restructuring. Therefore, IHR is not proposed as a full solution to all restructuring requirements. In particular, other restructurings should be applied both before and after IHR to achieve the best results. Casais[Casais 90] summaries other approaches which assist in improving a class hierarchy, which can be applied in conjunction with the approach described in this paper. For example, replacing case analysis with polymorphism by introducing new subclasses[Opdyke 92] will have an effect on the results of IHR, but is not part of the restructuring described in this paper. Similarly, the result of IHR can also be improved afterwards. In particular, some of the new parent objects created may be too large and lack cohesion, hence require restructuring. The factoring of common parts of methods, rather than complete methods, can be integrated into Guru to further improve the results. Furthermore, the application of other restructuring techniques, for example those developed for class-centric and non object oriented systems, can be investigated for object-centric systems. Work is continuing in these areas for the restructuring of Self programs.

An important area not yet addressed is that of the user interface for restructuring tools for Self. The latest Self user interfaces emphasise direct manipulation of objects, rather than the tool-oriented user interface style of Smalltalk. Research into the integration of restructuring tools into such a style of interface is required.

7 Conclusions

Inheritance hierarchies evolve, and hence need continual, occasional restructuring to keep them well designed. Many developers are reluctant to restructure inheritance hierarchies manually. This is not surprising, as restructuring inheritance hierarchies is difficult.

Guru tackles the problem of automatically restructuring an inheritance hierarchy, while preserving the behavior of programs. Firstly, copies of the objects to be restructured are created, in which the inheritance hierarchy is thrown away, removing overridden slots and resends. Then, a replacement inheritance hierarchy is built which ensures no duplication of slots.

Initial results have shown that the inheritance hierarchies produced by Guru are easy to understand when restructuring well written code. For poorly written code, inheritance hierarchies created by Guru can assist the programmer in identifying the faults of the original design.

Work is continuing on improving Guru.

Acknowledgements

The author would like to thank the EPSRC for funding this work, George Paliouras and Richard Banach for their useful discussions about the algorithm used for creating optimal inheritance hierarchies, and Jon Taylor, Mario Wolczko and Tim Clement for their comments on this paper. Many thanks also to Trevor Hopkins for supervising the first year of my PhD studies, and the Self group for interesting discussions about all aspects of the Self system.

References

[Agesen 95] Ole Agesen
The Cartesian Product Algorithm
Proceedings of ECOOP (1995)

[Casais 90] Eduardo Casais
Managing Class Evolution in Object-Oriented Systems
Object management, Centre Universitaire d'Informatique, Geneve (1990)

[Hoeck 93] Bernd H. Hoeck
A Framework for Semi-Automatic Reorganisation of Object-Oriented Design and Code
University of Manchester (1993)

[Johnson 88] Ralph E. Johnson, Brian Foote
Designing Reusable Classes
Journal of Object-Oriented Programming, pp22-35, June-July (1988)

[Lieberherr 91] Karl J Lieberherr, Paul Bergstein, Ignacio Silva-Lepe
From objects to classes: algorithms for optimal object-oriented design
Software Engineering Journal Vol 6 (1991)

[Opdyke 92] William F. Opdyke
Refactoring Object-Oriented Frameworks
University of Illinois at Urbana-Champaign (1992)

[Pun 89] Winnie W. Y. Pun, Russel L. Winder
Automating Class Hierarchy Graph Construction
Technical Report, University College London (1989)

[Ungar 87] David Ungar, Randall B. Smith
Self: the power of simplicity
Proceedings of OOPSLA (1987)

[Ungar 94] David Ungar
self-interest@self.stanford.edu mail group (1994)