AUTOMATIC RESTRUCTURING OF OBJECT-ORIENTED PROGRAMS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

October 1996

By
Ivan Ronald Moore
Department of Computer Science

Contents

Abstract

This thesis shows that automatic restructuring improves object-oriented programs. Most programs are imperfectly designed, and their imperfections tend to increase with maintenance and evolution. Even object-oriented programs suffer from these faults, and so are more expensive to maintain, harder to understand and larger than necessary. This thesis explores automatic restructuring of object-oriented programs in the language Self, and describes the implementation of a restructuring tool called Guru.

Many forms of restructuring are possible. This thesis describes an algorithm for creating inheritance hierarchies from object definitions. Solutions are given to problems which arise when applying such an algorithm to restructuring actual programs. Inheritance hierarchies can be restructured by Guru into equivalent ones in which there are no duplicated methods.

Furthermore, Guru can refactor shared expressions from methods at the same time as restructuring a hierarchy, resulting in hierarchies in which no methods, and none of the expressions that can be factored out, are duplicated.

Results from applying Guru to real Self code are described. Restructured programs are smaller, more consistent and have better code reuse than the original programs. Inheritance hierarchies resulting from the application of Guru have exactly the structures that should be expected of well designed hierarchies. Guru is shown to preserve the behaviour of programs, by replacing original objects with their equivalent restructured objects.

Previous work on restructuring object-oriented programs has considered only

class-centric languages. This thesis explores the restructuring of programs in Self, an object-centric language.

Complementary tools which provide program analyses and restructurings, useful both in combination with and separately from Guru, are described, and the way that a user can interact with such tools is considered.

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright Notice

Acknowledgements

I would like to thank the EPSRC for funding this work. I am very grateful to my ex-supervisors: to Trevor Hopkins for enthusing me about objects and program restructuring, to Mario Wolczko for introducing me to Self and for his continued assistance despite being thousands of miles away, to Jon Taylor for suggesting that I 'wrote some papers and did some science', and to Tim Clement for mathematically confirming the properties of the central algorithm, and my current supervisor Chris Kirkham for his support in writing this thesis.

I am also indebted to the Self group, in particular, Ole Agesen, Randy Smith, Dave Ungar and Mario Wolczko, for advice and informative discussions about the Self system, and for making the world safe for objects. Many thanks also to Richard Banach, Carole Goble, Sean Bechhofer, George Paliouras and Eduardo Casais for their comments on the central algorithm used, and to James Noble, Marianne Huchard and Daniel Bardou for their comments on the restructuring tool produced for this work. I also thank my family and friends for their understanding and support during my studies at Manchester University; in particular my fiancée Linda, my mother Jean, my brother Dug, sisters Sue and June, in-laws George, John and Irene, and all my friends, including Phran, Mashhuda, Dan, Bernd, Zoltan, John, Dimitri, Derek, Stuart, Alan, Julie, Ann, Richard ... and many more.

The Author

The author obtained a Bachelor of Science degree in Computer Science from the University College of Wales, Aberystwyth, in 1989, and a Master of Science degree in System Design from the University of Manchester in 1993.

Chapter 1

Introduction

The software development process is iterative, whether it is planned that way or not. The optimal design of a system depends on its current requirements. As requirements inevitably change, a system will evolve, requiring restructuring if it is to remain well designed. It is impossible to predict future requirements, so attempts to design in a way that allows for change cannot be entirely successful. Even if a system is initially well designed, subsequent extensions and modifications may turn out to be best handled by radical restructuring, and this seldom happens. Many programmers are reluctant to restructure a system manually, 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 a sufficiently detailed level to restructure it. Manual restructuring is also error prone and, if a system works, however badly structured it is, the temptation is to leave it alone. Even in the basic libraries of object-oriented languages there is often scope for improvement [Cook92, Meyer90]. One of the distinguishing features of the most productive programmers is that they spend considerable effort in perfecting and modifying code [Brooks87]. By automating restructuring, it is hoped that more frequent restructuring is encouraged and made more feasible.

The iterative and evolutionary nature of developing software has motivated the development of object-centric programming languages. In these languages, objects are created and modified directly, without the need to define classes. This allows objects to evolve by direct modification, rather than having to commit objects to being of a particular class. Class-centric programming languages require classes to be defined before instances of those classes can be created. This reflects a Platonic philosophy of objects and abstractions [Plato], in which the abstractions (classes) must exist in order to understand their instances. The object-centric philosophy is to create objects first, and then to create abstractions based on the objects that exist, rather than to try to define the abstractions first.

Automatic structuring and restructuring of programs based on the objects in a system is motivated by the evolutionary nature of software development. It recognises one of the most fundamental philosophical driving forces behind object-centric programming languages: the need to base abstractions on objects rather than vice versa.

Section 2.4 gives an example of how programs deteriorate, and how restructuring can improve them. It relies on some knowledge of object-oriented concepts, and so is left until the end of the chapter which introduces these concepts (Chapter 2).

The restructuring system developed and described in this thesis is called Guru, because it assists in Self improvement.

1.1 Structure of the thesis

Object-oriented programming has become increasingly popular, but not everyone agrees on the meaning of 'object-oriented' or on the associated terminology. Chapter 2 introduces the concepts and terminology of object-oriented programming used in this thesis. Object-centric programming is a new approach to object-oriented programming. The differences from conventional (class-centric) object-oriented programming are explained in Chapter 2, along with an introduction to Self [Ungar87], the object-centric programming language used for this work.

Inheritance is introduced in Chapter 2 as one of the defining characteristics of object-oriented programming. It allows programs to capture the shared characteristics of objects, at different levels of abstraction. Designing inheritance hierarchies is hard, precisely because it requires the identification of appropriate abstractions. Texts on object-oriented design such as [Meyer88] provide guidelines, but it remains an art rather than a science. An alternative approach to constructing a hierarchy by hand is to infer one from the features of the objects that a program creates, or will need to create. Chapter 3 describes the *extended inheritance hierarchy inference* algorithm, which infers an inheritance hierarchy that satisfies well justified criteria, such as eliminating duplication of features, for any set of objects.

Having created an inheritance hierarchy, many developers think of it as static; they may add new classes, but are reluctant to restructure it. This is not surprising, as restructuring inheritance hierarchies is difficult and error prone. Automatic restructuring requires no programmer effort and does not risk the introduction of errors. Chapter 4 shows how the algorithm described in Chapter 3 has been used in the construction of an automatic restructuring tool called Guru. Any group of Self objects can be restructured by Guru into an inheritance hierarchy which meets the criteria defined in Chapter 3 (such as no duplication of methods) while preserving the behaviour of programs.

Restructuring hierarchies in order to factor shared methods into traits objects (classes) is only one form of factoring. Another is factoring common code into methods, which allows systems to be more compact and improves consistency. Manually designing and restructuring inheritance hierarchies and methods which maximise factoring is inherently difficult. Chapter 5 describes how Guru has been extended, from automatically restructuring an inheritance hierarchy to automatically refactoring shared expressions from methods at the same time. Method refactoring is integrated into the restructuring of inheritance hierarchies described in Chapter 4, to enable the maximum amount of factoring of methods and expressions. In the resulting inheritance hierarchies, none of the methods and none of the expressions that can be factored out are duplicated.

Chapter 6 describes experiments performed in order to assess the restructuring, with and without refactoring of methods, performed by Guru. The results indicate that the structure of inheritance hierarchies produced by Guru is exactly that expected of well designed inheritance hierarchies. The amount of code is reduced, by removing redundancies, and the consistency and amount of code reuse increased.

The restructurings, with and without refactoring of methods, described in Chapters 4 and 5 are useful by themselves, as shown by the results described in Chapter 6, but are not the only possible useful tools for restructuring or analysis of a system. Chapter 7 describes other analysis and restructuring tools which are useful either individually, or in conjunction with other restructurings.

Finally, conclusions derived from the work described in this thesis are presented in Chapter 8, along with a critique assessing the value of this research and discussing areas of future work which could address some of the limitations, and build on the achievements, of this research.

Chapter 2

Background

Object-oriented programming is increasing in popularity, but not everyone agrees on the meaning of 'object-oriented' or on the associated terminology. Section 2.1 introduces the concepts and terminology of object-oriented programming used in this thesis.

Object-centric programming is a new approach to object-oriented programming and the differences from conventional object-oriented (class-centric) programming are explained in Section 2.2.

Self [Ungar87], the object-centric programming language used for this work, is introduced in Section 2.3.

Section 2.4 shows an example of how programs deteriorate, and how restructuring can improve them.

2.1 Object-oriented programming

This section introduces the concepts and terminology of object-oriented programming necessary to understand the work described in this thesis. More complete introductions, including the principles behind object-oriented programming and the reasons for its increasing popularity, can be found in [Budd91, Meyer88].

This section introduces class-centric object-oriented programming, and the following section introduces object-centric object-oriented programming. Unless otherwise stated, 'object-oriented' means 'class-centric object-oriented'.

The following concepts and mechanisms are considered as defining object-oriented programming:

- Objects.
- Inheritance and classes.
- Message sending polymorphism.

2.1.1 Objects

An object-oriented system is composed of objects, which have identity and encapsulate state and behaviour. 'Identity' means that an object has an identity distinct from other objects, independent of its value (or state). For example, two people can have the same name, but are distinct individuals with their own identities. Conversely, one person can be known by different names (using aliases), but the person's identity does not depend upon which name is used. 'State' means that an object has its own unique state, which can change and is unique to it. For example, a person has an age which may be different to other people's ages. A person's age changes on each birthday, which may be on a different day to other people's birthdays. Each object has a particular behaviour; for example, the noise made by a dog is different to the noise made by a sheep.

2.1.2 Inheritance and classes

Objects are created as belonging to a 'class' and are each called an 'instance' of their class. Classes are used to define the behaviour shared by all of their instances. Furthermore, behaviour can be shared between classes, using inheritance. A class which inherits from another class defines its behaviour not only itself, but also shares all of the behaviour defined by classes from which it inherits.

Many statically typed object-oriented languages use inheritance to define subtyping relationships as well as for sharing implementation. The issues associated with using the same inheritance hierarchy for both subtyping and sharing implementation are discussed in [Cook90].

2.1.3 Message sending polymorphism

Behaviour (and in some languages, state) is invoked (accessed) using message sends. A message is sent to an object to make it do something. For example, an object representing a set can be sent a message to add another object to the set (such as '`aSet add: anObject`'). Message sending polymorphism means that different objects can be sent the same message, but respond to it in different ways. For example, the message to add an object to a set is the same as the message to add an object to a list, but the response to the message is different depending upon whether it is a set or a list receiving the message. A set responds to the message by checking whether it already contains the object, in which case the set will not add the object. A list responds to the message by adding the object to the end of itself. The code executed in response to a message is typically known as a *method*. The mechanism which determines which method to execute in response to a message send is called *method lookup*, and is implemented using *dynamic binding*. In a class-centric language, the method executed in response to a message send is determined by the class of the object receiving the message, this object being called the *receiver* of the message.

2.2 Object-centric programming

Object-centric programming is distinguished from class-centric programming by the absence of classes. In object-centric programming, inheritance is between objects and there are no classes. New objects are created by copying other objects, rather than as instances of a class.

Conceptually, object-centric programming is as powerful as class-centric programming, in that everything that can be modelled using a class-centric language can be modelled in an object-centric language, as discussed in [Lieberman86, Stein87].

Object-centric languages differ in details such as restrictions concerning which objects can inherit from other objects [Dony92]. The specific features of Self are dealt with in the following section.

2.3 The Self language

Self [Ungar87] 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. The version of Self used for the work reported in this thesis was Sun Microsystems Laboratories Self, Version 4.0.

Self was chosen as the language to investigate object-oriented program restructuring because object manipulation in Self is powerful and easy to use. Furthermore, it is a simple, consistent and powerful language.

Self is similar in many ways to Smalltalk [Goldberg90], but with some differences which will need explaining in order for this thesis to be understood by readers familiar only with Smalltalk or other object-oriented programming languages.

Self does not have static type declarations, which means that the inheritance hierarchy is not used as a static typing hierarchy, i.e. to statically constrain the compatibility of objects, but rather as a means of sharing behaviour and 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. The latter are also called *assignable slots*. Assignable data slots are analogous to instance variables in Smalltalk. However, in Smalltalk, the names of an object's instance variables are defined by its class. Therefore, classes define the *instance variable structure* of their instances. All instances of a class have the same number and names of instance variables, but each object has its own specific instance variables, which are different to other objects' instance variables. Instance variable structure is inherited by classes, just as methods are inherited, from their superclasses. In Self, the names of assignable data slots (instance variables) are defined by *prototype objects*. When a new object is created as a copy of a prototype object, it will have the same number and names of data slots as the prototype object. Therefore, the Self equivalent of instance variable structure is defined by prototype objects. As instance variable structure is defined by copying a prototype object, it is not inherited.

The Self 4.0 programming environment allows the programmer to indicate that an object should include slots, called *copied-down slots*, copied from another object, called its *copy-down parent*. This is particularly useful for specifying that an object copies (at least part of) its instance variable structure from a prototype object. Slots subsequently added to a copy-down parent are also added to its copy-down children. Similarly, modifications to copied-down method slots are propagated to copy-down children. Copied-down slots are indicated by being displayed in a different colour to other slots.

Figure 2.1 shows a Self object as represented in the Self 4.0 user interface. Messages can be sent directly to an object using an *evaluator*. In this example, the result of pressing the 'evaluate' button of the evaluator is the object resulting from sending the message ageNextBirthday to objectA. (In this thesis, Self code fragments are shown in sans serif font.) This causes the method slot ageNextBirthday to execute, which evaluates the expression 'age + 1'. Messages without an explicit receiver are sent to self (hence the name of the language). In this expression, age is a message, which, as it does not have an explicit receiver, is sent to self, namely objectA. Sending age to self causes the (assignable) data slot age to return the object it refers to, which is the object 2. Then, the message '+' is sent to 2 with the argument 1. The object resulting from this message send is 3, which is returned by the method ageNextBirthday. Therefore, the object 3 is returned as the result of sending the message ageNextBirthday to objectA.

Sending the message growOld to objectA causes method growOld to execute, resulting in sending the message age: to self with the result of the expression ageNextBirthday as an argument. The message send age: ageNextBirthday makes the assignable data slot age refer to the object resulting from sending ageNextBirthday to self. Therefore, age will now refer to the object 3.

The arguments of a method are interspersed with the name of the method using ':' to separate each argument. A method has as one ':' for each argument. For example, a method 'aMethod:' with one argument 'oneArgument' is written 'aMethod: oneArgument'. A method 'aMethod:WithTwoArguments:' with two arguments, 'argumentOne' and 'argumentTwo', is written 'aMethod: argumentOne WithTwoArguments: argumentTwo'.

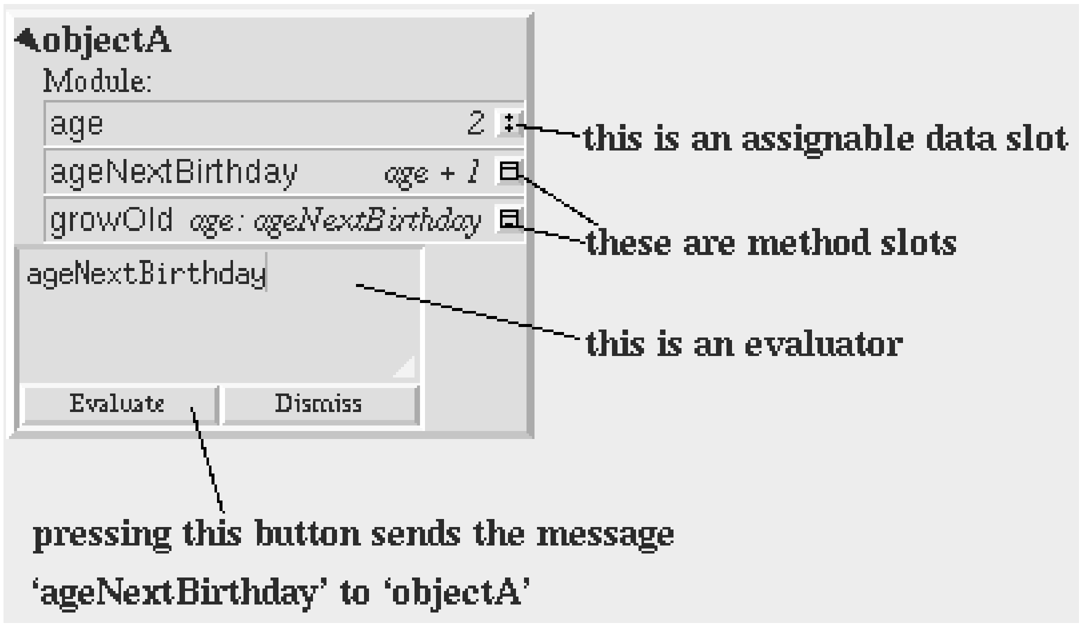

objectA
Module:
age 2
ageNextBirthday age + 1
growOld age: ageNextBirthday
ageNextBirthday
Evaluate
Dismiss
this is an assignable data slot
these are method slots
this is an evaluator
pressing this button sends the message
'ageNextBirthday' to 'objectA'

Figure 2.1: User interface representation of a Self object.

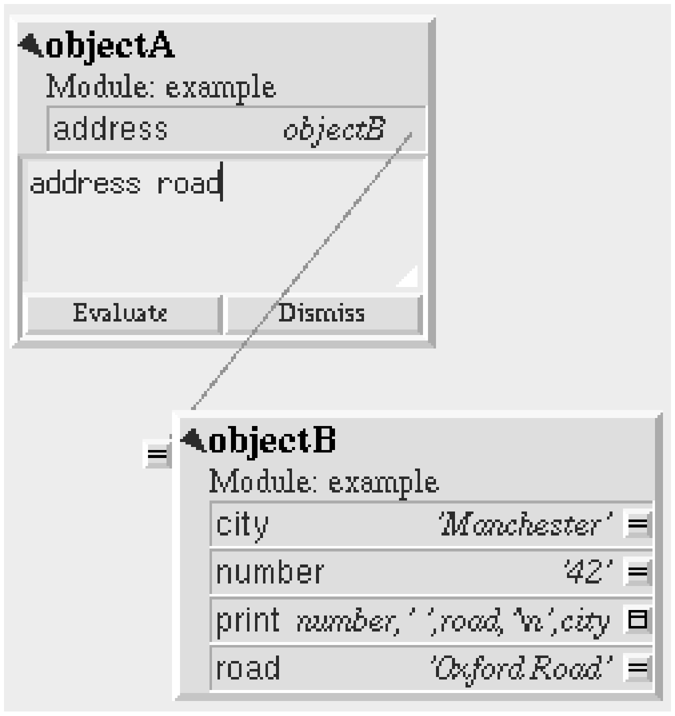
Data slot accesses and assignments are made using message sends which are indistinguishable from message sends that invoke methods. Therefore, methods which send the message age can be written independently of whether age is a data slot or a method slot. For example, an object might have a 'dateOfBirth' data slot, and a method slot called 'age' which calculates the object's age based on its 'dateOfBirth' and the current date. Similarly, a message send which causes an assignment to a data slot is indistinguishable from a message send which invokes a method with one argument.

Any object which has assignable slots is *mutable*; that is, it has *mutable state*. Examples of objects which are *immutable* are integers and floats.

A data slot can refer to any other Self object. Consider the object shown in Figure 2.2. The data slot address refers to an object (objectB), which itself has several data slots. Sending the message 'address road' to objectA causes the message road to be sent to the result of sending the message address to objectA. The object resulting from sending the message address to objectA is objectB. Sending the message road to objectB returns the string object "Oxford Road".

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. Objects inherit from their parent object(s) using parent slots. Figure 2.3 shows an example of an object which has a parent slot. A parent slot is indicated by a '*' after its name. Inheritance is a mechanism for sharing slots amongst objects. Parents can be seen as the shared parts of their children [Chambers91].

If the message birthday is sent to objectA, then, as objectA does not itself define a slot called 'birthday', the slot is looked for in its parent objects (it has only one parent object, labelled 'objectB'). As objectB defines a method slot birthday, this method is executed. Executing the method birthday results in the message party being sent to self (objectA), which consequently results in the method party being executed. Executing method party results in methods eat, drink and beMerry being executed, with self still referring to objectA. The details of these methods will not be discussed. Then, the message growOld is sent to objectA, which as we



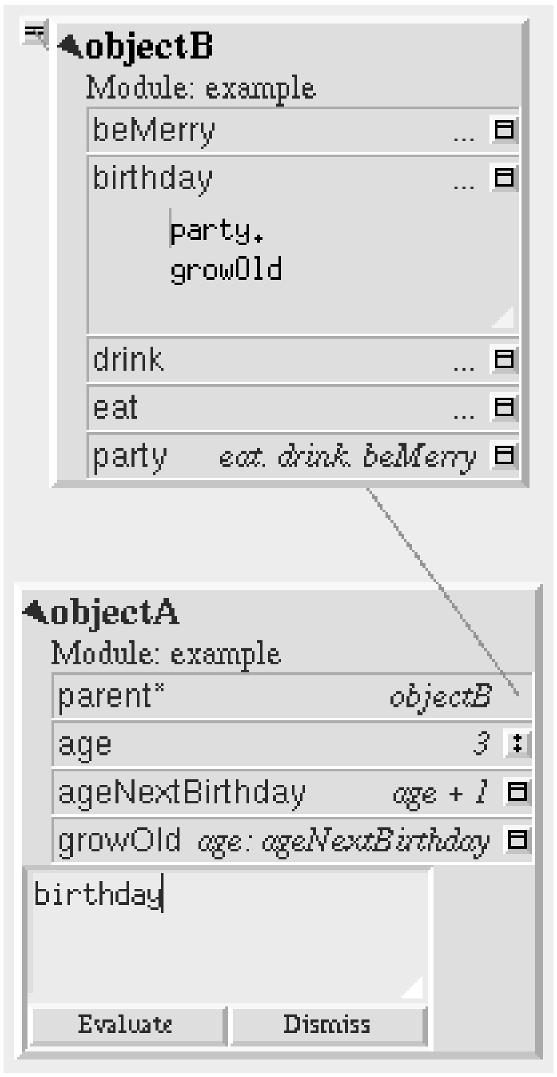
```
objectA
Module: example
address        objectB

address road

objectB
Module: example
city     'Manchester'
number   '42'
print    number,' ',road,'\n',city
road     'Oxford Road'
```

Figure 2.2: A Self object with a data slot.

have seen results in `age` being increased by 1; hence in this example `age` will now refer to the object `4`.

An object does not have to inherit from any other object; it can be totally self-contained, as in Figure 2.1. Consequently, there are multiple roots of the inheritance hierarchy. 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 (called *cyclical inheritance*); an object can inherit from an object which inherits from itself (in fact, an object can even inherit directly from itself).

Parent objects can also be referred to, without being inherited from, by using non-parent (data) slots, as shown in Figure 2.4. This provides a convenient way to refer to parent objects 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 behaviour of objects, that is, their shared parent object. For example, '`traits set`' is used to refer to the parent object of all



objectB
Module: example

```
beMerry ...
birthday ...
    party.
    growOld
drink ...
eat ...
party  eat. drink. beMerry
```

objectA
Module: example

```
parent*          objectB
age              3
ageNextBirthday  age + 1
growOld          age: ageNextBirthday
```

```
birthday
```

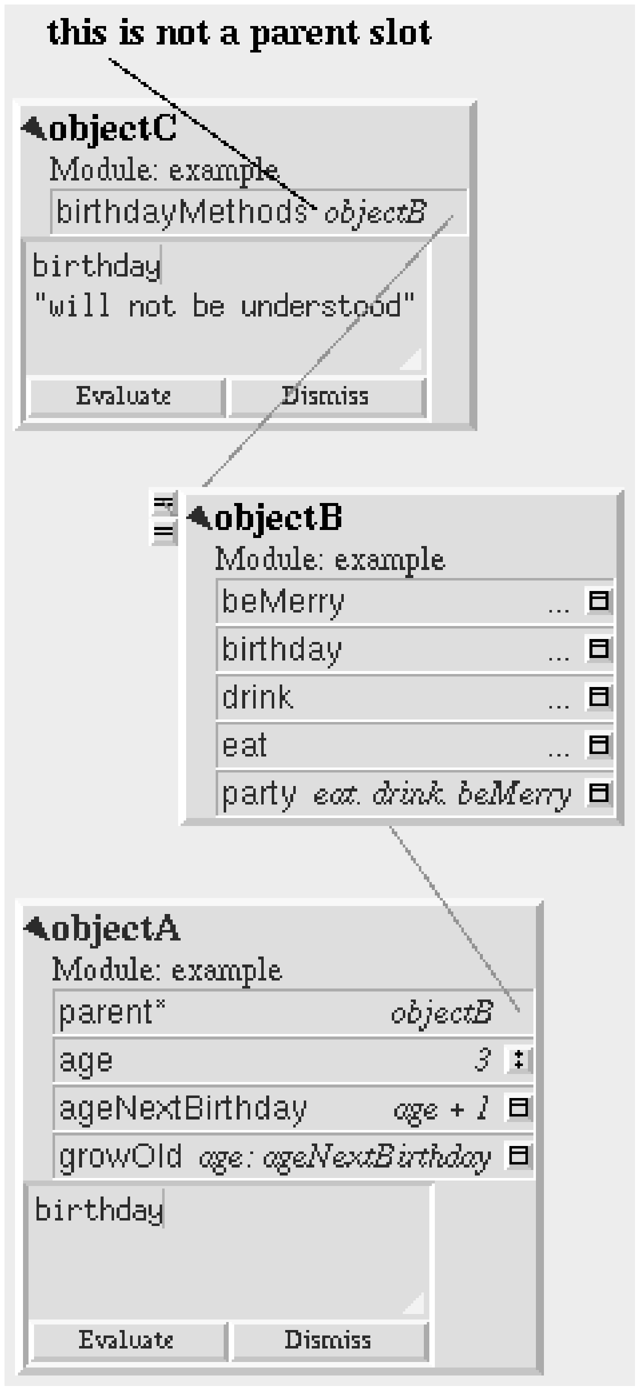
Figure 2.3: Self object with a parent slot.

'set' objects. In this respect, traits objects are analogous to classes in class-centric languages. In the example shown in Figure 2.4, objectA inherits from objectB, but objectC does not. Therefore, objectA will respond to the message 'birthday' as expected, but if sent to objectC, this message will cause an error.

Objects can have more than one parent slot. An object inherits from all of its immediate parent objects equally, that is, there is no preference between parent objects. If a message is sent to an object which it does not implement itself, then an implementation is looked for in all its parent objects, and all their parent objects etc. If only one implementation is found, even if it is found by multiple paths, then this implementation is used in response to the message send. Finding more than one implementation results in a run-time error called an 'ambiguous selector error'. Consider Figure 2.5[1]. Sending the messages 'x', 'm', 'n' and 'p' to object A result in methods 'x', 'm', 'n' and 'p' in objects A, C, D and E respectively being invoked. If the messages 'r', 's', 't' or 'y' are sent to object A, then 'ambiguous selector errors' are reported.

Assignable parent slots are used for modifying, or adding to, the behaviour of an object dynamically. For example, a collection object could be implemented using an assignable parent slot, which inherits from the appropriate parent object depending on whether the collection is empty or not-empty. Figure 2.6 shows this collection object and its two possible parent objects. This implementation allows the methods in notEmpty to be written more simply than if they required tests for isEmpty. The only complication is that remove: now requires a test for whether the collection has become empty. This use of assignable parent slots is called *dynamic inheritance.*

[1]A diagrammatic representation of Self objects has been used in preference to screen images from Self 4.0 in order to make the figures as simple as possible. In all figures, features of the same name are equivalent unless otherwise stated. Arrows represent inheritance, and are shown in the direction from children to parents. Where necessary, the name of an object appears in the top left of the object. Slots are shown inside the object in which they are defined. If the details of the implementation of a slot are not important, only the name is shown.



```
birthday
"will not be understood"
```

```
birthday
```

Figure 2.4: A parent object referred to by both a parent and non-parent slot.

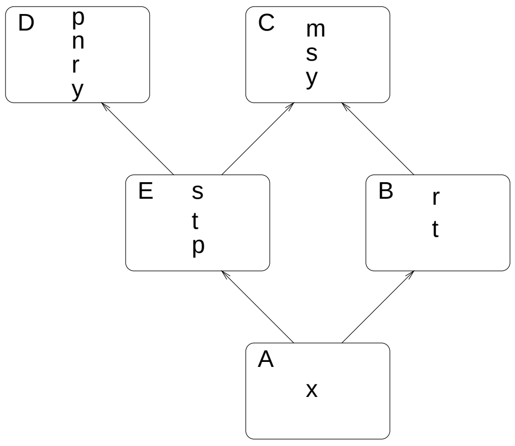


Figure 2.5: An object defining multiple inheritance.

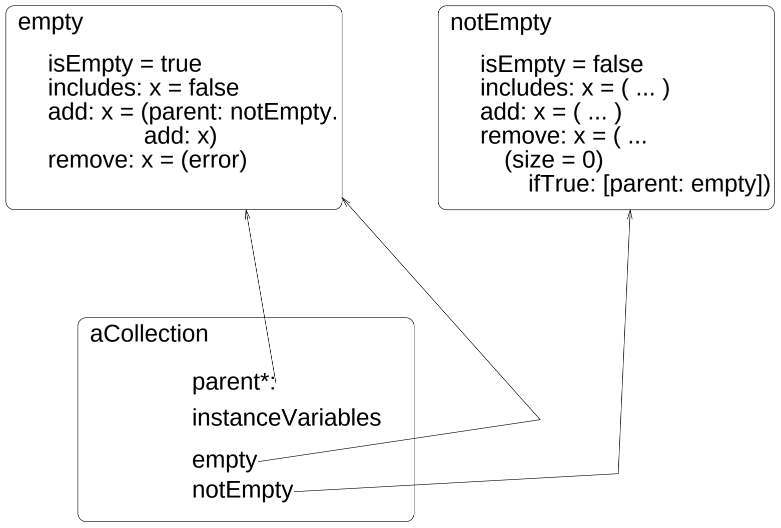


Figure 2.6: An object defining an assignable parent slot.

Another example of the use of assignable parent slots is for dynamically creating objects which are the same as another object, with added behaviour. An object can be created which defines the added behaviour and inherits the original behaviour using an assignable parent slot. Such objects are often called *wrappers* or *decorators*, and the object inherited from using an assignable parent slot is called a *data parent*. Consider a 'pen' object which can draw only black lines, and a 'colour wrapper' which defines only a colour and an assignable parent slot. The colour wrapper can become a 'coloured pen' by assigning the original pen object to its assignable parent slot. The colour wrapper now behaves exactly the same as the pen object, including sharing the same state, but has the additional state of a colour and additional behaviour of drawing coloured lines. Figure 2.7 illustrates this example.

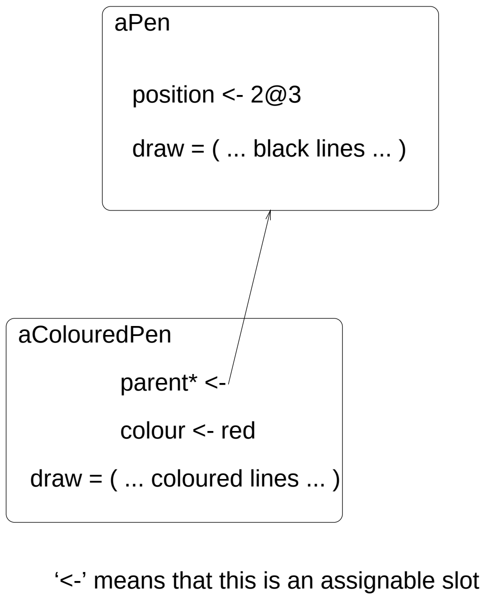

aPen
position <- 2@3
draw = (... black lines ...)
aColouredPen
parent* <-
colour <- red
draw = (... coloured lines ...)
'<-' means that this is an assignable slot

Figure 2.7: Using an assignable parent slot to add behaviour to an object.

2.3.1 Resends

If object A in Figure 2.5 is sent the message p, then the method p in object E will be invoked. However, if the programmer wants method p in object E to reuse method p in object D, this cannot be achieved using a normal message send. A message send p in method p in object E sends the message p to self. If method p in object E results from a message send p to object A, then self is object A. Therefore, if method p in object E sends the message p to self it will invoke itself recursively. *Resends* allow a method to invoke another method which has been overridden. A very similar mechanism is provided in other object-oriented programming languages; for example, 'super' in Smalltalk. The expression 'resend.p' inside method p in object E invokes method p in object D. The method invoked by a resend can be statically determined (if parent slots are not assignable) by looking for an implementor of the resent message (in this example p) in the immediate parents of the object where the method containing the resend is defined. Unlike message sends, resends can be *statically bound*; the search for the method to execute in response to a resend can be done before the method containing the resend is executed, and does not rely on the value of self. The receiver of resends is always self.

Resends can be either directed or undirected. An undirected resend, specified using 'resend.' as in the example above, means that the lookup for the method to be invoked considers equally all parents of the object where the method containing the resend is defined. If more than one implementation is found then this causes a run-time error. For example, consider the objects shown in Figure 2.8. The expression 'resend.m' inside method m in object A would result in a run-time error. A directed resend is specified using the name of a parent slot followed by '.'. The parent slot name specifies in which parent to look for the implementation of the method to be invoked. For example, the expression 'p1.m' is a directed resend, and inside method m in object A would invoke method m in object B.

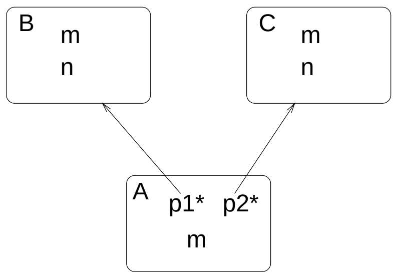


Figure 2.8: Objects including methods containing resends.

One use of directed resends is to resolve ambiguities which would cause 'ambiguous selector errors'. For example, in order to specify which method 'n' to execute in response to a message n sent to object A, a method n can be defined in object A as either 'n = (p1.n)' or 'n = (p2.n)' as required.

2.3.2 Modules

Self 4.0 provides modules for grouping together related slots and objects for 'filing-out' as text files, so that they can be 'filed-in' to other Self images. This provides a flexible mechanism for transporting application source code between Self images [Ungar95].

Those slots which belong to a module are called 'well-known' slots. Other slots which exist in the system are either copies of 'well-known' slots, or have not been put in any module.

2.3.3 The Self 4.0 User Interface

Self 4.0 provides a sophisticated user interface [Smith95] for interacting with objects in a Self image. The user interface supports direct representation and manipulation of objects, allowing the user to add, remove and modify slots of objects, and to send messages directly to objects using evaluators. Figures 2.1, 2.2, 2.3 and 2.4 show how objects are represented in the Self 4.0 user interface.

The user interface is built using *morphs*, which are graphical components which can be assembled together to build complex structures. The user interface supports the direct graphical assembly of morphs.

2.3.4 Mirrors and reflection

Self is a reflective language; that is, programs can be written in Self which analyse and modify objects and methods in the Self image.

In application programs, reflection is discouraged, and separated from normal objects into meta-objects called *mirrors*. The philosophy is that, in application programs, objects should be 'talked to rather than talked about' [Self4.0]. 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 (assignable data slots) or its inheritance relationships. In a classless system it is not important to know an object's parent(s).

The behaviour of a program that uses reflection depends upon things other than how an object responds to messages, such as which slots an object defines rather than inherits. Therefore, the behaviour of such programs can change if objects are modified, even if the behaviour of those objects (in response to messages) remains unchanged.

A mirror on an object is itself an object, which can be sent messages in order to analyse and modify the object it reflects, called its *reflectee*. A mirror is a collection of its reflectee's slots; each slot is itself a Self object, which can be examined and manipulated. Slots can be added to objects, removed from objects, and modified dynamically. The different types of Self slots: parent, data and method slots, are represented by different sorts of slot object.

Consider an object which represents a collection. It will respond to the message `size` by returning the number of elements it contains. If the message `size` is sent to a mirror on this collection object, then the mirror responds with the number of slots in the collection object (which in general will be different to the previous number). To add an object to the collection, the collection object is sent the appropriate message, for example aCollection add: anObject. To add a slot to the object representing the collection, the collection object's mirror is sent the appropriate message, for example aCollectionMirror addSlotFromString: 'newMethodSlot = (some method text)'[2].

An example of an analysis available using mirrors includes being able to check whether the reflectee of a mirror understands a particular message. For example _Mirror understands: 'aMessage'[3]. ('_Mirror' creates a mirror object of the receiver.) Note that this does not result in aMessage being sent to the reflectee object, therefore the method (or error message) which would be invoked if aMessage were sent to the reflectee object is *not* invoked.

2.3.5 Blocks

Self supports anonymous deferred functions called *blocks*, similar to blocks in Smalltalk. All control structures in Self are implemented using blocks. A block is denoted by code enclosed in square brackets ('[' and ']'). For example, [1 + 2] is a block. A block can be passed as a parameter to methods, and sent messages, in the same way as any other object. Blocks understand the message value, which causes the code they contain to be executed. For example, the result of evaluating [1 + 2] value is 3. Blocks can take arguments, for example the block [| :a | a + 3] takes one argument. Blocks understand messages taking corresponding arguments, such as value:. For example, the result of evaluating [| :a | a + 3] value: 2 is 5. A block can be evaluated as often as required; therefore blocks are used as parameters to iteration methods, such as do: defined for collections, which evaluates a block, passing it each element in turn. Blocks can contain *non-local returns*, signified by a '↑', which cause their enclosing method to return at that point. For example, the statement condition ifTrue: [↑ answer] inside a method will cause the method to return answer if condition is true, but

[2]Actually, the addSlotFromString: message is not part of the standard Self image, but is straightforward to define.

[3]In fact, understands: is not a standard Self method, but is simple to define.

otherwise the rest of the method will execute. Unlike Smalltalk, in Self blocks can only be evaluated before the method they appear in has returned. Blocks which violate this are called *non-lifo* blocks, and their evaluation is not supported in Self 4.0. Such blocks can be created simply by returning a block as the result of a method.

2.3.6 Delegation

Self uses *delegation* to implement inheritance. If a message is sent to an object A, and delegated to an object B, the method lookup starts in object B, but self is bound to object A. For example, consider object A delegating a message to object B, using the following message ''aMessage' sendTo: self delegatingTo: B' sent to object A. If B responds to the message aMessage by executing the method aMessage = (size + 1), then the result of the delegated message send will be the result of adding 1 to the result of sending the message size to A. Note that the method executed does not have to be inherited by A, and A does not have to understand the message aMessage. There does not need to be any specific relationship between the objects A and B. An object X inheriting from object Y (using a parent slot) is effectively the same as delegating all messages not understood by object X to object Y.

It is important to understand the difference between delegating a message to another object, and sending a message to another object. In the example above, if the message aMessage is sent to object B by object A then the result is that of adding 1 to the result of sending the message size to B, *not to A*.

The meaning of delegation in Self should not be confused with the meaning often misunderstood by C++ programmers. Many C++ programmers use the terminology that, if the message `aMessage` is sent to object A and delegated to object B, this means that A merely sends B the message `aMessage`. Correctly, this is called *forwarding* a message. The correct definition of delegation in C++ is similar to forwarding, but with the additional feature that a reference to the object originally sent the message is passed to the object that the message is delegated to [Johnson91]. Therefore, if the message `aMessage:` is to be sent to object A and delegated to object B, then B could be sent the message[4] '`B m: self`' (`self` refers to object A), where the method `aMessage:` is defined for B as `aMessage: delegator = (delegator size + 1)`. Messages which would normally be sent to `self` are now sent to the argument `delegator` instead.

2.3.7 Other technicalities

In Self, any string can be evaluated as if it were the source code of a method. A simplified version of this is that strings can be used to define messages to send to objects; for example 'aMessage' sendTo: anObject. (The equivalent in Smalltalk is anObject perform: #aMessage). The string used can be the result of a method, and this is commonly called a *computed selector*. In the case of computed selectors, it may not be possible to statically determine the message that will be sent (just as it is not possible to statically determine the result of an arbitrary piece of code).

Each object representing a method slot contains a bytecode vector, which is a parsed version of the method's source. Guru uses parse trees created from these bytecode vectors rather than from a method's source code. Parse trees, rather than bytecode vectors, are used by Guru as they are simpler to use for method comparison (see Section 4.4.1) and for splitting statements into expressions (see Sections 5.2 and 5.3).

Message sends to local variables and arguments of methods appear the same as any other message sends. However, the scoping rules for Self mean that such message sends are easily identified. The scoping rule is that message lookup is from the 'most local' to the 'least local'. For example, consider the following method:

[4]In Self style syntax.

```
m = ( | t |
t a.
[ | t | t a ] value.
self t a)
```

(Ignore the fact that t is uninitialised; it is not important for this example).

- In the first statement, 't a', a is sent to the local variable t of the method m.
- In the second statement, '[| t | t a] value', a is sent to the local variable t of the block, not the local variable of the method.
- In the third statement, 'self t a', a is sent to the result of sending t to self. Note that in this case t does not refer to the local variable of the method, but to some message understood (or otherwise) by self.

Self has *primitive* methods, which are invoked using a message send. All primitive method names start with '_' followed by an uppercase letter. Primitive methods invoke code in the Self virtual machine. The most commonly used primitive methods in application code are '_Clone' and in reflective code '_Mirror'. '_Clone' creates a copy of the receiver in which slots of the copied object refer to the same objects as the slots of the same name in the receiver. (In Smalltalk terminology, '_Clone' creates *shallow copies.*) '_Mirror' creates a mirror object of the receiver, as mentioned in Section 2.3.4.

The Self equivalent of arrays, called vectors, are special objects for which the primitives _At: and _At:Put: are defined. These primitives are used (usually indirectly) to access and assign to *indexed slots*, similar to *indexed instance variables* in Smalltalk. That is, vectors have assignable data slots indexed using integers. The mirror objects (mirrors vectors) associated with vectors are different to those associated with other objects. There are some other objects which also have specific sorts of mirrors, including: byteVectors, canonicalStrings, smallIntegers, floats, and mirrors.

2.4 Example of program deterioration and restructuring

The following example demonstrates how program structure can deteriorate, and how restructuring can be necessary. Consider the objects in Figure 2.9.

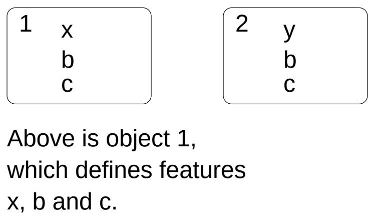


Figure 2.9: Example objects.

The 'natural' inheritance hierarchy for these objects is shown in Figure 2.10. The arrows represent inheritance, and are shown in the direction from children to parent objects. In this example, object 1 defines feature x itself, and inherits features b and c from object 4.

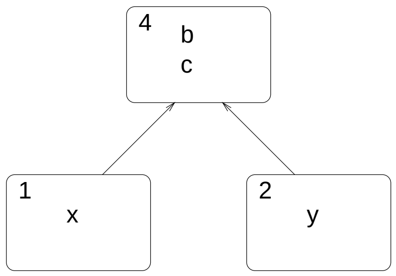


Figure 2.10: Example objects hierarchy.

If a new object is to be added, defining features z b c', where feature c' has the same name as the feature c in the other two objects, but has a different implementation, then the most 'natural' way to extend the inheritance hierarchy is shown in Figure 2.11. (Note that feature c' in the new object (called 'object 3') *overrides* the version of feature c in object 4). This may or may not be the 'best' hierarchy for these objects, but is the way that most programmers would extend the hierarchy of Figure 2.10 to add the new object. When a programmer has to extend an existing hierarchy, he or she may be tempted to choose the easiest way, which may be different to the 'best' way.

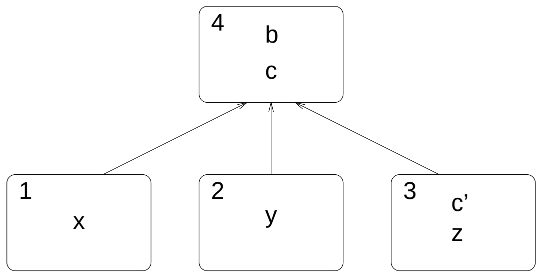


Figure 2.11: Example objects hierarchy.

Now, if feature d is to be added to objects 1 and 2 but not to object 3, the hierarchy will deteriorate unless it is restructured. Consider the options for adding feature d to objects 1 and 2 without restructuring the existing hierarchy:

- Feature d can be duplicated in objects 1 and 2, resulting in the hierarchy shown in Figure 2.12. This is not a good solution as changes to feature d will have to be made in both objects 1 and 2 or they will become inconsistent. Also, this duplication will use more space than necessary.

- Feature d can be defined in object 4, but overridden in object 3 with a special feature which *cancels* the inheritance of this feature, as shown in Figure 2.13. The special cancellation feature states that this feature should not exist for object 3. In this solution, there is no duplication of feature d, but the cancellation of inheritance is seen as a design fault.

- Feature d can be defined in a new object, which is inherited by objects 1 and 2 using multiple inheritance, as shown in Figure 2.14. There is no duplication of feature d, but the features shared between objects 1 and 2 are now split between two objects, which is not a good design.

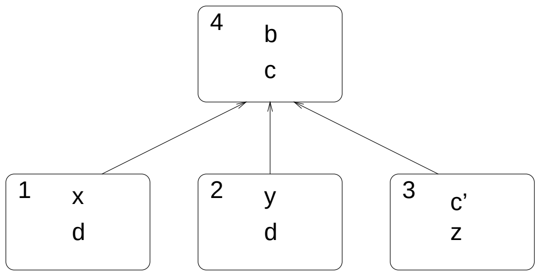


Figure 2.12: Duplicating feature d.

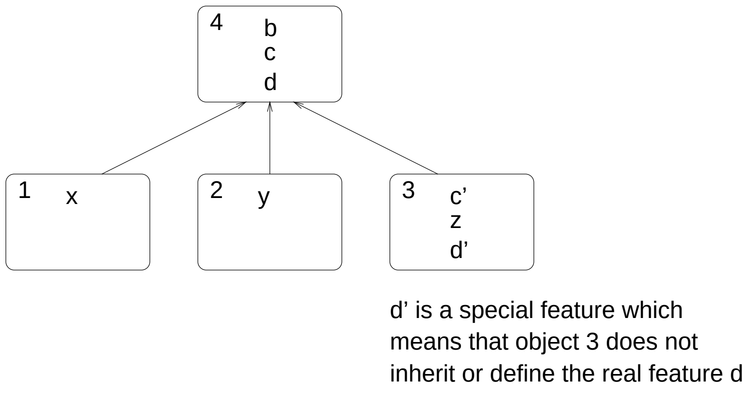

4
b
c
d
1
x
2
y
3
c'
z
d'
d' is a special feature which means that object 3 does not inherit or define the real feature d

Figure 2.13: Cancellation of inheritance.

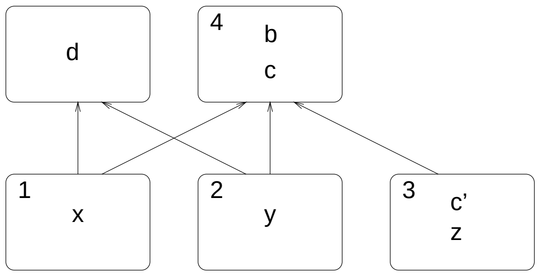


Figure 2.14: Inappropriate multiple inheritance.

The best solution for adding feature d to objects 1 and 2 is to restructure the hierarchy, resulting in the hierarchy shown in Figure 2.15. In this hierarchy, feature d is not duplicated, and the hierarchy shows the relationships between the features and the objects in a clear and simple way. It is now clear that objects 1 and 2 share features c and d, and objects 1, 2 and 3 share feature b.

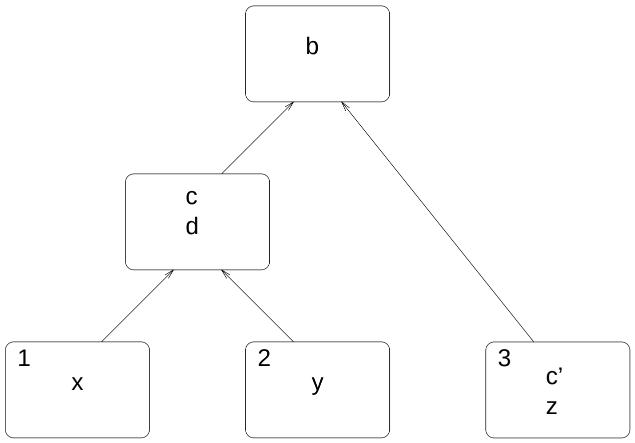


Figure 2.15: Restructured hierarchy.

The aim of Guru is to restructure hierarchies from any badly designed hierarchy or collection of objects into a well designed hierarchy. Any of the hierarchies of Figures 2.12, 2.13 or 2.14 will be restructured into the hierarchy of Figure 2.15.

Chapter 3

Automatic Inheritance Hierarchy Design

Inheritance is one of the defining characteristics of object-oriented programming. It allows programs to capture the shared characteristics of objects, at different levels of abstraction. Designing inheritance hierarchies is difficult, as it requires identifying suitable abstractions and the relationships between them. Texts provide guidelines [Meyer88], but design of inheritance hierarchies remains an art rather than a science. Rather than designing an inheritance hierarchy by hand, an alternative approach is to infer one from the objects that a program creates, or will need to create. This chapter will describe a small collection of algorithms for inferring inheritance hierarchies for any set of objects.

The algorithms are described in two parts; firstly, the *inheritance hierarchy inference* (IHI) algorithm [Moore96a] is presented, which infers hierarchies which satisfy well justified criteria, but do not contain any overriding of slots. Then, algorithms for introducing overriding into the hierarchies produced by the IHI algorithm are presented.

The IHI algorithm was discovered independently of other algorithms, such as those described in [Cook92, Mineau90, Mineau95], which produce similar results. The differences between the IHI algorithm and those described elsewhere are discussed in Section 3.8.

An example of the different levels of abstraction shared by inheritance is that the object 3 is not only an integer but also a number. Hence it shares abstract behaviour with other numbers, as well as more specific behaviour with other integers. The structure of the inheritance hierarchy reflects the abstractions shared between objects. In Self, behaviour (method slots) and data (data slots) can be shared by inheritance, as described in Section 2.3. Instance variable structure is not shared by inheritance in Self, but rather by copying. In this chapter, those aspects of objects that can be shared by inheritance will be called features. Using inheritance to share features increases code reuse, improves consistency and makes maintenance of a system easier.

The algorithms presented are applicable to the design of inheritance hierarchies in any object-oriented programing language, but are described in the terminology of Self (see Section 2.3). Other work [Bergstein91, Casais90, Cook92, Godin93, Hoeck93, Lieberherr91, Mineau95, Pun89] uses the term 'class hierarchy', but as Self is classless, the term 'inheritance hierarchy' is used in this thesis.

Section 3.7 discusses the use of the algorithms for other languages and other possible applications. Chapter 4 describes how these algorithms can be used for automatic restructuring of inheritance hierarchies.

3.1 Introduction

An example of the action of the IHI algorithm is shown below: from the objects of Figure 3.1 it will infer the hierarchy of Figure 3.2. The arrows represent inheritance of features and are shown in the direction from children to their parents. In the inferred hierarchy (Figure 3.2), object A defines feature f3 for itself, inherits feature f2 from its immediate parent and inherits feature f1 from its parent's parent.

In general, the algorithm may construct hierarchies with multiple inheritance,

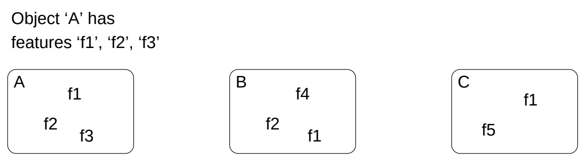


Figure 3.1: A collection of objects with their features

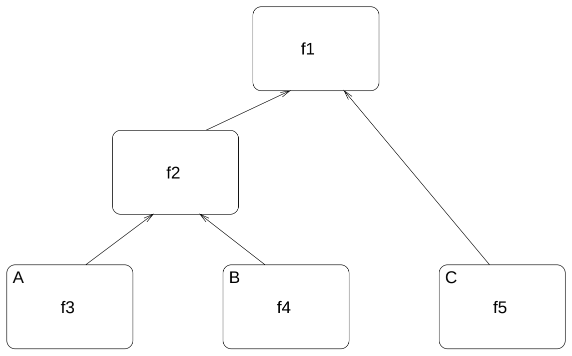


Figure 3.2: The inferred inheritance hierarchy

in which case the diagram will be a graph rather than a tree.

The objects for which a hierarchy is to be inferred (A,B, and C in Figure 3.1) will be called the *original objects*. The objects in the inferred inheritance hierarchy which define or inherit the same features as the original objects (A,B, and C in Figure 3.2) will be called the *replacement objects*. Objects in the inferred hierarchy which are not replacement objects will be called *traits objects*. Where it is unnecessary to make a distinction, replacement objects and traits objects are collectively called (simply) *objects*.

3.2 Criteria for an inferred inheritance hierarchy

A system is defined by its objects and their features; the inferred hierarchy must preserve the features of the original objects, so each original object must have a corresponding replacement object which inherits or defines exactly the set of features that the original defines. The structure of traits objects above the replacement objects and their inheritance links, being new, can take any form, so many hierarchies will satisfy this correctness condition, including the one in which the replacement objects are the same as the original objects, and there are no traits objects. Further criteria must be met if the hierarchy is to be representative of the structure inherent in the objects.

The first of these is that there should be as much sharing of features as possible. That is, every feature should be introduced in exactly one object in the hierarchy. It must appear at least once if it appears in any of the original objects to meet the correctness condition. The motivation for this criterion is the same as the motivation for having inheritance in the language in the first place: sharing makes a system more compact and easier to maintain. The maximum sharing of features is used elsewhere [Pun89] as the sole criterion for constructing a hierarchy. It is desirable to keep the hierarchy as simple as possible, so that its structure can be more readily understood.

The second criterion is that the fewest possible traits objects should be used in the hierarchy. Since for correctness every feature must appear in some object and no feature can appear in a traits object that will be inherited from by an object that does not contain it, this means that objects will contain more than one feature if and only if that combination of features is always found together in the objects which inherit from it.

The third and fourth criteria characterise the inheritance links. The third is that all inheritance that is consistent with the objects should be present in the hierarchy. Thus if all the objects which inherit from some object C also inherit from object D, then C should itself inherit (directly or indirectly) from object D. Therefore, the hierarchy of Figure 3.3 satisfies the third criterion, but the hierarchy of Figure 3.4 does not.

The motivation is that if the objects containing the features of object C (f3 in

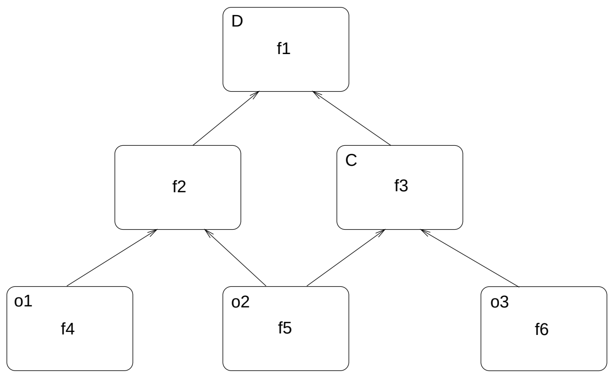


Figure 3.3: Hierarchy with all inheritance consistent with the objects

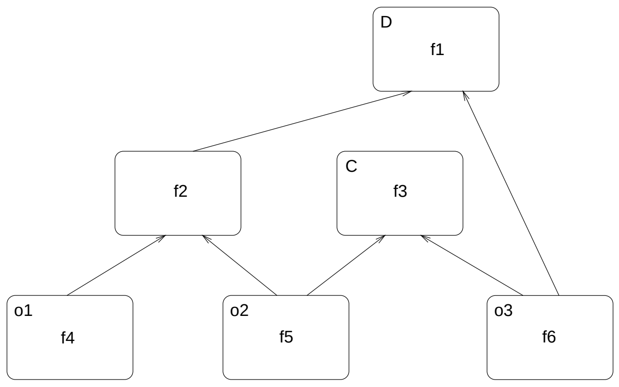


Figure 3.4: Hierarchy which does not satisfy the third criterion

the example) should not inherit from D (those containing f1), the set of original objects should include at least one where f3 does not occur with f1, just as there are objects where f1 occurs without f3.

The fourth criterion applies the general requirement for simplicity to the links; links which are implied by the transitivity of inheritance should not be made explicit. This is equivalent to requiring the minimum number of inheritance links necessary to satisfy the other criteria. To make the implied links explicit only makes the hierarchy more complex than necessary and sends the readers of the code to look immediately at objects that they will eventually encounter anyway.

The first four criteria together define the (maximally compact) (Galois) *knowledge space* [Godin93, Mineau90, Mineau95], also called the *Galois SubHierarchy* [Dicky96].

The fifth criterion is that replacement objects should be leaves of the final inheritance hierarchy. In class based languages, this corresponds to inheritance being only from abstract classes, a criterion for hierarchy design suggested in [Johnson88]. In many cases it is easy to modify hierarchies inferred with this criterion into ones in which this criterion is relaxed, and vice versa. However, there are some cases in which the effect of this criterion makes such conversions more complex. For example, consider the objects (identified with a number, containing features identified by letters) defined in Figure 3.5, taken from an example in [Casais94].

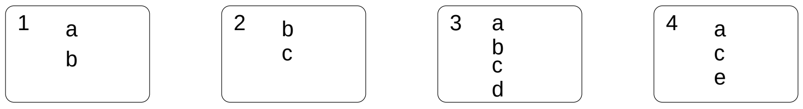


Figure 3.5: Example objects to explain effect of the fifth criteria.

The graph satisfying all of the criteria, including the fifth criterion, is shown in Figure 3.6. The graph satisfying only the first four criteria is shown in Figure 3.7. Note that the fifth criterion results in one more inheritance link than otherwise, as object 3 cannot inherit from objects 1 and 2.

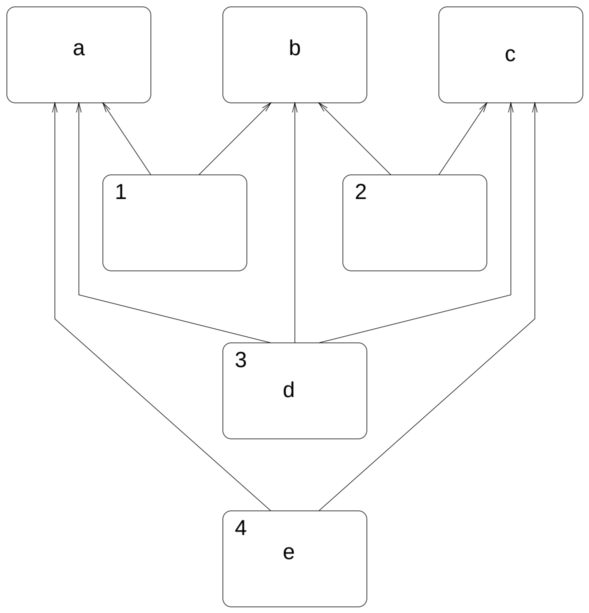


Figure 3.6: Example solution including the fifth criterion.

The five criteria together are sufficient to uniquely define the hierarchy inferred from any set of objects (see Appendix A.1 for a brief justification). Different criteria produce different results; for example, fewer objects would be required if features could be duplicated.

In order to satisfy the first criterion, there must exist a definition of equality of features. A subtle consequence of the features being essentially *atomic* is that overriding will not exist in inferred hierarchies. The original objects cannot define features which override each other, as all the features are defined by the objects directly and not inherited. The IHI algorithm ensures that each replacement object only inherits the features defined by the original object it replaces. Overriding could only exist if replacement objects could have more features in their inheritance paths than they actually inherit (that is, they could appear to inherit more features than they really do). In this case, some of those features

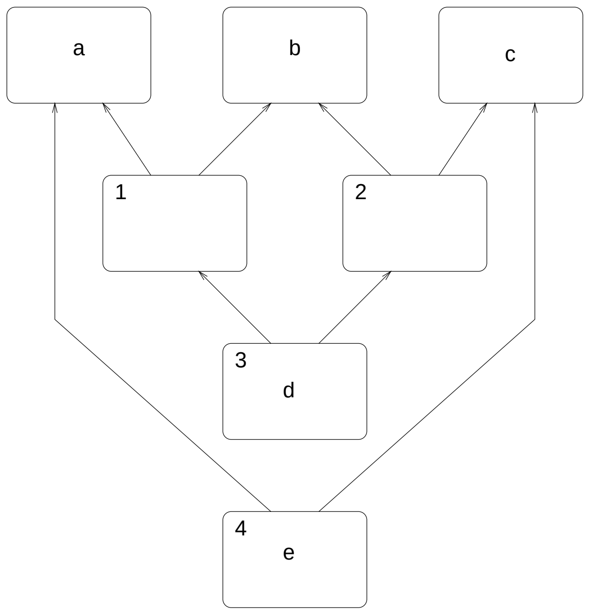


Figure 3.7: Example solution without the fifth criterion.

would have to be overridden, so that they were not actually inherited by the replacement object. As replacement objects can only inherit exactly those features that they require, none of them can be overridden.

3.3 The inheritance hierarchy inference algorithm

The simplest way to describe the algorithm is through an example. It will be presented using graphs with three types of vertex, called FeatureVertices, ObjectVertices and TraitsVertices, and two types of edge, InheritanceEdges and FeatureEdges. ObjectVertices represent the objects for which an inheritance hierarchy is to be inferred. FeatureVertices represent features and FeatureEdges show the ObjectVertices or TraitsVertices in which a feature is defined. TraitsVertices represent the inferred traits objects, and InheritanceEdges represent the inferred inheritance links. Graphs are used to explain the algorithm because they provide an implementation independent representation which is easy to understand. Also, although there is not a direct correspondence with the graph representations used in previous work [Dicky96, Godin93, Hoeck93, Lieberherr91, Mineau90, Mineau95] there are similarities which make comparison easier (see Section 3.8).

Consider the objects in Figure 3.8, shown with their features inside them.

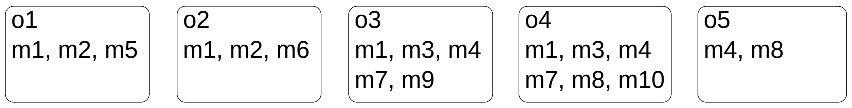


Figure 3.8: Example problem objects

The first step of the algorithm creates a bipartite graph [Schmidt93] with a unique FeatureVertex for each feature, and FeatureEdges from each FeatureVertex to the objects that they appear in. For the example, this would produce the graph shown in Figure 3.9. If each FeatureVertex were used to define a traits object and each FeatureEdge were used to define an inheritance link (in the opposite direction), then replacement objects would inherit all the necessary features and this graph would satisfy the criterion that no feature is duplicated. However, sets of features which are shared by the same sets of objects (such as m3 and m7) are not grouped together, so there may be more traits objects than necessary and the second criterion is not satisfied.

To minimise the number of traits objects in the hierarchy, TraitsVertices are introduced. The next step of the algorithm creates a TraitsVertex for each set of ObjectVertices connected to a FeatureVertex in the initial graph, and labels it with that set. (If more than one FeatureVertex has the same set of objects, this is done only once. For example, 'o3, o4' is a TraitsVertex for both the features m3 and m7.) The FeatureEdges from the FeatureVertices to their objects are then replaced by a single FeatureEdge to the TraitsVertex with those objects as its label. The effect on the example is shown in Figure 3.10.

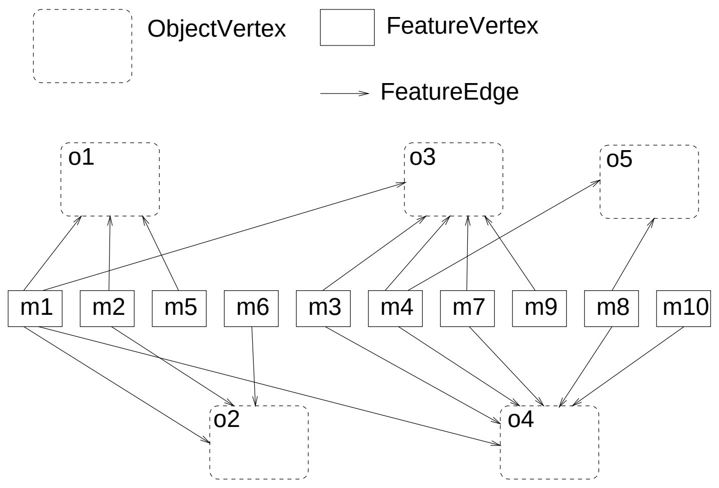


Figure 3.9: Initial grouping graph

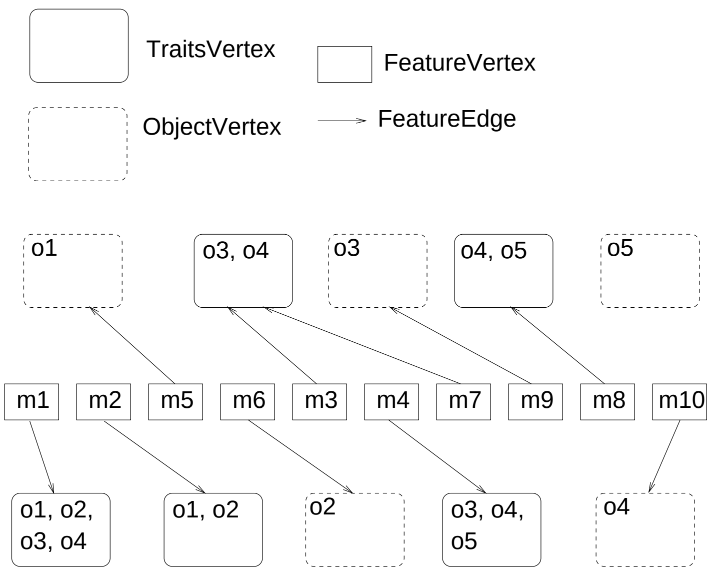


Figure 3.10: Mapping graph with FeatureEdges and FeatureVertices

This graph now represents all the objects that will appear in the inferred hierarchy and the features that they will have. It is called the *mapping graph*. By construction, the TraitsVertices and ObjectVertices partition the features amongst themselves (each FeatureVertex has exactly one FeatureEdge) so there will be no duplication of features in any resulting hierarchy. The label of every TraitsVertex identifies exactly the set of objects containing the features given by the FeatureEdges and no two TraitsVertices have the same label, so the number of traits objects is as small as possible. The mapping graph is useful in its own right because it shows what inherent classifications exist, and which objects belong to those classifications.

However, the InheritanceEdges remain to be constructed. This will be done in two steps. First, enough InheritanceEdges will be added to make sure that objects will inherit the features necessary without inheriting inappropriate features. Then, InheritanceEdges which are not needed, due to transitivity, will be removed. To simplify the following figures, FeatureEdges and FeatureVertices will be shown as just the FeatureVertex labels inside the appropriate TraitsVertex or ObjectVertex, as in Figure 3.11, which represents the same mapping graph as Figure 3.10.

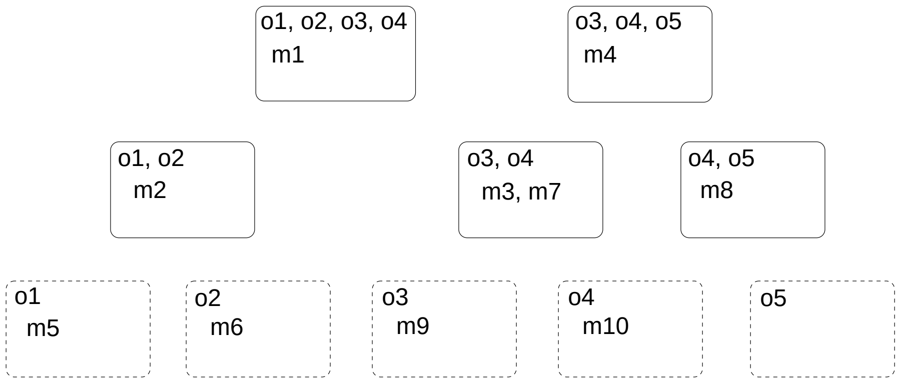


Figure 3.11: The mapping graph as labelled objects

The InheritanceEdges which may be needed in the hierarchy are those connecting each ObjectVertex or TraitsVertex (let its label be the set *os*) to every other TraitsVertex with a superset of *os* as its label. (Considering the labels alone, the resulting graph is a subgraph of the subset inclusion lattice [Schmidt93].) For example, an InheritanceEdge should be added from ‘o1, o2’ to ‘o1, o2, o3, o4’ because ‘o1, o2, o3, o4’ is a proper superset of ‘o1, o2’. The edges from ObjectVertices to TraitsVertices are enough to ensure that each object inherits the features it needs (as Figure 3.12 shows) but gives only a two level inheritance hierarchy. The complete set of edges gives the hierarchy shown in Figure 3.13, and satisfies the third criterion by construction, but clearly does not satisfy the fourth criterion.

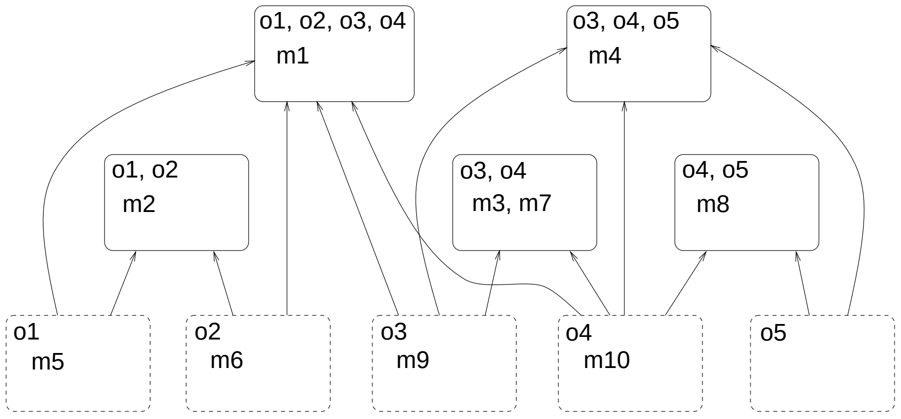


Figure 3.12: Inheritance graph with replacement object-traits object links only

In order to remove InheritanceEdges which are unnecessary due to transitivity; for each TraitsVertex PV and all TraitsVertices CV with an InheritanceEdge (that exists before any edges are removed) to PV, remove all InheritanceEdges to PV from all vertices (both TraitsVertices and ObjectVertices) which have an InheritanceEdge to CV. For example, the InheritanceEdge from (o3) to (o3, o4, o5) is removed because (o3) has an InheritanceEdge to (o3, o4), which has an InheritanceEdge to (o3, o4, o5). The resulting graph will now represent the inferred inheritance hierarchy as objects and their immediate parents, as shown

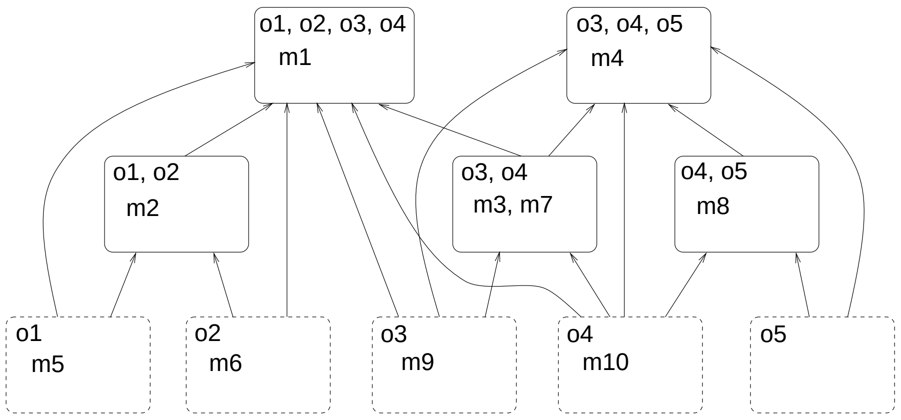


Figure 3.13: Inheritance graph with all edges from third step

in Figure 3.14.

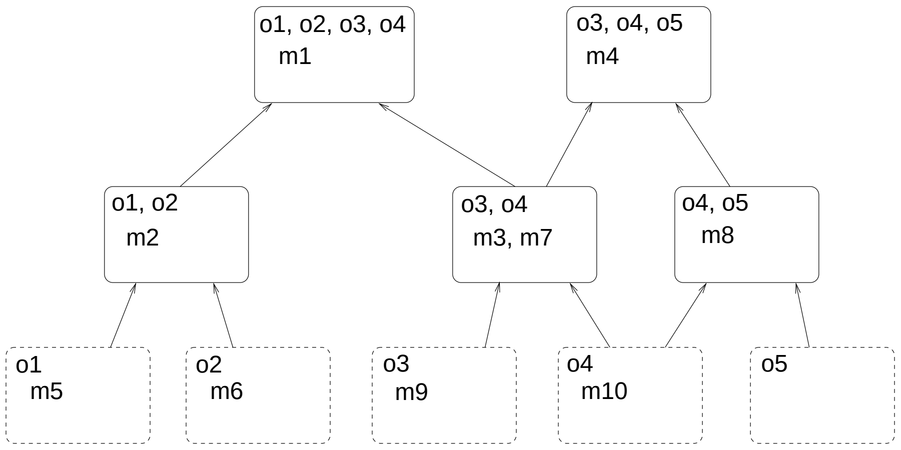


Figure 3.14: Inferred inheritance hierarchy

The time complexity of this algorithm is $O(o^3)$, where o is the number of objects, or somewhat better: a further discussion of complexity is in Appendix A.3.

3.4 An implementation of the inheritance hierarchy inference algorithm

An implementation of the IHI algorithm is described by showing how it would work on the example used in Section 3.3.

Consider the following objects (the same as those shown in Figure 3.8):

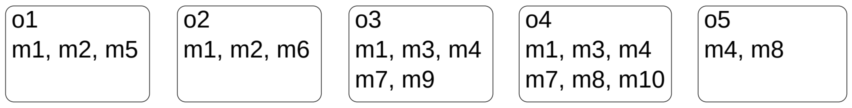


Figure 3.15: Example problem objects

A dictionary is created where the keys are the objects and the values are their features. A dictionary is a collection of key value pairs ('key $\rightarrow$ value'). In the dictionaries used to implement the algorithm, the values are themselves collections.

objects	->	features
o1	->	m1, m2, m5
o2	->	m1, m2, m6
o3	->	m1, m3, m4, m7, m9
o4	->	m1, m3, m4, m7, m8, m10
o5	->	m4, m8

Assuming that objects are different to features, add each key to each value (collection). This is an untidy but efficient way to make sure that the original objects have an equivalent replacement object in the solution, even if that solution object has no unique features and defines only inheritance (as object 5 in this example). The dictionary created is:

objects	->	features
o1	->	m1, m2, m5, o1
o2	->	m1, m2, m6, o2
o3	->	m1, m3, m4, m7, m9, o3
o4	->	m1, m3, m4, m7, m8, m10, o4
o5	->	m4, m8, o5

This dictionary is 'swapped around', so that the elements in the value collections are now the keys.

For example, for the key value pair 'o1 $\rightarrow$ m1, m2, m5, o1' the following key value pairs are added in the resulting dictionary:

features	->	objects
m1	->	o1
m2	->	o1
o1	->	o1
m5	->	o1

such that if there is already a value for the key, e.g. if adding 'm1 $\rightarrow$ o1' when 'm1 $\rightarrow$ o2' already exists, the value is added to a collection, that is, the key value pair in the resulting dictionary would become 'm1 $\rightarrow$ o1, o2'.

The resulting dictionary has features as keys, and the objects they appear in as values. This dictionary is the equivalent of the 'initial grouping graph' in Figure 3.9.

features	->	objects
m1	->	o1, o2, o3, o4
m2	->	o1, o2
o1	->	o1
o2	->	o2
m5	->	o1
m6	->	o2
m3	->	o3, o4
m4	->	o3, o4, o5
m7	->	o3, o4
o3	->	o3
m8	->	o4, o5
o4	->	o4
o5	->	o5
m9	->	o3
m10	->	o4

This dictionary is then ‘swapped around’ again, but in a different way. The ‘swap around’ is done for the values as a whole, rather than for each element in the collections which are the values. For example, the key value pair, ‘m3 $\rightarrow$ o3, o4’ adds ‘o3, o4 $\rightarrow$ m3’ to the result (taking into account whether ‘o3, o4’ already exists). For example, if the key value pair ‘o3, o4 $\rightarrow$ m7’ is already in the result, it becomes ‘o3, o4 $\rightarrow$ m3, m7’. In the resulting dictionary, all combinations of objects which share a feature or features are the keys, and the values are all the features they share.

collection of objects	->	features
o1	->	o1, m5
o2	->	o2, m6
o1, o2	->	m2
o3, o4, o5	->	m4
o1, o2, o3, o4	->	m1
o3, o4	->	m3, m7
o3	->	m9, o3
o4, o5	->	m8
o4	->	m10, o4
o5	->	o5

This dictionary now represents all the new traits objects that will be needed and all the features that they will have. It is called the *mapping dictionary*, and is the equivalent of the ‘mapping graph’ in Section 3.3. Each ‘collection of objects’ entry in the dictionary represents a new inferred object. Each ‘collection of objects’ entry with only one original object will be the corresponding replacement object in the restructured hierarchy. By construction, there will be exactly one such entry per original object.

The new traits objects that will be needed have been discovered, and the next stage is to work out the inheritance hierarchy.

A dictionary is made with each collection as its key and those collections which are proper subsets as its value, for those collections which have proper subsets. This represents the inheritance links from parents to children in the inferred hierarchy (rather than the conventional representation of inheritance links used in Section 3.3, from children to parents). This results in the following dictionary, called the *offspring dictionary*:

collection of objects	->	collections of objects which are subsets
o4, o5	->	(o5), (o4)
o3, o4, o5	->	(o3, o4), (o4, o5), (o5), (o4), (o3)
o1, o2	->	(o1), (o2)
o1, o2, o3, o4	->	(o3, o4), (o1, o2), (o1), (o4), (o2), (o3)
o3, o4	->	(o4), (o3)

This now represents the new traits objects and all their offspring, that is, not only their children but their children’s children etc. It is the equivalent of the ‘Inheritance graph with all (inheritance) edges’ (Figure 3.13) in Section 3.3 (with the direction of inheritance represented in reverse).

In order to remove unnecessary inheritance (due to transitivity), a copy of the offspring dictionary is created in which, for each traits object, the children of those children which also have children are not included. For example, the traits object labelled (o3, o4, o5) has children (o3, o4), (o4, o5), (o5), (o4) and (o3). Of those children, (o3, o4) and (o4, o5) also have children, so their children, labelled (o5), (o4) and (o3) are not included in the collection of children of (o3, o4, o5) in the new dictionary. This is effectively the same as the description in section 3.3, but may appear superficially different due to the representation of inheritance from parents to children used in the implementation. Note that only the children of objects in the collection of objects need to be considered when looking for subsets of the labels of parent object labels, rather than looking for the all possible subsets of an object's label. Furthermore, objects whose label is one element do not need to be considered as they cannot have children (their label cannot have any non-empty proper subsets). Therefore, in the example above, for object (o3, o4, o5) only the children (o3, o4) and (o4, o5) need to be considered for the possibility that they themselves have children. This is more efficient than if all subsets of a label have to be considered; for example if we had to check whether there was an object with the label (o3, o5) in addition to the objects with labels (o3, o4) and (o4, o5). Calculating all subsets of a collection has exponential performance.

The resulting dictionary (called the *inheritance dictionary*) is:

collection of objects	->	collections of objects which are immediate children
o4, o5	->	(o5), (o4)
o3, o4, o5	->	(o3, o4), (o4, o5)
o1, o2	->	(o1), (o2)
o1, o2, o3, o4	->	(o3, o4), (o1, o2)
o3, o4	->	(o4), (o3)

This now represents the new inheritance hierarchy, as objects and their immediate children. This is the equivalent of Figure 3.14 in Section 3.3 (with inheritance represented in the reverse direction).

Some objects in the mapping dictionary may not appear in the inheritance dictionary produced, as not all objects will have children. Remember that all of the new objects and their non inherited features will be given by the mapping dictionary. The only additional information of the inheritance dictionary is the parentage of new objects. The result can be converted into a dictionary with inheritance from children to immediate parents, by swapping around the dictionary, such that the items in the collection values become the keys, and their keys become the values. This is the equivalent of Figure 3.14 in Section 3.3.

This produces the following dictionary:

collection of objects representing children	->	collections of objects representing immediate parents
o4, o5	->	(o3, o4, o5)
o1	->	(o1, o2)
o2	->	(o1, o2)
o4	->	(o4, o5), (o3, o4)
o5	->	(o4, o5)
o3, o4	->	(o3, o4, o5), (o1, o2, o3, o4)
o1, o2	->	(o1, o2, o3, o4)
o3	->	(o3, o4)

This (or the inheritance dictionary) and the mapping dictionary now define the inferred hierarchy.

3.5 Reintroducing overriding into inferred hierarchies

The IHI algorithm does not consider overriding, and this can in some circumstances lead it to infer unnatural inheritance hierarchies. While too much overriding may be an indication of poor design, appropriate use of overriding can improve the design of inheritance hierarchies. Overriding can capture the informal idea that something almost falls into a particular ‘class’ (that penguins are birds, for example, even though they do not fly) and hence reduce the number of traits objects in the hierarchy. Without overriding, inheritance hierarchies may have a structure where the majority of some behaviour is inherited from one object, but other small objects may have to exist for sharing related behaviour between subsets of those objects. Thus, related behaviour may have to be split between more objects than necessary if overriding is not allowed. Overriding can allow an object to inherit most of its behaviour from another object, but override some slots in order to specialise its behaviour, thus avoiding the need for an additional object to define that specialised behaviour. Figure 3.16 shows a hierarchy (used as an example in the released version of Guru [Guru]) which does not contain overriding. Figure 3.17 is an equivalent hierarchy which does contain overriding.

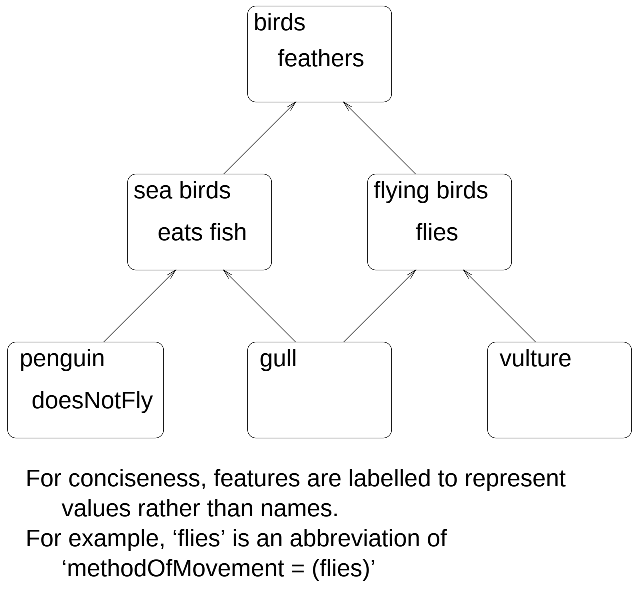

birds
feathers
sea birds
eats fish
flying birds
flies
penguin
doesNotFly
gull
vulture
For conciseness, features are labelled to represent values rather than names.
For example, ‘flies’ is an abbreviation of ‘methodOfMovement = (flies)’

Figure 3.16: Hierarchy without overriding.

Two algorithms for reintroducing overriding are presented. Chapter 6 includes the results of applying the approach described in Section 3.5.2 on realistic examples.

The reintroduction of overriding has been implemented so that it can be used with any hierarchy, irrespective of whether it results from the IHI algorithm. The IHI algorithm followed by the reintroduction of overriding is called the *extended IHI algorithm*.

In the following algorithms, features are now defined both by their *name* and their *implementation*. Features of the same name can override each other. Features are only equivalent if they have the same name *and* the same implementation. A feature with the name ‘A’ is simply called a ‘feature A’. If an object defines or inherits a feature A, then it is said to *understand* A.

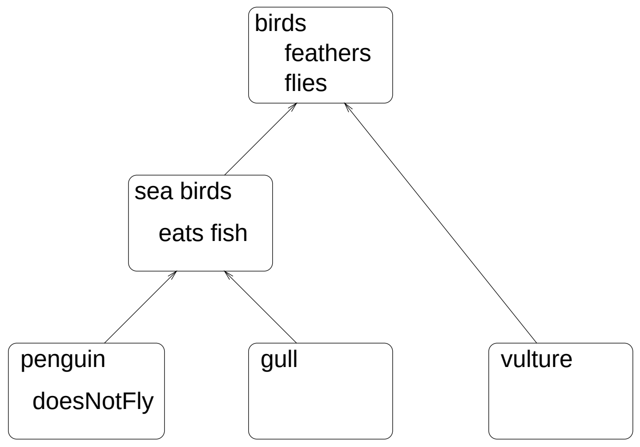


Figure 3.17: Hierarchy with overriding.

3.5.1 Identifying default implementations

This approach attempts to reintroduce overriding of a 'default' or 'standard' implementation of a feature by 'non-standard' versions of that feature.

The *replacement offspring* of an object A are those replacement objects which inherit from A. The *shared protocol* of object A is the set of feature names understood by all the replacement offspring of A, excluding features inherited from any object which A inherits from. This includes the set of feature names defined by A, but will also include other feature names if the replacement offspring of A define features of the same name.

All features whose names are in the shared protocol of A, in all the objects which inherit from A, and which do not override another feature defined in an object which inherits from A, are considered for being moved to A. The following restrictions determine whether a feature M is moved to A.

Firstly, it must be *possible* to move feature M to object A. Some features cannot be moved because of their implementation, which will depend on the details of the use of the reintroduction of overriding algorithms. Details specific to the reintroduction of overriding in Self programs are considered below; other considerations may be necessary for other languages or uses. Self assignable slots cannot be moved from leaf objects. If an assignable slot is moved from a leaf object to any object which it inherits from, then copies of this leaf object will share one assignable slot rather than each defining their own assignable slot. Therefore, the behaviour of programs in which leaf objects are copied would not be preserved, and copying leaf objects is very typical in Self programs. Also, Self methods that contain resends may not be moved in some circumstances. If a method contains a resend, it may not be moved to an object from which the resend invokes a different method. For example, consider the objects shown in Figure 3.18. The resend inside method `n` in object `o1` causes method `m` in object `p1` to execute. If method `n` is moved to object `p1`, then the meaning of this method is altered, as the resend would cause method `m` in object `p2` to execute instead.

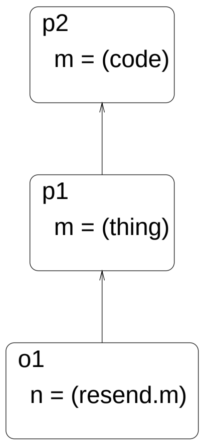


Figure 3.18: Reintroducing overriding must not alter resends.

Similarly, a slot should not be moved if it will result in a resend elsewhere in the system being altered. In Figure 3.19, if slot `s` is moved from object `o2` to object `t1`, then a resend of `s` in a method in object `o1` will be altered, as it will now cause `s` in object `t1` to execute instead of `s` in object `t2`.

Secondly, A must not already contain a feature M. This is because it is not possible to move another feature M to A in such circumstances.

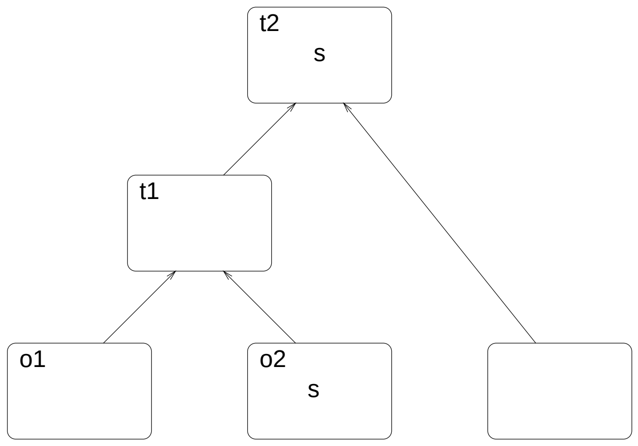


Figure 3.19: Reintroducing overriding must not alter resends.

Thirdly, moving feature M to A must not introduce an ambiguity. In Figure 3.20, if method m in object Y was moved to object A, then the message ‘m’ sent to object Y would be ambiguous. This introduces an error unless the ambiguity is resolved. Such ambiguities can be resolved by automatically creating a method which uses a directed resend to disambiguate the method to be executed (see Section 2.3.1). If a disambiguating method needs to be created in order to move a method, and no slots or objects are removed by moving that method, then an extra method will have been introduced into the hierarchy, with no benefit, which is not desirable. Therefore, it is better not to introduce the ambiguity in the first place.

The fourth consideration is that the implementation of M must be the one used by the largest number of replacement offspring of A. This is to meet the requirement that the ‘default’ implementation of M is inherited by all the replacement offspring of A, and overridden by those replacement offspring of A which require a ‘non-standard’ implementation. Note that this does not mean that the implementation must be duplicated many times, but that this implementation is used to implement M for the largest number of replacement offspring. In the example shown in Figure 3.21, the replacement offspring of object A are the

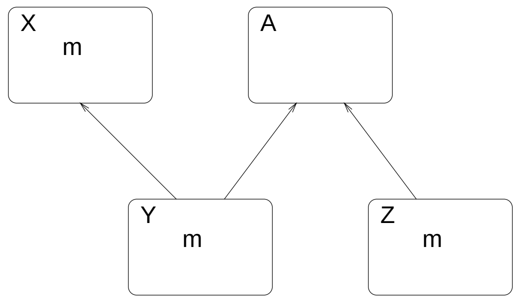

X
m
A
Y
m
Z
m
The implementations of m in objects X,Y and Z are different.

Figure 3.20: Reintroducing overriding must not introduce ambiguities

leaves of the hierarchy. Feature m in object B is inherited by 3 of the replacement offspring of A, and m in object C is only inherited by 2. Therefore, it is m in object B which is moved to object A. If there is no single implementation used by the largest number of replacement offspring, then none is chosen.

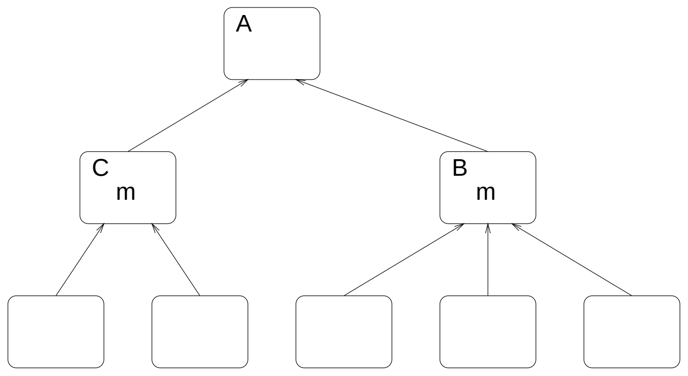


Figure 3.21: The 'default' implementation is to be overridden by the 'non-standard' implementations.

The IHI algorithm will result in there being no duplication of implementations of features of the same name; however, the reintroduction of overriding has been designed so that it can be used for hierarchies whether they result from the IHI algorithm or not. Different features which have the same implementation are treated as equivalent, and so when an implementation of a feature is moved to reintroduce overriding, all equivalent features that can then be removed are removed. Figure 3.22 shows an example where three equivalent implementations (in objects D, E and F) would be removed, and one of them moved to object A.

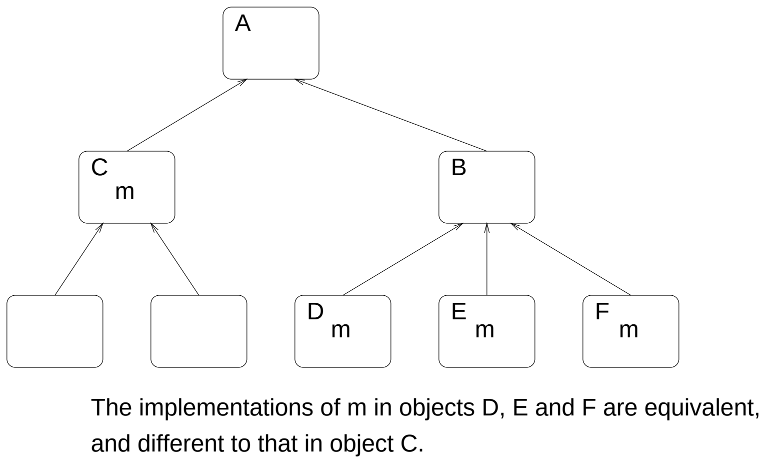

A
C
m
B
D
m
E
m
F
m
The implementations of m in objects D, E and F are equivalent, and different to that in object C.

Figure 3.22: The 'default' implementation may be defined in more than one method.

Finally, a feature is not moved to A unless its implementation is a *possible* implementation for all the replacement offspring of A, even though it is overridden by some of them. In Self, this means that all replacement offspring of A understand all the messages sent to implicit self in that implementation (the messages sent to implicit self are those messages which do not have an explicit receiver, as explained in Section 2.3). The reason for this is to make sure that the feature to be moved represents a 'default' implementation to be overridden by some objects, rather than an implementation which is applicable only to certain objects. In Figure 3.23 the method includes: in object OrderedCollection can be moved to object Collection because all the messages sent to self in this method (i.e. do:) are understood by Set, so this implementation would work for Set. The reason Set has a different implementation is for efficiency. In this example, moving includes: from OrderedCollection to Collection 'makes sense' as it is overridden by includes: in Set for efficiency, rather than because it is not applicable. In Figure 3.24 the method add: cannot be moved from object OrderedCollection to object Collection because it sends the messages makeSpaceAtEnd and last: which are understood by children of OrderedCollection but not by Set. Therefore, this implementation is not applicable to Set, so should not be in its inheritance path.

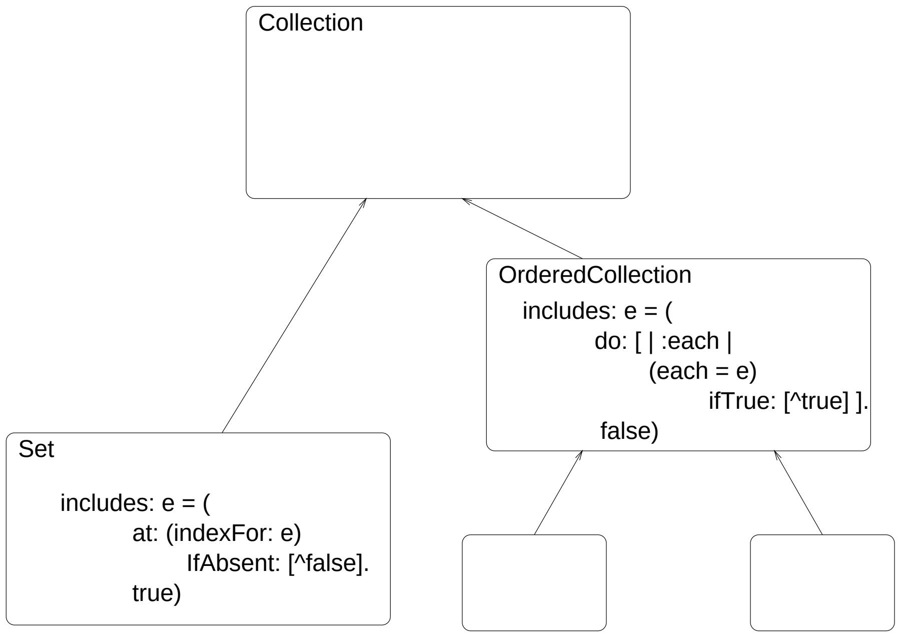

Collection
OrderedCollection
includes: e = (do: [| :each | (each = e) ifTrue: [^true]]. false)
Set
includes: e = (at: (indexFor: e) IfAbsent: [^false]. true)

Figure 3.23: A 'default' implementation which is applicable to all replacement offspring.

If moving features results in any objects defining no features (because all of them have been moved higher in the inheritance hierarchy) then that object should be removed as described in Section 3.5.3.

3.5.2 Removing anomalous traits objects

An alternative approach to reintroducing overriding is described in this section. Rather than attempting to identify default implementations which can be overridden where necessary, the approach described in this section is to simplify the inheritance hierarchy by removing objects which represent anomalies due to overriding not being considered by the IHI algorithm. The approach considers only objects below an arbitrary size (number of features), which can be chosen by the

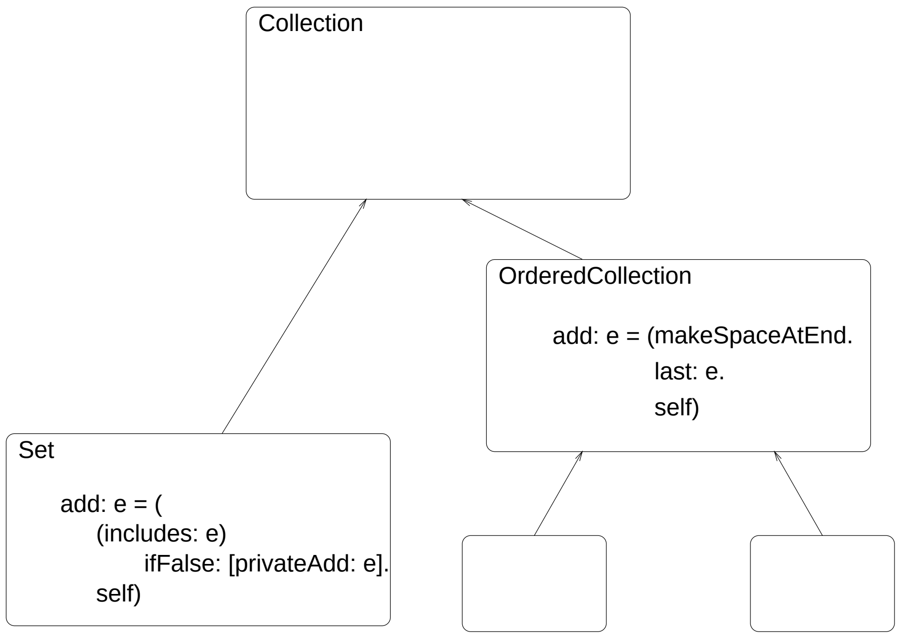


Figure 3.24: A 'default' implementation which is not applicable to all replacement offspring.

user.

If all the features of such an object may be moved to a parent, then they are moved and the hierarchy restructured as described in Section 3.5.3. If this empty object, E, is not removed from the hierarchy before the next object is considered for the reintroduction of overriding restructuring, features could be moved into E, and the chance to remove it from the hierarchy would be lost.

The order that potentially anomalous traits objects are removed affects the resulting hierarchies. In order to remove as many as possible, the potentially anomalous traits objects are considered for removal in the order of highest (that is, the objects with the deepest chain of children) first.

A feature may only be moved to a parent if it does not introduce an ambiguity and if it is a possible implementation for all the replacement offspring of the parent, as explained in Section 3.5.1.

3.5.3 Removing empty traits objects

This section describes how a traits object which defines no features (because they have been moved by the reintroduction of overriding) can be removed from an inheritance hierarchy. All the children of such an object are modified so that their parents include all the parents of the object to be removed. Figure 3.25 shows an example of a hierarchy containing a traits object which does not define any features. The children of object A are modified so that they inherit from the parent of object A, thereby removing object A from the hierarchy, as shown in Figure 3.26.

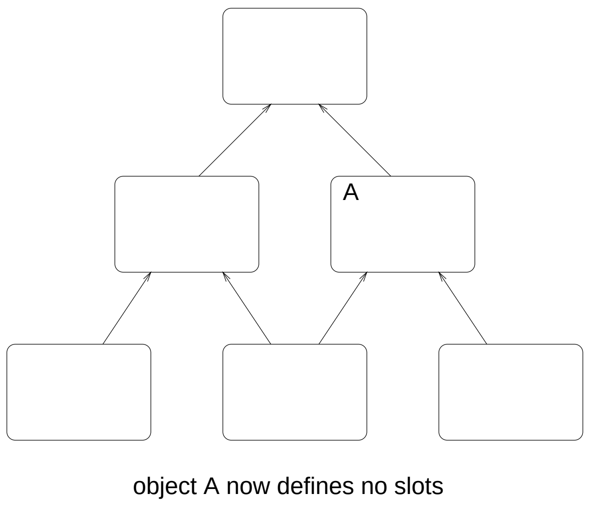


Figure 3.25: Hierarchy containing an empty object

Note that removing an empty object can result in inheritance links which are unnecessary due to transitivity, as the example in Figure 3.26 shows. These unnecessary inheritance links can be removed using an algorithm similar to that described in Section 3.3. The resulting hierarchy is shown in Figure 3.27.

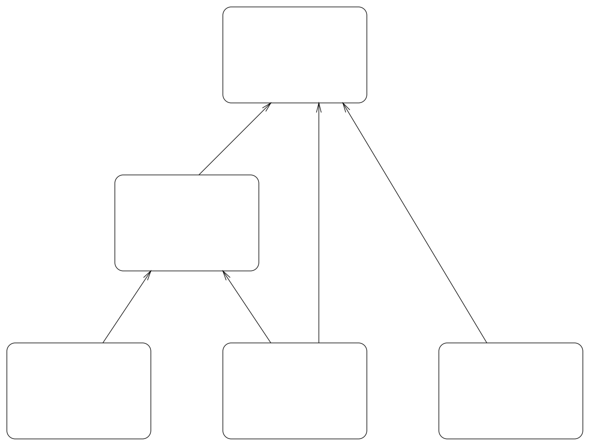


Figure 3.26: Hierarchy after removing empty object

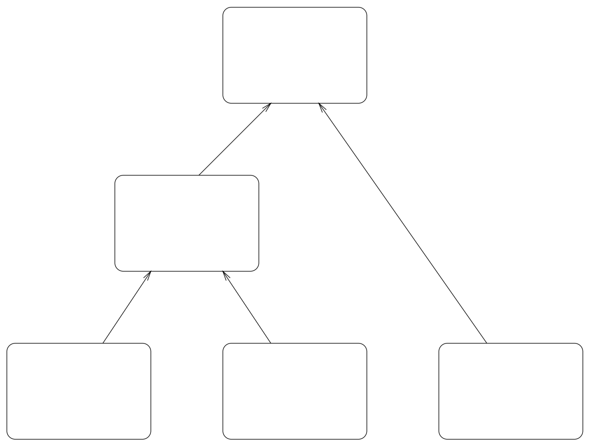


Figure 3.27: Removing transitively unnecessary inheritance

3.6 Limitations and problems

The extended IHI algorithm aims to infer hierarchies which reflect the inherent structure of objects. This hierarchy may not be ideal for future reuse and may not reflect real world abstractions, as these are not possible to infer from objects and their features alone. A programmer may use information from outside a system when creating an inheritance hierarchy, in particular domain knowledge and knowledge gained from experience, to produce hierarchies which reflect predictions for future extensions and reuse of a hierarchy.

However, as discussed in Chapter 4, even programmers cannot predict the future accurately, so hierarchies may often require restructuring after changes have been made, despite attempts to make them easy to evolve and reuse.

The reintroduction of overriding relies on the existence of suitable parent objects to move slots into. In some cases, such parent objects may not exist, and the algorithms for reintroducing overriding will not be able to work. In practice, such situations are rare as, if objects are closely enough related that they could share a feature which was overridden by some of them, then they should already share at least one feature not overridden by any of them, and hence a suitable parent object will exist.

3.7 Applications of the inheritance hierarchy inference algorithm

The extended IHI algorithm has been designed to create inheritance hierarchies for any group of objects. The precise meaning of features and their equality has to be defined according to the intended use of the algorithm, and the sort of programming language for which it is being used.

If features are methods defined only by name, then the result will be a hierarchy of messages understood, which is used as the meaning of 'type' by some users of dynamically typed languages. Hierarchies of messages understood are discussed in [Cook92], and are called *protocol hierarchies*.

Alternatively, for inferring a 'type hierarchy' for a statically typed language, features could be defined by their name and their type. This definition of features has been used in the implementation of a design tool [Pun90].

Chapter 4 explains how the extended IHI algorithm has been used in the implementation of a restructuring tool [Moore95] for the dynamically typed language Self [Ungar87], defining features by their name and their 'meaning', where the meaning of a method is a parsed version of it. For restructuring a statically typed language, method features would also have to include the types of arguments and return values.

The extended IHI algorithm may be useful for data mining [Holsheimer94] and conceptual clustering [Fisher87] on small sets of data. However, the extended IHI algorithm uses only information about the features of objects and does not use heuristics for approximating a good hierarchy for very large collections of objects, or heuristics based on domain knowledge. Therefore, there are only limited circumstances in which the extended IHI algorithm is likely to be suitable for such applications.

3.8 Comparison with previous work

Problems similar to inferring inheritance hierarchies appear in several areas of research, in particular data mining [Holsheimer94] and conceptual clustering [Fisher87]. However, these applications are significantly different to the inference of inheritance hierarchies because they involve inferring an approximate hierarchy from a large set of possibly incomplete data (for example, [Cheeseman88] creates hierarchies based on probabilities), rather than inferring a precise hierarchy from a (relatively) small set of complete data.

There have been other investigations of automatic inheritance hierarchy construction from object descriptions, producing hierarchies which satisfy different criteria. The criteria used by the IHI algorithm will be referred to as the IHI criteria.

3.8.1 Bergstein, Lieberherr et al

Bergstein [Bergstein91] presents a notation for representing inheritance hierarchies, called *class dictionary graphs*, and a collection of transformations. He proves that these transformations preserve the behaviour of objects, and are the minimal number necessary to be able to transform a class dictionary graph into any other equivalent one. He limits the hierarchies considered to those which do not contain cyclical inheritance. His work is used as the basis for the inheritance hierarchy restructuring algorithms described in [Lieberherr91].

Lieberherr et al [Lieberherr91] adopt the first two IHI criteria, but replace the other IHI criteria with a requirement that the number of inheritance links should be minimised. This clearly subsumes the fourth IHI criterion, but gives different results from the third IHI criterion. For example, while the third IHI criterion will give the hierarchy shown in Figure 3.28, the minimality requirement of [Lieberherr91] forces one of the top edges to be removed, leaving a hierarchy like that of Figure 3.29. (Any other top edge could be removed instead.) This reduces the amount of inheritance, which is generally desirable, but at the expense of an arbitrary decision which does not reflect the structure inherent in the objects.

The first part of Lieberherr's algorithm produces a graph in what he calls *Common Normal Form*, which is equivalent to the mapping graph produced in the IHI algorithm. The construction of the inheritance hierarchy from the Common Normal Form graph involves the equivalent of examining pairs of vertices for every combination of the incoming InheritanceEdges of each of the TraitsVertices, and modifying the graph for one pair of vertices at a time, until no more modifications can be made. Despite its greater complexity, it does not guarantee to satisfy either

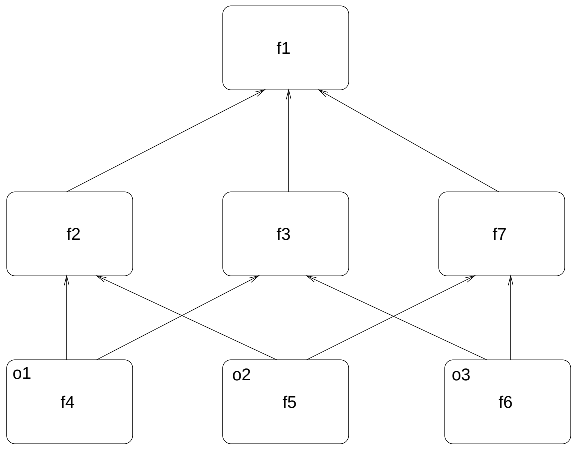


Figure 3.28: Hierarchy produced by the IHI algorithm

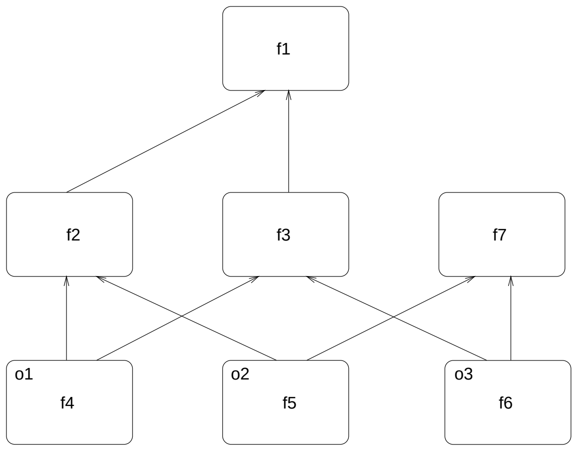


Figure 3.29: Hierarchy satisfying minimal inheritance criterion

their criteria or the IHI criteria; a reason for this is discussed in Appendix A.3.

The IHI algorithm can be extended to meet Lieberherr's criterion by adding a new final step. The label of a TraitsVertex is given by the set of replacement objects which inherit from the traits object. This must be equal to the union of the labels of its children, since the inheritance passes through them, but their labels will not necessarily be disjoint (due to multiple inheritance paths). The new step considers every TraitsVertex CV, to find the smallest set S which has the union of the labels of S equal to the label of CV. This is not necessarily unique, as Figures 3.28 and 3.29 show. The InheritanceEdges to CV from the children not in S are then deleted. The result is a hierarchy where the features inherited by each object are unaffected, but the number of InheritanceEdges is minimised. This new final step will require time exponential in the number of children of each TraitsVertex, as it is equivalent to the minimum cover problem which is known to be NP-complete [Garey79]. In the worst case, the number of children of a TraitsVertex may be the number of original objects. Therefore, in the worst case the final step will be exponential. However, in practice the maximum number of children of any TraitsVertex is often small enough for this final step to be feasible if required.

3.8.2 Cook

Cook [Cook92] describes algorithms for extracting *protocol* information from Smalltalk classes, and for creating a *protocol hierarchy* for that set of classes. The meaning of *protocol* is the set of messages understood by objects of a class, ignoring methods which cancel inherited methods (shouldNotImplement) and methods which have not been implemented (subclassResponsibility). Cook's motivation for creating protocol hierarchies is to identify design problems in inheritance hierarchies; he analyses the Smalltalk collection classes as an example. He concludes that the protocol hierarchy represents a *client's* view of a collection of classes, and an inheritance hierarchy is an *implementor's* view. Furthermore, these two hierarchies are different in the case of the Smalltalk collection classes, and may be different in general.

The IHI algorithm is similar to Cook's algorithm for creating a protocol hierarchy from a set of protocols, but was discovered independently. There is a considerable difference in the detail that Cook gives in describing his algorithm; he does not explicitly state all the properties of the hierarchies produced, nor does he fully explain how to handle classes which do not define any unique features[1], and his explanation of building the inheritance links is terse [Cook92]:

> "A topological sort by set inclusion of the sets of classes in the domain *hierarchy* gives the protocol hierarchy."

(The 'domain *hierarchy*' is the equivalent of the 'initial grouping graph' in Figure 3.9.)

3.8.3 Dicky et al

Dicky et al [Dicky96] describe an algorithm for incrementally inserting a class into a hierarchy while maintaining a Galois SubHierarchy. Consequently, their hierarchies satisfy the criteria of [Mineau90, Mineau95] and the first four criteria of the IHI algorithm.

The main innovations of their work are considering overriding of features, and incremental insertion of classes while maintaining the properties of the hierarchy.

The hierarchies produced by their algorithms will not necessarily be the same as those produced by the extended IHI algorithm. This is because their algorithm relies on a pre-determined order that features can override each other. They suggest ways of semi-automatically determining a suitable ordering for overriding of features. As the algorithms described in Sections 3.5 use whatever ordering of overriding that is most useful given the results of the IHI algorithm, they may

[1]In Section 3.4 it is explained how a unique feature is explicitly added to each 'class' to avoid this problem.

be capable of producing better results than if constrained by a pre-determined order.

An algorithm which allows classes to be inserted incrementally into a hierarchy would appear to be preferable to a non-incremental algorithm such as the extended IHI algorithm. However, there are problems using such an incremental algorithm in practice, because it relies on the hierarchy being in a particular state before inserting a new class. An inheritance hierarchy cannot realistically be maintained in the appropriate state, as this would involve restructuring the hierarchy every time a method or object is added, deleted or modified. Furthermore, as their algorithm maintains a Galois SubHierarchy, it cannot be applied to part of a hierarchy, but must be applied to an entire hierarchy. In practice, a programmer will not want a hierarchy to be restructured too often, and will want to only restructure part of a hierarchy, as the overhead for learning a restructured hierarchy may be considerable. In practice, restructuring is best performed at programmer determined times and on programmer specified groups of objects. For example, a programmer can specify a restructuring for a hierarchy once the (predicted) improvement in hierarchy structure more than compensates for the overhead of learning a new restructured hierarchy.

3.8.4 Godin et al

Godin et al [Mineau90, Mineau95] describe an algorithm, discovered independently, which produces results similar to those of the IHI algorithm. Their algorithm was originally used for knowledge acquisition from data bases, but Godin [Godin93] also describes using Galois lattices as the basis for inheritance hierarchy structuring.

Their hierarchies satisfy the first four IHI criteria, but not the fifth. That is, in the results of their algorithm, the equivalent of preserved objects do not have to be leaves of the inheritance hierarchy. Section 3.2 shows how this can affect the hierarchies inferred.

In their algorithm, features of objects are represented as a collection of triples of the form (object,relation,object), for example, (koala,eats,gumTreeLeaves). For each object all those triples are also included with 'wild-cards' (indicated here as '?') replacing the actual values in the triples. For example, (?,relation,object), (object,relation,?), (?,relation,?), (object,?,object), (?,?,object) et cetera. This requires eight such triples for every feature. The benefit from including these triples is that additional potentially useful information is inferred, in particular generalisations can be inferred, such as 'all animals have a certain number of legs'.

Their algorithm involves building a table of these triples, and then constructing the inferred inheritance hierarchy using a single pass through each element of the table. The way their algorithm constructs inheritance edges is similar to the IHI algorithm, but uses one slightly more complicated stage rather than two simple stages.

The computational complexity of their algorithm is claimed to be $O(n^2)$, where 'n' is the number of original objects.

3.8.5 Light

Light [Light93] describes an algorithm for inserting a new 'class' into an existing inheritance hierarchy. Despite the difference in application area between his work, which is in natural language lexicons, and object-oriented programming, there are similarities which motivate the inclusion of a reference to his work. He presents a formalisation of the problem of inserting a class into an existing hierarchy, and shows that (for his definition) this is an NP-complete problem. He then presents an efficient, but approximate, algorithm for solving this problem.

3.8.6 Pun and Winder

The hierarchies constructed by Pun and Winder [Pun89] satisfy the first IHI criterion, but are not guaranteed to satisfy the other IHI criteria. Furthermore, they favour multiple inheritance over single inheritance. The stronger IHI criteria, which have been justified on general grounds, seem to lead to better hierarchies. An example taken from [Pun90] shows the difference between the results of their algorithm and the results of the IHI algorithm. The objects in this example are defined as shown in Figure 3.30. Their algorithm produces the hierarchy in Figure 3.31, while the IHI algorithm produces the hierarchy shown in Figure 3.32.

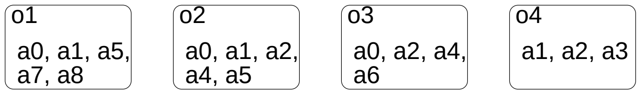


Figure 3.30: An example from [Pun90]

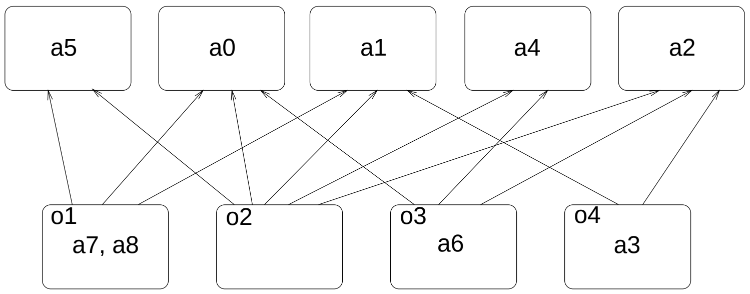


Figure 3.31: Hierarchy produced by [Pun90]

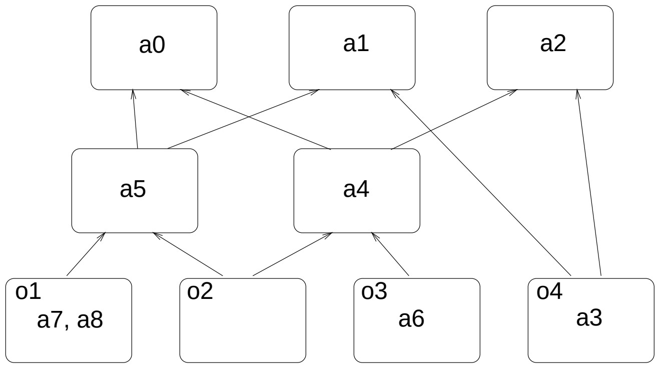


Figure 3.32: Hierarchy produced by the IHI algorithm

Their algorithm involves iteratively factoring-out the feature that is repeated most often, until no more factoring can be done.

3.8.7 Wolff

Wolff [Wolff94] suggests that good design of software may be seen as information compression, and provides an example of discovering an inheritance hierarchy from object definitions. However, Wolff does not explain the details of his algorithm and admits that the hierarchies discovered depend on the ordering of object definitions. Wolff's example does not require multiple inheritance, and no mention is made of whether multiple inheritance hierarchies may be constructed by his algorithm.

3.9 Summary

The IHI criteria are formally described in Appendix A.1 and the algorithm itself is formally described in Appendix A.2. A formal discussion of the complexity of the algorithm is presented in Appendix A.3.

The design of inheritance hierarchies is important, and difficult. The extended IHI algorithm infers an inheritance hierarchy which satisfies well justified criteria, for any set of objects. This algorithm is useful for a variety of applications, in particular for inheritance hierarchy design and restructuring [Lieberherr91, Moore95, Pun89]. Compared to other algorithms for achieving similar results, the extended IHI algorithm is simple and easy to understand and implement and produces results which satisfy better justified criteria. A simple implementation is described which has been efficient enough for use on problems of up to 500 objects.

Chapter 4

Inheritance Hierarchy Restructuring

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 to it, but are reluctant to restructure it. This is not surprising, as restructuring inheritance hierarchies is difficult and error prone. Automatic restructuring requires no programmer effort and does not risk the introduction of errors. This chapter describes Guru [Moore95], a prototype tool for automatically restructuring an inheritance hierarchy into one that contains no duplicated methods, while preserving the behaviour of programs.

The inevitability of change in software development has been known for many years [Brooks75]. Software has to evolve to meet changing requirements. This should include evolution of inheritance hierarchies, and consequent restructuring to keep them well designed [Johnson88]. Without restructuring, inheritance hierarchies can deteriorate, as shown by the example in Section 2.4. Even the core libraries of object oriented programming languages are imperfect [Cook92] and have required restructuring between releases [Meyer90]. As mentioned earlier, manual restructuring of inheritance hierarchies is difficult and error prone.

By providing a tool for automatic restructuring of inheritance hierarchies, 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 only restructure non-reflective programs. Reflective programs depend on the structure of their objects, as explained in Section 2.3.4. Changing the structure of objects involved in a reflective program can change the behaviour of those reflective programs. As Guru is intended not to change the behaviour of programs, it can only be used safely for non-reflective programs.

Guru has been designed to restructure only run-time error free programs. Incorrect programs may cause Guru to fail to work, or to produce restructured programs which do not preserve the (incorrect) behaviour of the original programs.

4.1 Introduction

The user specifies which objects are to be included in a restructuring (called the *original* hierarchy. See Section 7.1.2 for a description of a user interface for specifying the original hierarchy). These objects do not need to be related by inheritance, and do not need to form a complete inheritance hierarchy. Guru restructures the original hierarchy by discovering which of the objects in it need to have their behaviour preserved, removing their current inheritance hierarchy, and then inferring a *replacement* inheritance hierarchy using the extended IHI algorithm described in Chapter 3. The replacement hierarchy is created without affecting the original hierarchy, and the user can choose whether or not to use it to replace the original hierarchy. The inheritance hierarchy restructuring performed by Guru is called *IHR*.

An example of the effect of Guru is that the original hierarchy shown in Figure 4.1 can be restructured into the replacement hierarchy shown in Figure 4.2. If required, a different inheritance hierarchy inference algorithm could be used.

However, the extended IHI algorithm produces hierarchies which satisfy well justified criteria.

In the figures, slots with the same name are equivalent unless stated otherwise. Note that slots m1, m7 and m8 are duplicated in the original hierarchy shown in Figure 4.1, but in the restructured hierarchy no slots are duplicated. The objects o1 to o5 in the restructured hierarchy have the same behaviour as the objects o1 to o5 in the original hierarchy, and so can be used to replace them without affecting the behaviour of programs.

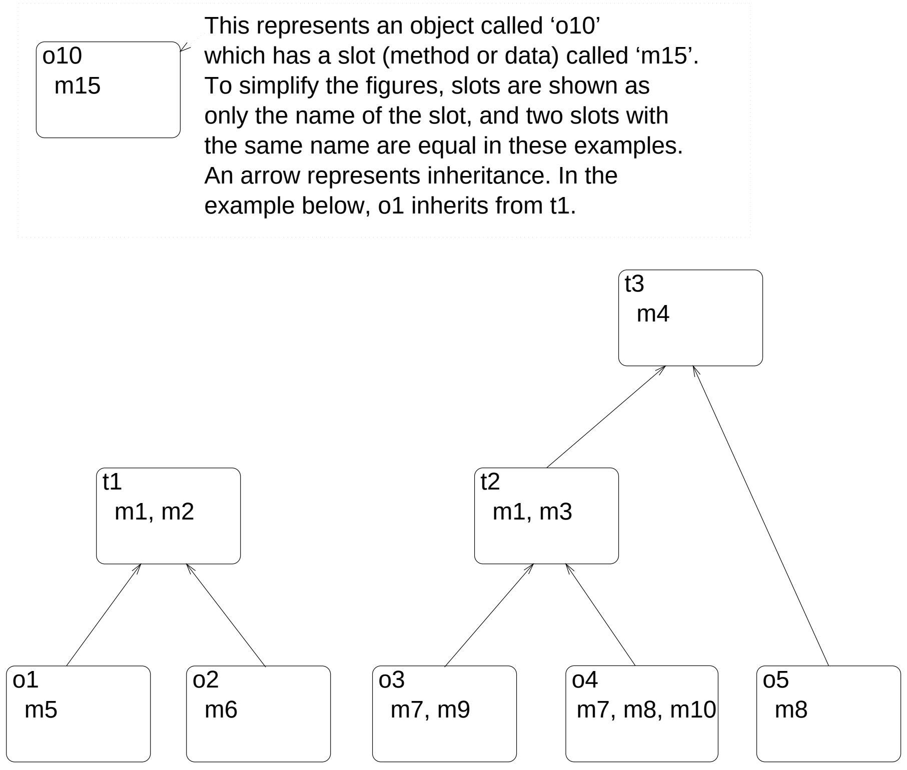

o10
m15
This represents an object called ‘o10’ which has a slot (method or data) called ‘m15’. To simplify the figures, slots are shown as only the name of the slot, and two slots with the same name are equal in these examples. An arrow represents inheritance. In the example below, o1 inherits from t1.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

Figure 4.1: Example inheritance hierarchy

The objects in the original hierarchy whose behaviour needs to be preserved will be called the *preserved* objects. Section 4.2.1 will explain how the preserved objects are identified. In the example shown in Figure 4.1, the objects o1 to o5 are the preserved objects. The objects in the restructured hierarchy which replace

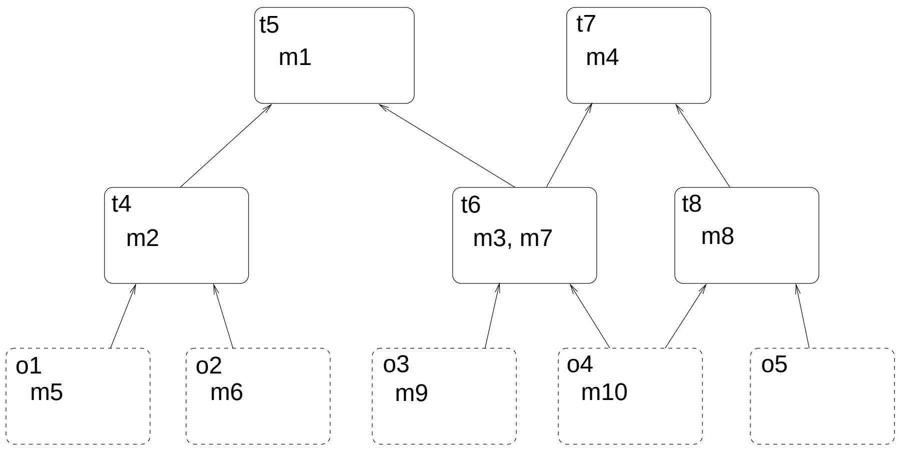


Figure 4.2: Example restructured inheritance hierarchy

preserved objects will be called *replacement* objects. Objects in the original hierarchy which are not preserved objects, and objects in the restructured hierarchy which are not replacement objects, will be called *traits objects*.

4.2 Preserving the behaviour of a system

In order for a restructuring to preserve the behaviour of the system, all of the objects in the original hierarchy 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.

4.2.1 Identifying which objects to preserve

Some simple heuristics are applied by Guru to determine which objects should have their behaviour preserved. These heuristics, described below, partly rely on the way that Self systems are structured [Ungar91], and in practice have been adequate. A more accurate way to determine which objects to preserve would be to use sophisticated type inferencing [Agesen95] to identify which objects are sent any messages; it is these objects which need to be preserved.

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 4.1 (see Figures 4.1 and 4.2). Note that the leaves of the hierarchy to be restructured must be preserved, even if they are not leaves of the complete inheritance hierarchy. This is necessary to preserve the behaviour of objects not included in the restructuring which inherit from restructured objects. Additional complications due to restructuring partial hierarchies are discussed in Section 4.3. Further heuristics, described below, are used to identify other objects which need to be preserved.

It is straightforward to allow the user to specify that certain objects should be preserved in addition to those determined automatically.

Assignable parent slots

The existence of assignable parent slots in Self requires further objects to be preserved. Objects inherit from the objects referred to by assignable parent slots in the same way that they inherit from non-assignable parent slots. In order to preserve the behaviour of an object containing an assignable parent slot, any object used as the value of an assignable parent slot needs to be preserved. In general it is impossible to predict which objects will be assigned to an assignable parent slot, as it could be any object in the system. At least the current object assigned to the slot is one possibility. In practice, the other possibilities are objects which are explicitly referred to, i.e. objects which are the values of non-parent slots. Preserving the behaviour of all leaf objects and all objects which are referred to by either assignable parent slots or by any other non-parent slots has been adequate in practice.

There is the possibility of failing to preserve an object which needs to be preserved, if an object which is not referred to by a non-parent slot (i.e. is only referred to by a parent slot) is either sent a message, or assigned to an assignable parent slot. Such situations do not normally occur in Self programs as they rely on the inheritance hierarchy structure and so are essentially reflective. Figure 4.3 illustrates a situation in which there is the possibility of failing to identify all objects which should be preserved. Object o1 defines an assignable parent slot called parent, and object o2 defines a (non-assignable) parent slot called parent. Objects o1, o2 and p1 will be identified as preserved objects. If p2 is not preserved, but in the original code is sent a message, or p2 is assigned to parent in object o1, then the behaviour of the original code will not be preserved.

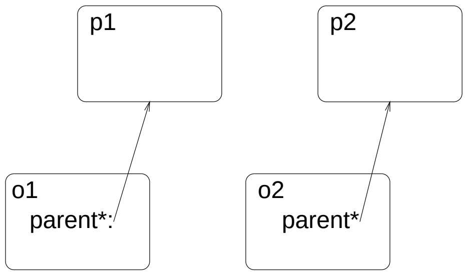


Figure 4.3: Objects referenced only by parent slots are not preserved.

Delegation

In the current version of Guru, delegation is not handled explicitly. If preserved objects are delegated to, then there is no problem with the restructuring, as this delegation will still work because the replacement object delegated to will cause the same code to be executed. However, if non-preserved objects are delegated to, then these delegations will not work, as equivalent objects may not exist in the restructured system. Non-preserved objects are those objects referred to only by non-assignable parent slots, and hence exist only for inheritance hierarchy structuring. Therefore delegating to non-preserved objects is reflective as it relies on the structure of the inheritance hierarchy, and hence should not occur in normal Self programs. However, if required, non-leaf objects which are delegated to could be manually included as preserved objects.

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 well as parent slots) as a convenient way to refer to them while developing a system. 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 be preserved, hence limiting the amount of restructuring possible and resulting in poor inheritance hierarchy structures, as discussed in Section 4.3.2. 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 [Ungar94b], as not only are they reflective but they also indicate a class-centric way of thinking, contrary to the philosophy of Self.

4.3 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. Consideration has to be given to situations in which not all the children of the specified objects are included in the restructuring, as their parents might not exist after restructuring. In such circumstances the system would not be able to replace the parent(s) of an object not included in a restructuring with a replacement object. In this situation, a child object not included in a restructuring would require the unrestructured version of its parent to exist after restructuring, in addition to the replacement object for the child's parent required by objects included in the restructuring. Consider the hierarchy shown in Figure 4.4. Figure 4.5 shows how the system would be restructured if object C is not included in the restructuring, and a simplistic approach taken. Such an approach is unsatisfactory, as it results in a considerable amount of duplication, and a worse structure, which is exactly contrary to the aims of restructuring.

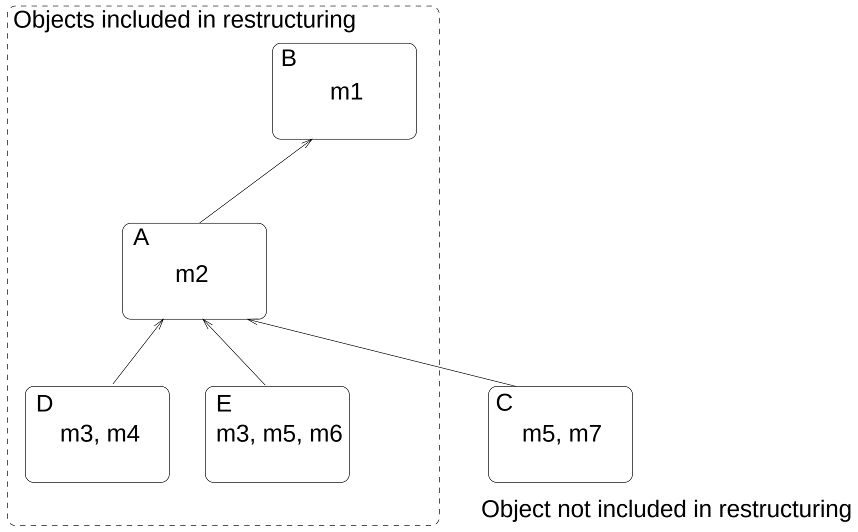

Objects included in restructuring
B
m1
A
m2
D
m3, m4
E
m3, m5, m6
C
m5, m7
Object not included in restructuring

Figure 4.4: Example where a child is not included in restructuring

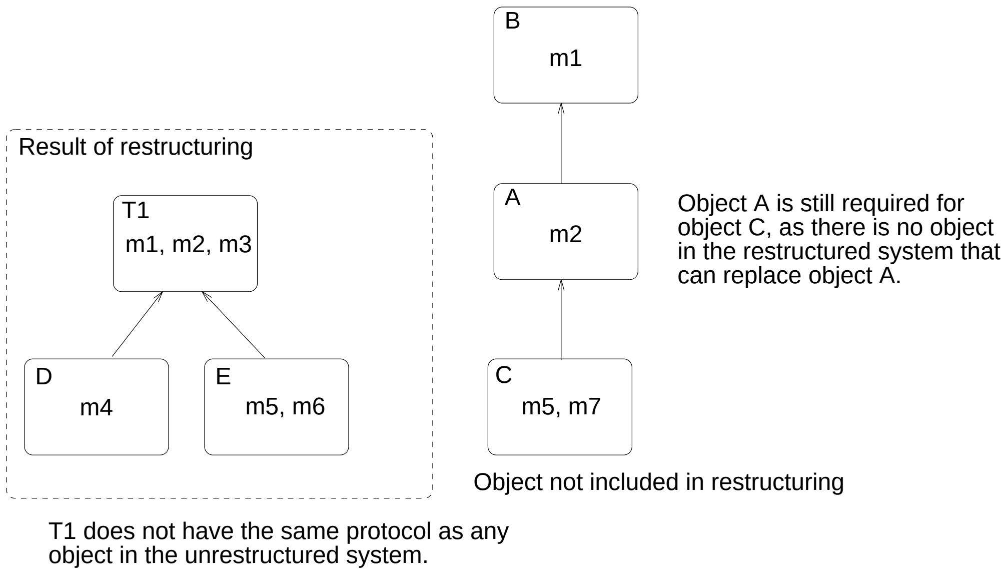

B
m1
Result of restructuring
T1
m1, m2, m3
A
m2
Object A is still required for object C, as there is no object in the restructured system that can replace object A.
D
m4
E
m5, m6
C
m5, m7
Object not included in restructuring
T1 does not have the same protocol as any object in the unrestructured system.

Figure 4.5: Result from simplistic approach

The approach taken by Guru is to include as a preserved object, any object which has children not included in the restructuring. The result of restructuring the hierarchy in Figure 4.4 (with object A as a preserved object) is shown in Figure 4.6.

An alternative approach would be for the user to specify for the restructuring only those objects for which all their children could be included (i.e. only objects D and E), resulting in the hierarchy shown in Figure 4.7. However, this would not allow as much restructuring as possible.

The best results would be obtained if all the children of the user's original choice of objects to be restructured can be included (i.e. objects A, B, C, D and E), as shown in Figure 4.8. However, this may not always be possible; for example, there may be too many children of an object to include all of them, or the programmer may require some of them to remain unrestructured for some reason (for example if they contain reflective code, so it is unsafe to restructure them).

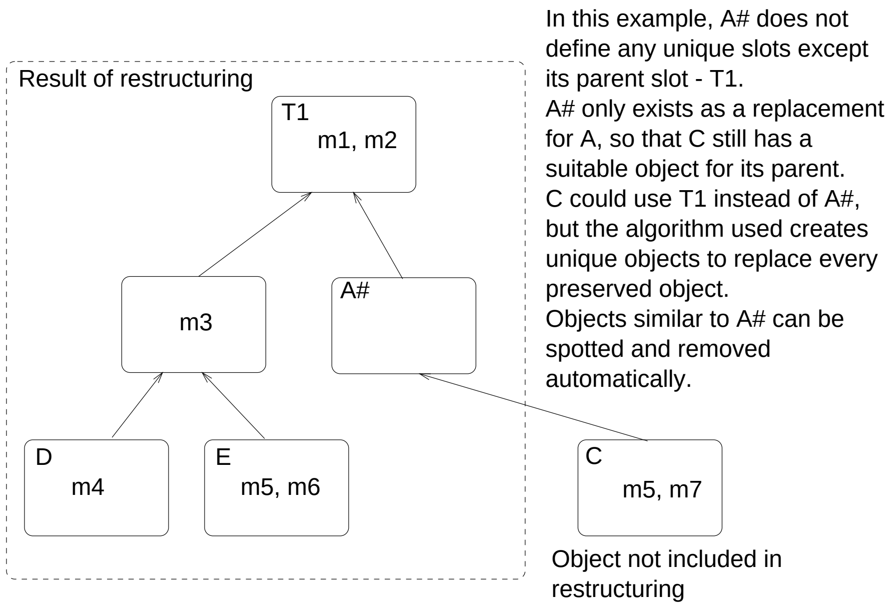

In this example, A# does not define any unique slots except its parent slot - T1.
A# only exists as a replacement for A, so that C still has a suitable object for its parent.
C could use T1 instead of A#, but the algorithm used creates unique objects to replace every preserved object.
Objects similar to A# can be spotted and removed automatically.
Result of restructuring
T1
m1, m2
m3
A#
D
m4
E
m5, m6
C
m5, m7
Object not included in restructuring

Figure 4.6: Result when A is included as a preserved object

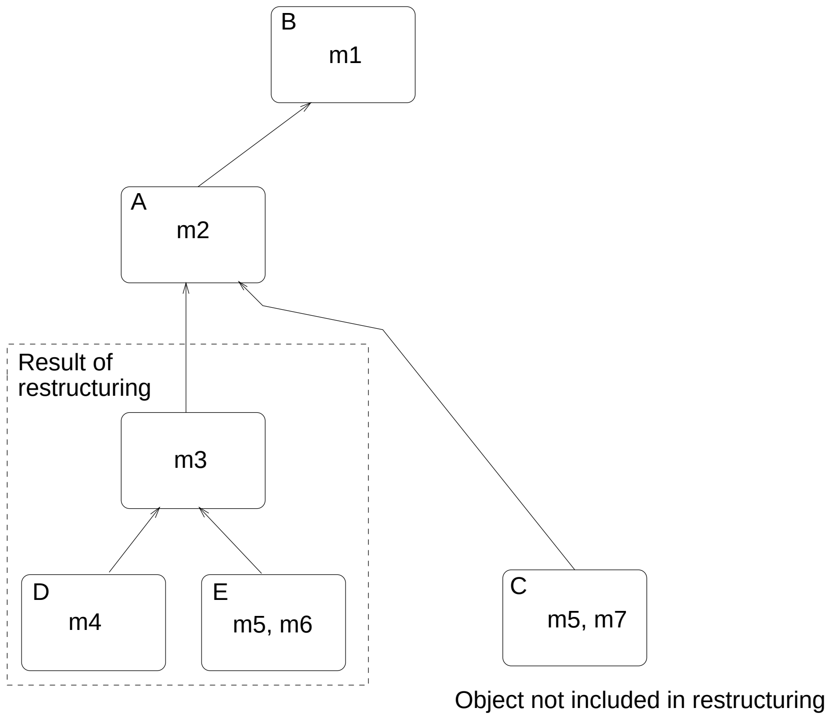


Figure 4.7: Result when A and B are not included in the restructuring

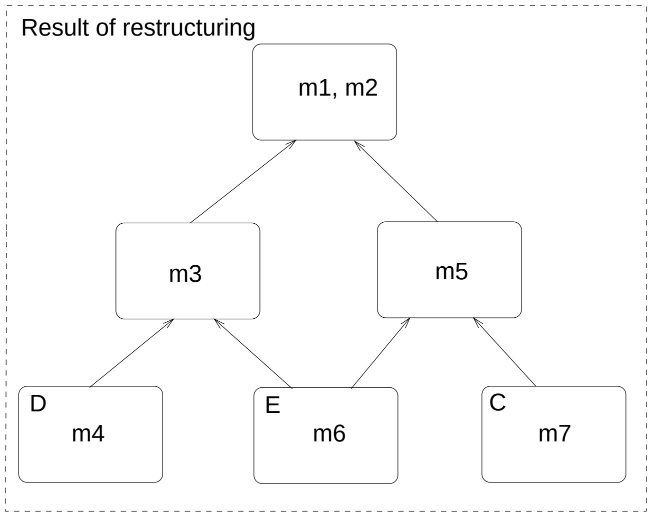


Figure 4.8: Result when A, B and C are included in restructuring

4.3.1 Introduction of ambiguities

Restructuring only part of a system can result in ambiguous message sends being introduced, as shown in Figure 4.9. Any ambiguities introduced are spotted and removed by automatically adding disambiguating methods.

Ambiguities can be introduced in a slightly more subtle way. Consider the hierarchy resulting from a restructuring shown in Figure 4.10. Object A was not included in the restructuring, and defines a method m which is overridden by m in object C. Due to the way message lookup is defined in Self, the message m is now ambiguous for object B, because two implementations are found by method lookup; one in A and one in C, even though the definition of m in C might be expected to override the definition in A. The language Cecil [Chambers93] includes an automatic disambiguating rule for such situations, but this was not included in Self due to the aim of making lookup semantics as simple as possible.

Note that ambiguities can only be introduced due to overriding of methods from objects not included in a restructuring. If a complete inheritance hierarchy is restructured then there will be no ambiguities introduced. Ambiguities can only exist if the restructured inheritance hierarchy includes multiple inheritance.

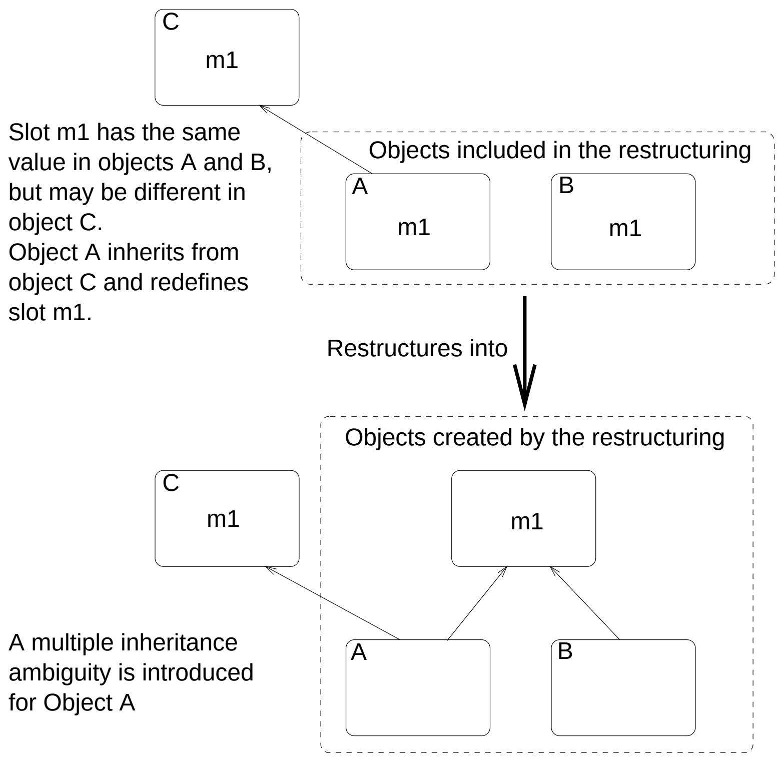

C
m1
Slot m1 has the same value in objects A and B, but may be different in object C.
Object A inherits from object C and redefines slot m1.
Objects included in the restructuring
A
m1
B
m1
Restructures into
Objects created by the restructuring
C
m1
m1
A
B
A multiple inheritance ambiguity is introduced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

Figure 4.9: Ambiguities introduced by restructuring.

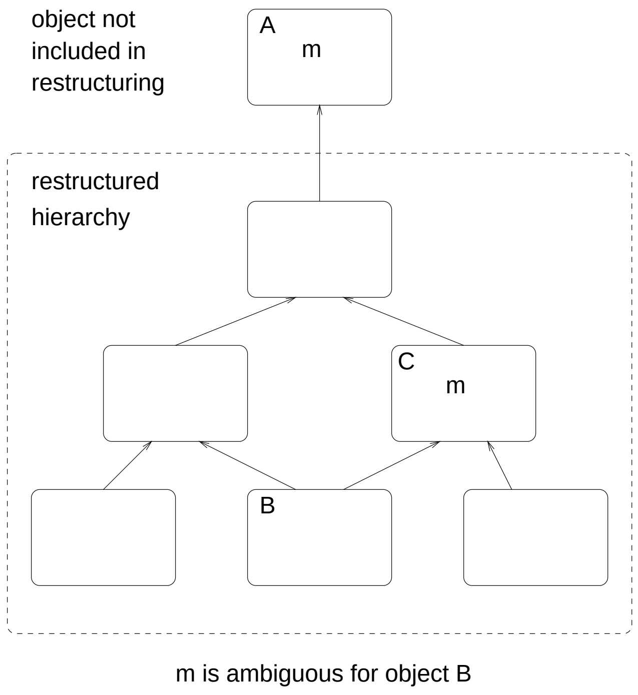

object not included in restructuring
A
m
restructured hierarchy
C
m
B
m is ambiguous for object B

Figure 4.10: Ambiguities introduced by restructuring.

4.3.2 Consequences of preserving traits objects

Guru preserves traits objects which are included in a restructuring, if all their children are not included, for the reasons described earlier. This approach can result in badly structured hierarchies in some circumstances. Consider the hierarchy shown in Figure 4.11.

If these objects are restructured, preserving the traits object 1 (as it has a child not included in the restructuring), the resulting hierarchy is shown in Figure 4.12.

If all the children of traits object 1 contain a method p (for example, if they are all copies of object 2) then a better restructured hierarchy would be that shown in Figure 4.13. However, in this hierarchy, the behaviour of traits object 1 is not preserved as it now understands the message p. If objects inherit from traits object 1 which do not contain a method p then their behaviour will also be changed in the same way. This change in the behaviour of objects would normally not be a problem, as code which relies on an object *not* understanding a message

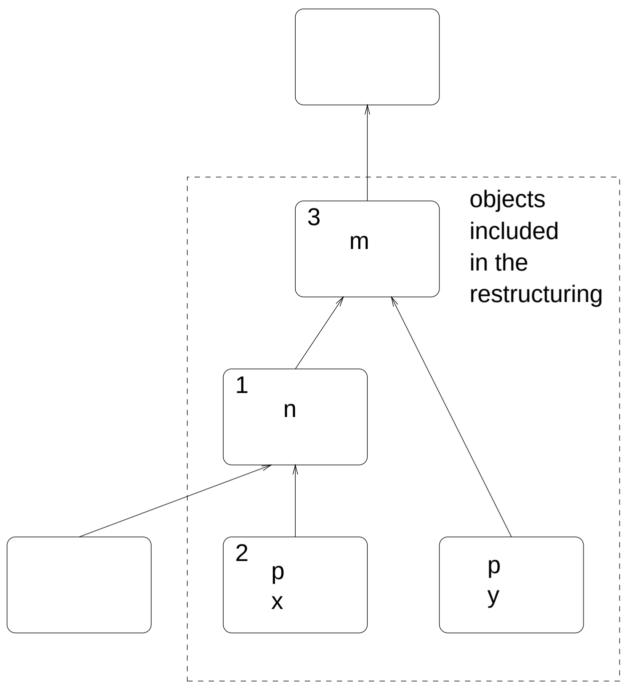


Figure 4.11: Example hierarchy

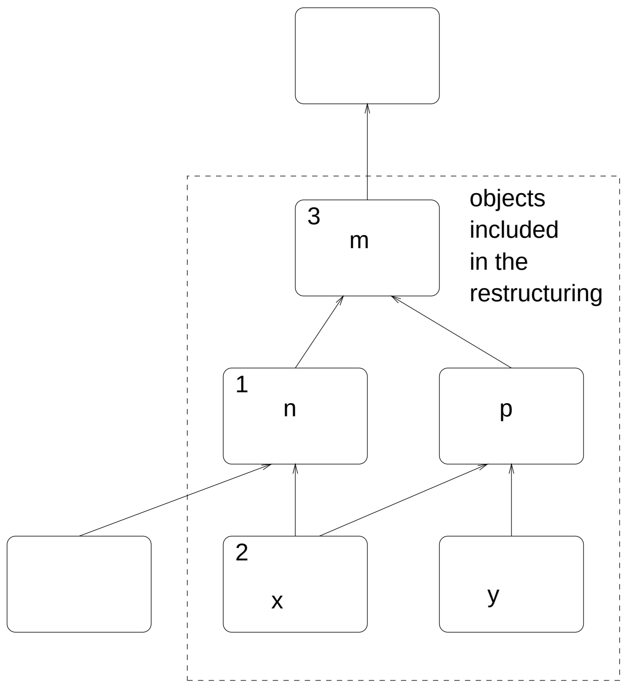


Figure 4.12: Hierarchy preserving a traits object

is unusual. However, if traits object 3 defines an inheritance link to an object which defines a different implementation of p to the one which it now overrides, then this is a more serious change in the behaviour of the system. In this case, objects which do not define p themselves but inherited the version of p defined in an object above traits object 3 in the hierarchy, will now inherit the version in traits object 3, and so behave differently in response to the message p.

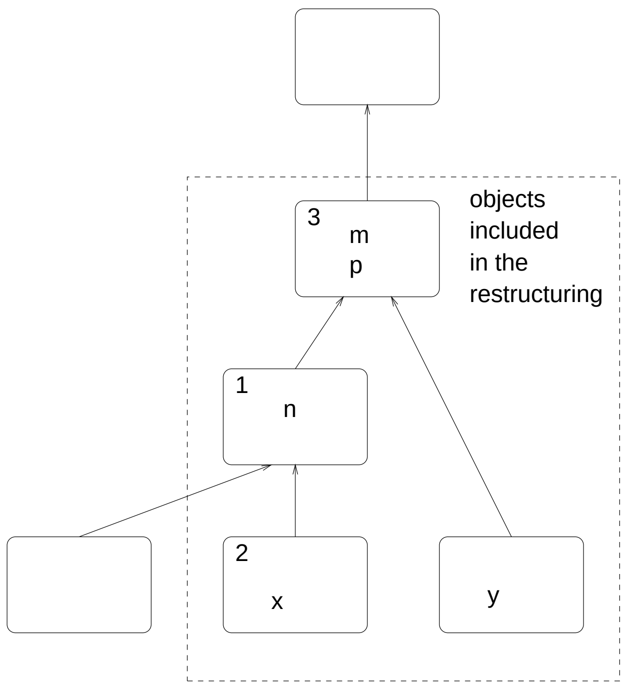


Figure 4.13: Hierarchy not preserving a traits object

Using Guru automatically, it will choose the *safest* option, that is, it will preserve any traits objects which have children not included in the restructuring. In this example, this results in the hierarchy shown in Figure 4.12. To allow such a traits object to *not* be preserved, in order to produce a better hierarchy such as that shown in Figure 4.13 in this example, requires user intervention.

4.3.3 'Cracking', and including copies of leaf objects

Consider restructuring the hierarchy shown in Figure 4.14, which includes only a prototype object and no copies of it, and a parent of the prototype object which has no children other than the prototype. The hierarchy of Figure 4.14 will be restructured into the hierarchy of Figure 4.15.

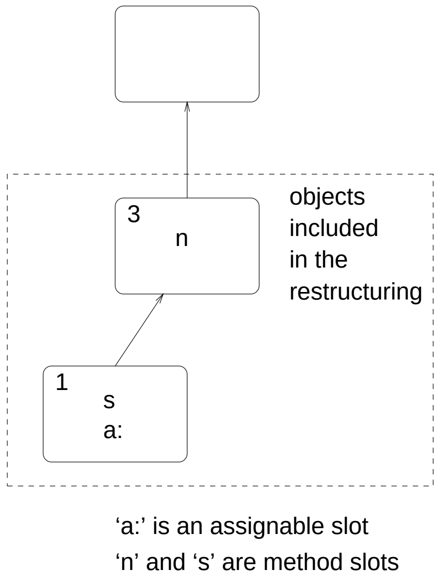

3
n
objects included in the restructuring
1
s
a:
'a:' is an assignable slot
'n' and 's' are method slots

Figure 4.14: Example hierarchy

The prototype object 1 now defines method slots, which may not be ideal, as instead of sharing these methods between copies of the prototype, each copy of the prototype will have its own copy of these methods. Therefore, there may be consistency problems if these methods are altered. One possible solution would be to make these methods into *copied-down slots* (see Section 2.3). The preferred solution may be to have the methods in a traits object, which the prototype object inherits from. A simple restructuring has been created for this, called 'cracking'. Applied to any object, it creates a new parent object and moves all the non-assignable slots there. Objects are left unaltered by this restructuring if they have either no assignable slots, or no non-assignable slots. Figure 4.16 shows how the hierarchy appears having applied 'cracking' to prototype object 1.

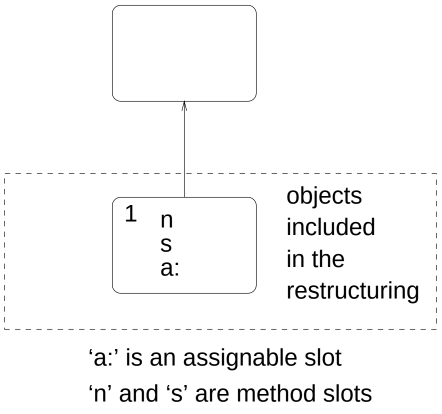

1
n
s
a:
objects included in the restructuring
'a:' is an assignable slot
'n' and 's' are method slots

Figure 4.15: Restructured example hierarchy

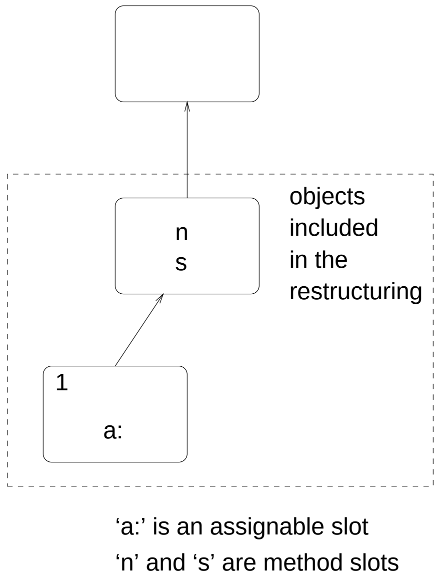

n
s
objects included in the restructuring
1
a:
'a:' is an assignable slot
'n' and 's' are method slots

Figure 4.16: Restructured example hierarchy, after 'cracking' object 1

If a copy of the prototype object exists in the system, but is not included in the restructuring, then traits object 3 will have to be preserved, and the result of restructuring will be the same as the original hierarchy. The result of applying 'cracking' to this hierarchy is shown in Figure 4.17. If the only children of traits object 3 that exist or will exist are copies of the prototype object 1, then this resulting hierarchy is not ideal as methods are unnecessarily split between two traits objects. A better hierarchy would be that of Figure 4.16.

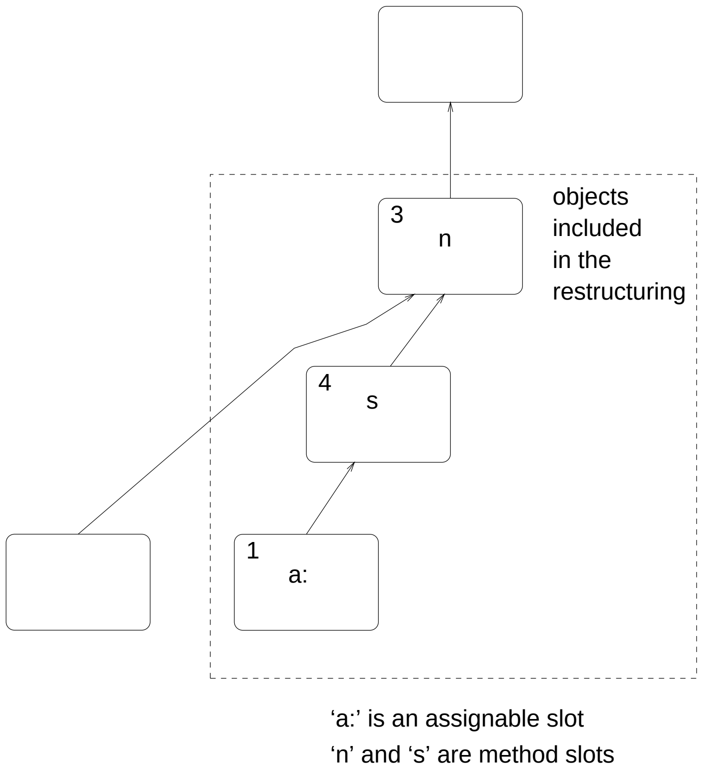

3
n
objects included in the restructuring
4
s
1
a:
'a:' is an assignable slot
'n' and 's' are method slots

Figure 4.17: Example hierarchy after 'cracking'

Consider the results of restructuring the same objects as in Figure 4.14, except the only copy of the prototype object is included in the restructuring, as shown in Figure 4.18. (Therefore, all the children of traits object 3 are included in the restructuring). Restructuring this hierarchy will result in the hierarchy shown in Figure 4.19. Note that assignable slots defined in different objects are not equal, even if their name and current value is equal, as described in Section 4.4.1.

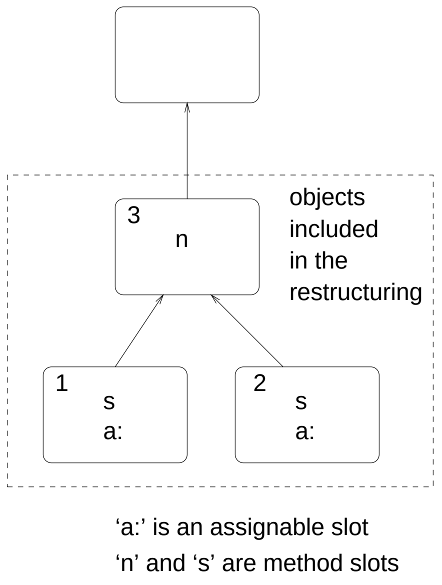


Figure 4.18: Example hierarchy, including copy of prototype object

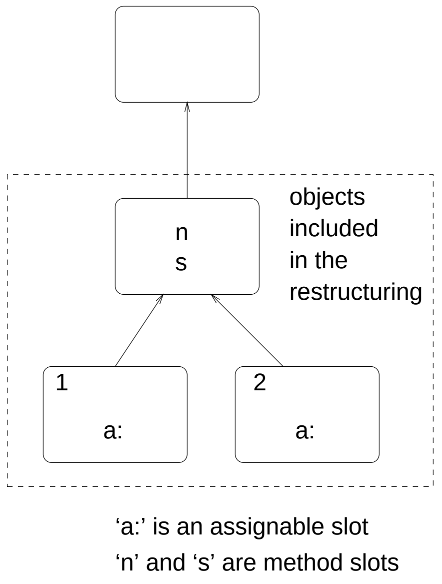


Figure 4.19: Restructured hierarchy, including copy of prototype object

As this appears to be the best hierarchy for these objects, users may think that including a copy of the prototype object will result in good hierarchies. However, this is not the case if only some of the copies are included, as the parent of the prototype will then have children not included in the restructuring. For example, if traits object 3 in Figure 4.18 has children not included in the restructuring, then the result of restructuring is shown in Figure 4.20.

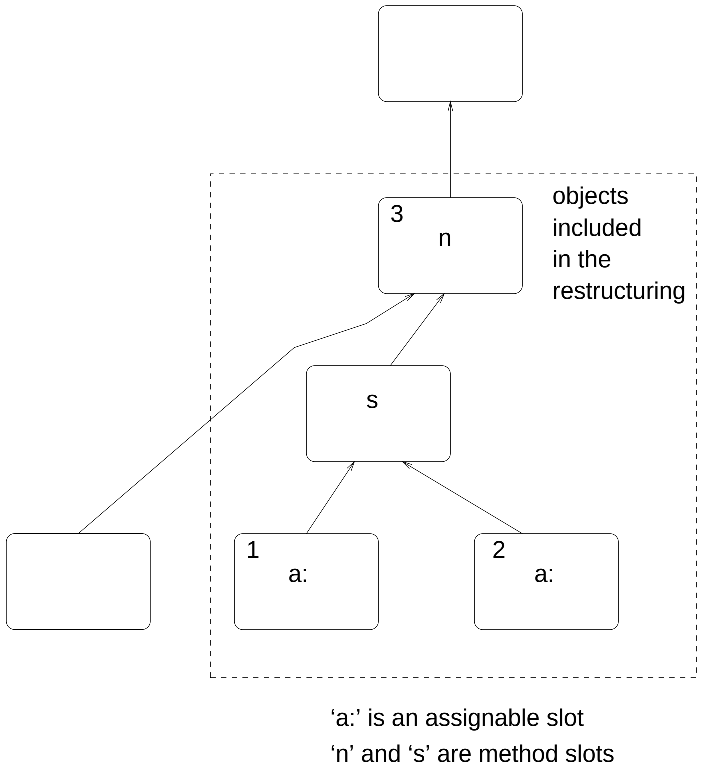


Figure 4.20: Restructured hierarchy, including one (but not all) copies of prototype

4.4 Removing existing hierarchies

This section explains what Guru needs to do in order to create a restructured inheritance hierarchy, after the objects to be restructured have been specified, and the preserved objects discovered. The approach used by Guru is to remove the existing inheritance hierarchy by 'flattening' the preserved objects, to discover the object definitions to be used by the extended IHI algorithm to create a replacement inheritance hierarchy. For all preserved objects, non-parent slots inherited from objects included in the restructuring are copied into each corresponding replacement object. For example, for the objects shown earlier in Figure 4.1, the flattened replacement objects are those shown in Figure 4.21.

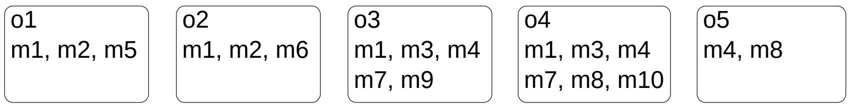


Figure 4.21: Example problem flattened replacement objects

Non-assignable parent slots of objects included in a restructuring are not copied into flattened objects. The aim of the restructuring is to discover an improved replacement hierarchy, so inheritance information about the original hierarchy is neither required nor appropriate.

Slots inherited from assignable parent slots are ignored by the flattening, as these slots change dynamically depending on the value of the assignable parent slot, and therefore cannot be statically copied into preserved objects. However, assignable parent slots are included in flattened objects. Objects which may be the values of assignable parent slots will be preserved (see Section 4.2.1), so preserved objects which include assignable parent slots will behave the same after restructuring, as they will still include the same assignable parent slot(s) and the possible objects referenced by assignable parent slot(s) will be preserved.

Non-assignable parent slots which reference objects not included in the restructuring need to be included in flattened objects, so that the same slots are inherited by preserved objects from objects outside the restructuring. This point is further discussed in the following section.

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 slots which are unused because they are always overridden will be removed, and the inheritance hierarchy inferred by the IHI algorithm will have no overridden slots, thus simplifying the resulting system. Having many slots overridden, or slots overridden many times, is often an indication of poor inheritance hierarchy design [Johnson88].

Flattening allows the removal of cancellation of inheritance from restructured hierarchies. Cancellation of inheritance is achieved by using slots defined as shouldNotImplement. Such slots are simply removed from flattened objects; hence, the restructured hierarchies will not contain any shouldNotImplement slots, and will also not inherit any of the slots which were overridden by those slots.

Slots which override an inherited slot, in an object outside the collection of objects to be restructured, with an equivalent definition ('overrides unnecessarily' in Section 7.1) are also removed from flattened objects.

A disadvantage of flattening is the extra performance cost which it entails compared to restructuring the inheritance hierarchy 'in place', but this is more than compensated by its benefits.

4.4.1 Equivalence of slots

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. One case which Guru does handle is where two methods are the same except for the names of arguments. The names of arguments are ignored, only their positions are important. Methods which are the same except for the position of their arguments are not treated as equal by Guru, because the messages sends resulting in the execution of methods would have to be modified in order to merge such methods.

There is an important subtlety to consider for comparison of methods containing resends. Two methods can have exactly the same source code without being equivalent if they contain resends. Consider the two methods called m in objects o1 and o2, shown in Figure 4.22. These two methods have exactly the same source code, m = (resend.n), but are *not* equivalent. They *cannot* be replaced by a single method, shared between the two objects o1 and o2. This is because resends determine which method to execute independent of the receiver (see Section 2.3.1). Section 4.4.4 describes how the removal of resends from methods allows method comparison to avoid this subtlety.

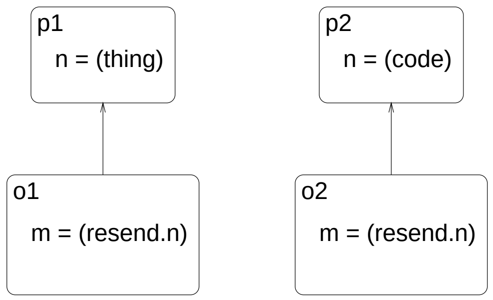


Figure 4.22: Example hierarchy showing non-equality of methods containing resends.

Assignable slots (including assignable parent slots) are only equivalent if they are identical (that is, have the same identity; they are equivalent if they are the same slot). Two different assignable slots can have the same name and the same current value, but are not equivalent. This is so that assignable slots are shared by exactly the same set of objects after restructuring as before, as is necessary to fully preserve the behaviour of a system. In Figure 4.23, the two slots labelled 'a:' are assignable slots with the same name and same (current) value. They must be defined by the same objects after restructuring. The behaviour of the system will not be preserved if objects o1 and o2 share the same slot 'a:' after restructuring. Similarly, in Figure 4.24, the assignable slot labelled 'a:' is shared by objects o1 and o2, and this must be the case after restructuring. If objects o1 and o2 define their own assignable slots ‘a:’ after restructuring then the behaviour of the system will not be preserved.

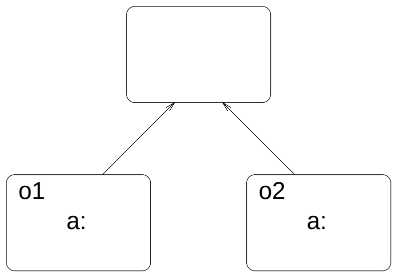


Figure 4.23: Example hierarchy showing non-equality of assignable slots.

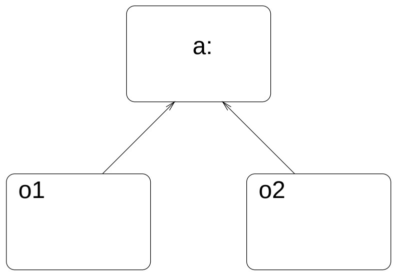


Figure 4.24: Example hierarchy showing equality of assignable slots.

Non-assignable non-method slots (including parent slots for objects not included in the restructuring, as explained in the previous section) are treated as equivalent if the name and the current value is the same. Therefore, the objects shown in Figure 4.25 are restructured into the objects shown in Figure 4.26. This may not preserve behaviour in all situations. If a non-assignable slot’s value is a mutable object (that is, has mutable state, such as the object referred to by slot d in objects o1 and o2 of Figure 4.25), then the mutable object should be shared by exactly the same set of objects in order to preserve the behaviour of the system (similar to when a slot is assignable). However, such programming style is very unusual, and this potential problem has not necessitated a simple correction to the system, which would be to check whether the object assigned to a non-assignable slot is mutable. Note that it is *possible* to change the reference of a non-assignable slot reflectively, but as mentioned earlier, the system can only restructure non-reflective programs.

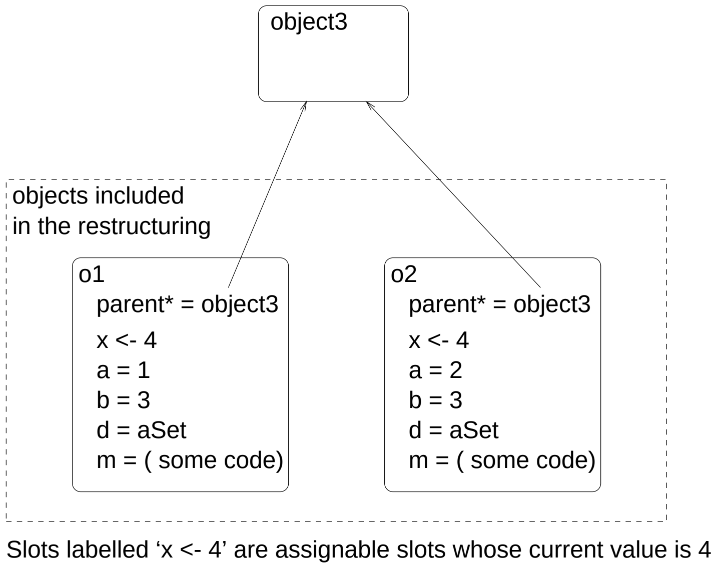


Figure 4.25: Example hierarchy for showing equality of non-assignable slots.

Note that if two or more method slots are equivalent, one of those slots will be chosen to be shared, replacing the others. Methods can be equivalent but have different layout, commenting, and naming of arguments. Guru currently choses one of the methods at random, but a more sophisticated approach would be to chose the ‘best’ of the equivalent methods based on those details which are different between the methods.

4.4.2 Removal of ‘place holders’

Sometimes methods are included which are intended to always be overridden, in the style of ‘something = (childMustImplement)’ (or subclassResponsibility in Smalltalk). Such methods, often called *place holders*, are used as a form of program documentation. Their purpose is to indicate that these methods need to

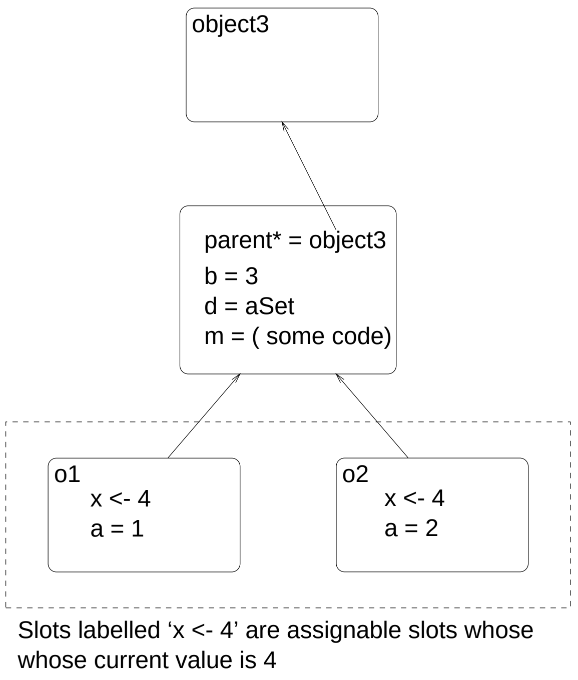


Figure 4.26: Restructured hierarchy showing equality of non-assignable slots.

be implemented (overriding the childMustImplement definition) in order to reuse other methods defined in the object containing the place holder methods.

If place holder methods are always overridden by the preserved objects of a restructuring (as they should be), then they will all be removed by flattening. This could remove valuable documentation. However, if such methods are included, they may often be redundant, and can be erroneous. Instead of relying on a programmer to include such methods, they can be discovered automatically, and reasonably accurately using the following algorithm: To find the methods which need to be defined in order to inherit from an object O, for all methods in O and all the methods it inherits create a set of all messages sent to implicit self (ignoring those messages which are arguments or local slots). The result is given by taking away these messages from the full protocol of O (i.e. all the messages it understands including those inherited).

The algorithm described has been implemented in Self (its user interface is described in Section 7.1.2), and applied to objects in the Self standard image. For example, applied to `traits point` this implementation indicates that '`x`' and '`y`' need to be defined by objects which inherit from it. There are some improvements that could be made to this; i.e. messages sent to explicit `self` and messages sent to the result of message sends that return `self` should be included, but the algorithm described is accurate enough in practice.

If any place holder methods exist in flattened objects, then it indicates that they are erroneous as they should have been overridden. In his investigation of the structure of a Smalltalk class hierarchy, Cook [Cook92] removed such methods automatically, and all methods which rely on these methods manually. It would be straightforward to implement the automatic removal of these methods from flattened objects for Guru, but this has not yet been found to be necessary in practice.

4.4.3 Removal of cyclical inheritance

Cyclical inheritance is removed by 'flattening', but does cause a small problem. The 'leaf' nodes have to be chosen in some way if they do not already exist, as shown in Figure 4.27. However this does not happen in practice, as at least one of these objects will be referred to by a non-parent slot, and hence be chosen in this way.

The behaviour of an object in a cyclical inheritance hierarchy is unaffected by flattening. Such an object will understand (and not understand) the same messages and respond to them in the same way. A naive approach to method lookup in the presence of cyclical inheritance could result in infinite lookup, but this is avoided in Self. In Self, method lookup does not consider the same inheritance link more than once. Therefore, an object with cyclical inheritance will fail in the same way as its flattened equivalent, when sent messages it does not understand.

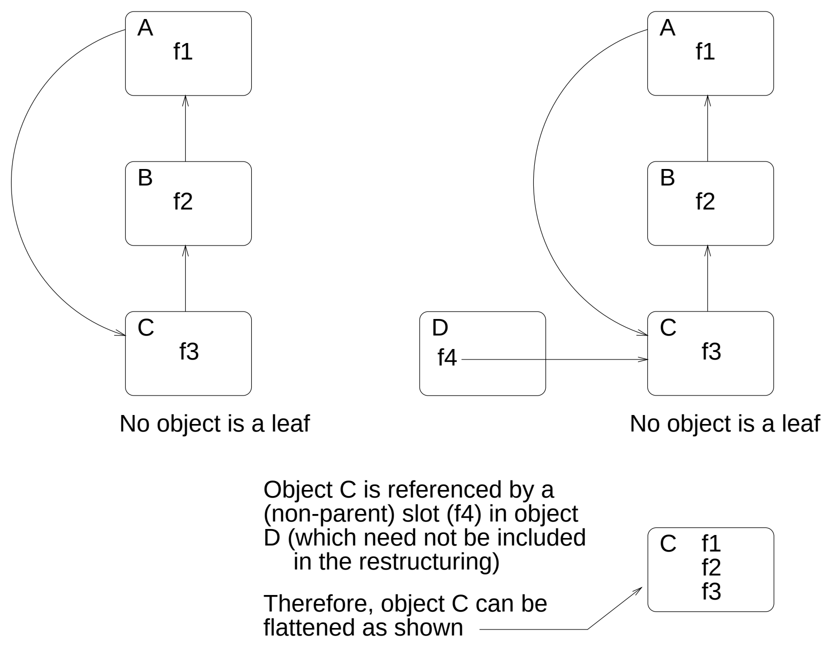

A
f1
B
f2
C
f3
No object is a leaf
A
f1
B
f2
D
f4
C
f3
No object is a leaf
Object C is referenced by a (non-parent) slot (f4) in object D (which need not be included in the restructuring)
Therefore, object C can be flattened as shown
C f1 f2 f3

Figure 4.27: Cyclical inheritance problem

4.4.4 Removal of resends

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 (see Section 2.3.1). Resends in methods would be meaningless if nothing were 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 defined within the same flattened object. A message send replacing a resend is called a *resend replacement send* (RR send) and the uniquely named method which it invokes is called a *resend replacement method* (RR method). The method invoked by the resend is called the *resend invoked method* (RI method). The name of the RR method is generated by concatenating an identifier representing the object where the RI method is defined (unique for each object) and the name of the RI method.

The implementation of the RR method is created as a copy of the implementation of the RI method. (The behaviour resulting from the resend is preserved, as self is the same object for the RR method as for the RI method.) RR methods are included in the appropriate flattened objects. Each RR method will be shared by IHR amongst all the objects which include an RR send in one of their methods which will execute the RR method. Resends in RR methods are also flattened by Guru. Figure 4.28 shows an example of how resends are flattened.

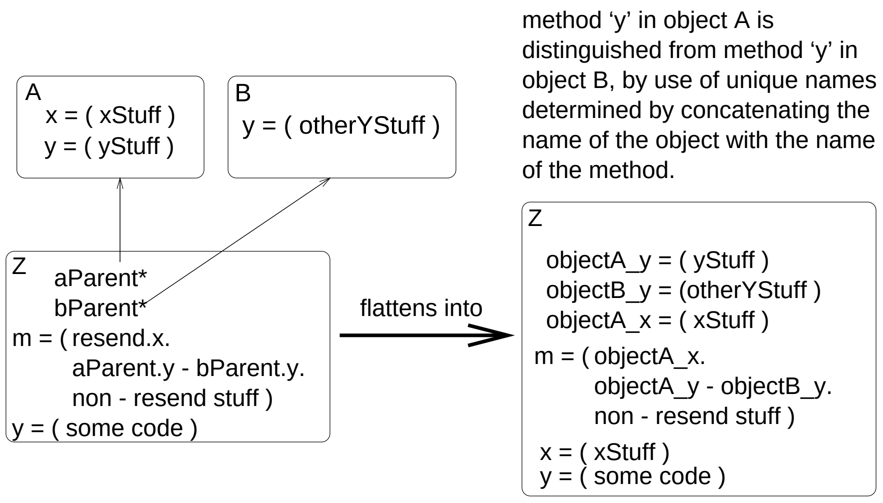


Figure 4.28: How resends are 'flattened'

The approach to removal of resends described in this section allows for comparison of methods to take into account the fact that different resends which have the same source code have different meanings, as explained in Section 4.4.1. The flattening with removal of resends will convert the objects o1 and o2 shown earlier in Figure 4.22 into the objects in Figure 4.29 (including only objects o1 and o2 in the restructuring). Now the two methods called m, in objects o1 and o2, are different, which is what is required. If the two objects were as shown in Figure 4.30, the removal of resends would result in the objects shown in Figure 4.31, which would be restructured into the hierarchy shown in Figure 4.32. In this case, the two methods called 'm' are equivalent and are shared between objects o1 and o2 by the restructuring, which is appropriate. Section 4.7 describes how these objects can be improved by replacing the RR send in method m with a resend.

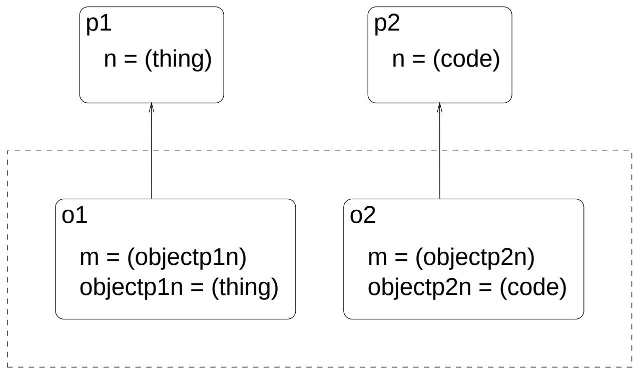


Figure 4.29: Removing resends complements method comparison.

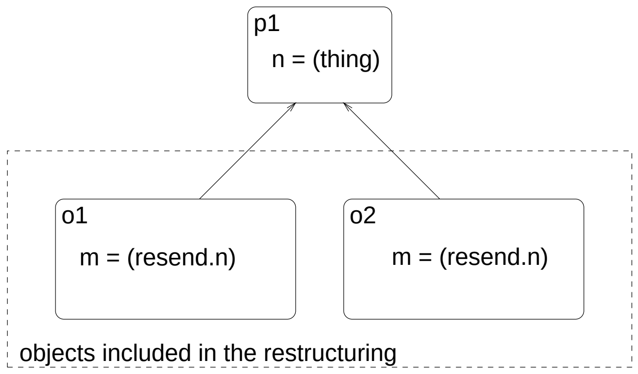


Figure 4.30: Example objects.

If an RI method is not overridden by all the children of the object where it is defined, then duplication of the implementation of the RI method will exist in the restructured hierarchy. The RI method and an RR method will both exist, with the same implementation but different names. This introduced duplication is contrary to the aims of IHR, and Section 4.7 describes how this duplication can be removed by reintroducing resends to replace RR sends.

In the case of resends to assignable parents, the resend is left as it is, because the restructuring process will guarantee that the assignable parent slot will still exist for the same objects as before the restructuring, and that the assignable

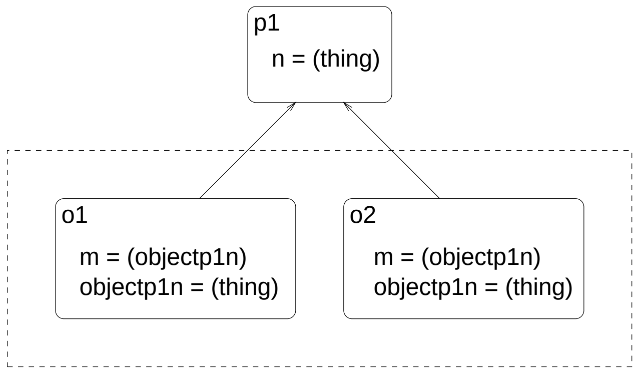


Figure 4.31: Example objects after flattening.

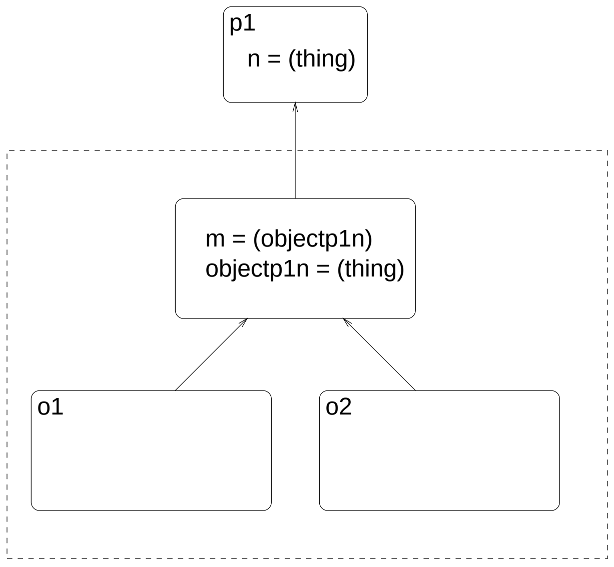


Figure 4.32: Example objects after restructuring.

parent slot will refer to an object which has the same behaviour as before the restructuring.

The system should check that the names of RR methods are not already used by any methods anywhere in the system to ensure that such methods are not executed inappropriately. This check is very simple to implement in Self. Similarly, the system should check that there are no message sends anywhere in the system which are the same as any RR send. This check is straightforward in most cases, but it is *possible* to write code which sends messages that cannot be determined until runtime (using computed selectors, described in Section 2.3). Such a send could execute an RR method inappropriately. More subtlely, such a send could have existed before restructuring with the result that it was not understood, but after restructuring it would be understood, thus altering the behaviour of the system. Programs which rely on an object *not* understanding a certain message are unusual, so this potential problem is very unlikely.

The approach to handling resends was taken from [Moore94], where it was used for a different purpose in a translator from Smalltalk [Goldberg90] into CLOS [Keene89]. Calls of uniquely named functions (rather than methods) were used to translate the equivalent of resends in Smalltalk (‘`super`’), because CLOS does not have a direct equivalent. Furthermore, this approach allowed static binding of translated resends in order to (marginally) improve the performance of translated code.

4.5 Applying an inheritance hierarchy inference algorithm

An inheritance hierarchy inference algorithm can now be applied to the flattened objects. In Guru, the extended IHI algorithm described in Chapter 3 is applied (using the reintroduction of overriding described in Section 3.5.2). The extended IHI algorithm simply requires the object definitions, and that the (name and implementation) equality of features is defined, and it can discover a restructured hierarchy which has no duplicated features and the structure reflects the object definitions. The extended IHI algorithm could be replaced by another algorithm if required.

Having discovered the replacement hierarchy, the actual replacement Self objects are built. The replacement inheritance hierarchy can be further improved through more restructuring, as described in the following sections.

4.6 Replacing original objects with restructured objects

The newly created replacement objects can be used to replace the original objects. The structure of the replacement inheritance hierarchy, and details of where methods are defined, or which objects they are inherited from, do not matter as long as replacement objects have the same behaviour as the preserved objects they replace. Each preserved object is modified, using 'mirrors' (see Section 2.3.4), so that it defines the same slots as its replacement object. Only preserved objects need to be modified, as all non-preserved objects must be referred to by only parent slots (otherwise they would have been preserved), and the newly created traits objects that are needed will be referred to by the preserved objects (either directly or transitively). Once the preserved objects have been replaced, the original non-preserved objects will be garbage collected automatically as they are not referenced by any (non-garbage) object.

A detail which has not been implemented is the correct replacement of original objects that have a non-standard mirror, such as mirrors vector (see Section 2.3.7). A replacement object should be created with the same sort of mirror as the original object. This minor implementation detail could be fixed in a future version of Guru, but this has not been found necessary in the current prototype version of Guru.

If a programmer does not want to replace all the objects in a restructuring, then this can cause redundancies to be introduced, similar to the situation when all the children of a traits object in a restructuring are not included, and a simplistic approach is taken (see Figure 4.5 in Section 4.3). If an object A is replaced, but a child object B of one of A's parent objects is not replaced, then the parent objects of both A and B will have to exist in order to preserve the behaviour of the system. For example, the objects in Figure 4.33 would be restructured into the objects in Figure 4.34. If only object A was replaced, then the parent object of B could not be replaced as otherwise B would not have a suitable object to inherit from. Therefore, the hierarchy shown in Figure 4.35 would result. In order to avoid introducing redundancies, all children of all parent objects of objects replaced should also be replaced. In other words, only complete hierarchies should be replaced. The current implementation of Guru will only perform the automatic replacement of preserved objects with replacement objects for *all* of the preserved objects in a restructuring.

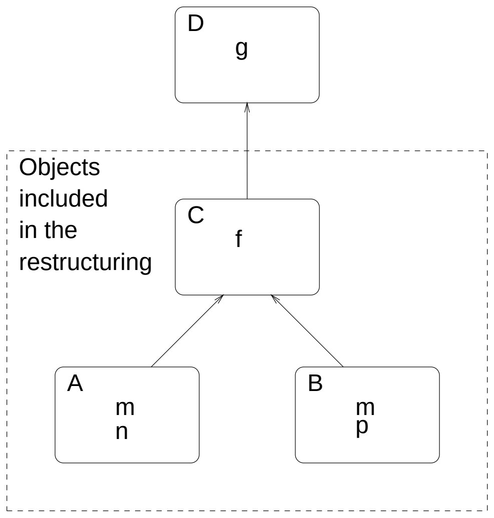


Figure 4.33: Original objects.

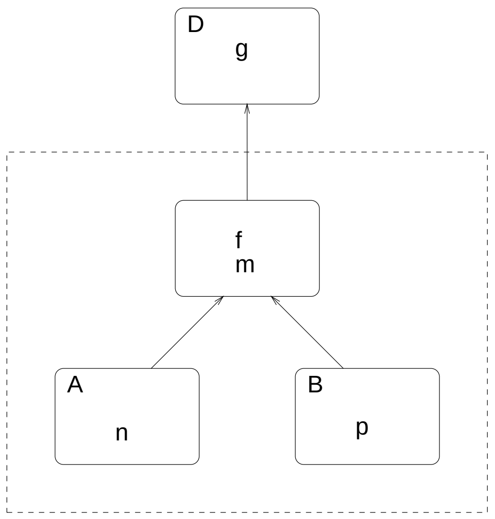


Figure 4.34: Replacement objects.

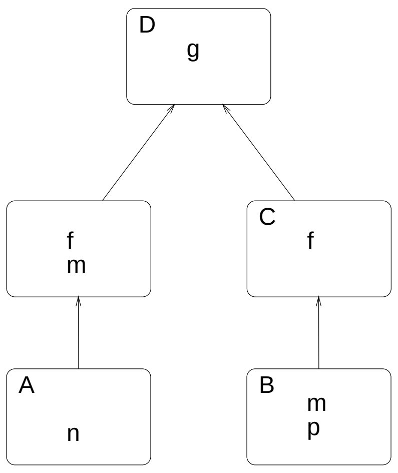


Figure 4.35: Objects if only A is replaced.

4.7 Reintroducing resends

Section 4.4.4 describes how the removal of resends may result in the introduction of methods with duplicated implementations. In order to eliminate this duplication, and remove the RR methods, the system attempts to convert RR sends back into resends.

The reintroduction of resends works by examining all RR sends (which are identified by their unique names). As described in Section 4.4.4, the system should not contain any methods which have the same name as any RR method (which will be unique). The system (statically) determines the method which will be executed (in the restructured inheritance hierarchy) by a resend of the name of the original RI method called before the resend was replaced by the RR send. If the method called by such a resend has the same implementation as the RR method called (which is also statically determinable because it has a unique name and will definitely be inherited), then the RR send is replaced by a resend of the original RI method.

The current implementation of the reintroduction of resends only considers reintroducing undirected resends (see Section 2.3.1). In some cases, it might be preferable to reintroduce directed resends; for example, if the original resend (replaced by an RR send) was originally directed, or if a method containing an RR send is in an object defining multiple inheritance, particularly if the RI method replaced by the RR method is now ambiguous.

Only RR sends, rather than message sends in general, can be replaced by resends, because RR sends can be statically bound, as they invoke uniquely named methods. That is, the method invoked by an RR send is statically determinable as being the RR method which is the equivalent of the original RI method. A normal message send is not statically determinable because of overriding, which may exist in restructured hierarchies, because of methods defined in objects not included in the restructuring, and by the reintroduction of overriding described in Section 3.5.

Having replaced as many RR sends as possible, any RR methods which are no longer required, because the resends they existed to replace have all be converted back into resends, are removed.

This simple technique has been successful in practice, but there are some circumstances where it fails to reintroduce resends. Consider the hierarchy shown in Figure 4.36. This will be restructured into the one shown in Figure 4.37. In this example, the resend removed by the RR method ‘object1m’ cannot be reintroduced. Furthermore, the hierarchy is now simply two separate objects, which may not be desirable. However, consider the hierarchy shown in Figure 4.38, which will be will be restructured as shown in Figure 4.39. In this example, the approach to reintroduction of resends described in this section can remove the RR methods ‘object1m’ and ‘object2n’, replacing the corresponding RR sends by resends. The resulting hierarchy is then the same as in Figure 4.38.

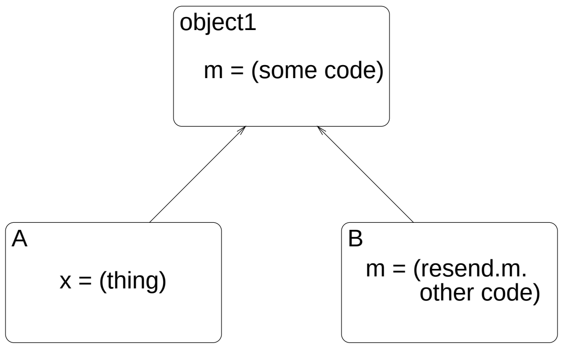


Figure 4.36: Example hierarchy before restructuring.

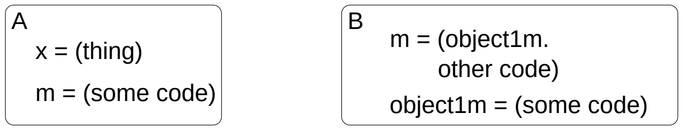


Figure 4.37: Restructured hierarchy.

Even if all RR methods cannot be removed by the reintroduction of resends, the duplication of implementation of RR methods and the original RI methods

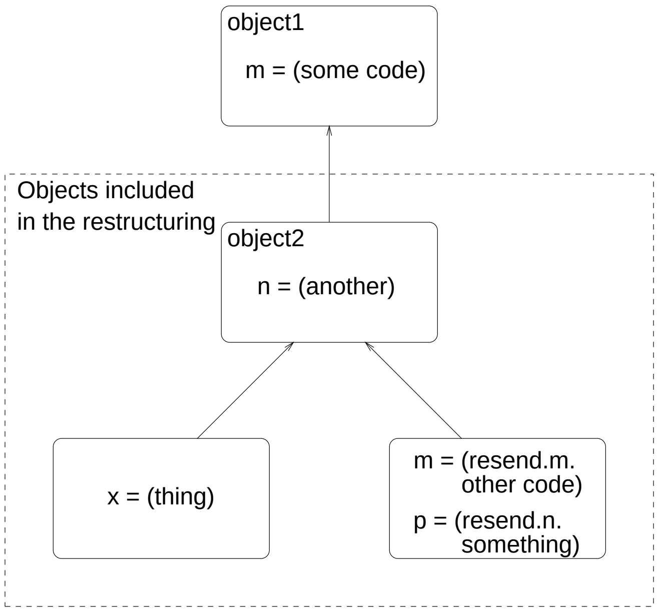


Figure 4.38: Example hierarchy before restructuring.

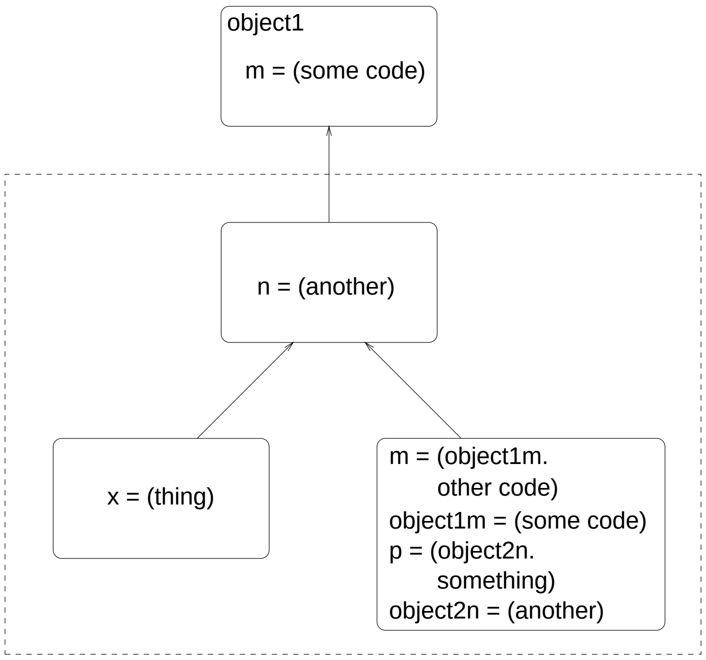


Figure 4.39: Restructured hierarchy.

could be removed using one of the following approaches. For example, each RI method could be implemented as the appropriate RR send for the RR method introduced to replace any resends invoking that RI method. Then, (flattened) objects including an RI method would need to include the corresponding RR method, as well as those objects containing the RR send replaced by that RR method. (For example, in Figure 4.37, object A would need to include the RR method `object1m` as it contains the RI method `m`, even though it does not contain any methods which include a resend which would be replaced by `object1m`.) Alternatively, a resend could be replaced by delegating the resent message to the object where the RI method is defined. This has the same effect as the resend, but does not require the inheritance hierarchy to be structured in a particular way, and does not duplicate the RI method. A consequence of this approach is that objects containing such a delegated send would have to be able to refer to the object to delegate to. This would require a uniquely named data slot to be created for every object delegated to, and that slot included in every (flattened) object which includes a corresponding delegated send. If this approach were taken, it would still be desirable to replace introduced delegated sends by resends in a similar way to that described in this section, thus allowing removal of the uniquely named slots introduced to refer to the objects to delegate messages to.

4.8 Describing the results of restructuring

The restructured system may be significantly different 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. 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 Section 4.1, the system could report, for example:

- Object o1 (in Figure 4.1) is replaced by object o1 (in Figure 4.2)
- 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

4.9 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 behaviour. There appears to be no simple solution to this, except to encourage the style of avoiding reflection in application code [Self4.0].

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 inferred from the system may not match recognisable real world abstractions. Chapter 6 discusses results from realistic applications of Guru, which demonstrate that the hierarchies produced are well structured in practice.

Section 4.4.4 describes how the removal of resends can potentially alter the behaviour of a system unless checks are made that no methods or message sends exist in the original system which have the same names as the automatically created RR methods. These checks are not currently implemented, but the possibility of introducing an error in the ways described is insignificant in practice, and has never been observed. The names created for RR methods are too unusual for a human programmer to have accidentally used the same names. Implementing the checks would be straightforward, apart from a small limitation described in Section 4.4.4.

The user of Guru has to chose the collection of objects to restructure, and when to perform the restructuring. The problem of extending an existing hierarchy to include a new object (whether a traits object or not), while minimising duplication of features or conforming to other criteria, is not addressed directly. One way of approaching this problem is to create the new object with no inheritance, duplicate the features which are required, and then restructure the hierarchy using Guru. Previous work which addresses this problem more directly is discussed in Sections 3.8.3 and 4.11.1.

4.10 Applying inheritance hierarchy restructuring to other languages

The IHR described for Self could also be implemented for other object-oriented programming languages.

In order to implement IHR for class-centric programming languages, classes rather than objects have to be considered. Classes usually define instance variable structure (rather than actual instance variables) which can be shared by inheritance[1]. Code within methods may access, or assign to, instance variables which must be defined (or inherited) for instances of the class that the method is defined in. Therefore, IHR must ensure that each inferred class which defines methods that access, or assign to, an instance variable must also include (or inherit) that instance variable in its definition of instance variable structure.

One of the most important considerations in implementing IHR for another language is the definition of equality of features. For example, to implement IHR for a statically typed language, equality of methods would have to consider equality of the type declarations of arguments, temporary variables and the method's return values, as well as equality of the statements within the methods.

In a class-centric language, the classes of objects which are sent messages should be preserved. In languages in which classes are objects, this means that metaclasses will also need to be preserved. In Smalltalk, the metaclass hierarchy is parallel to the class hierarchy, but, using the IHR described in this chapter, there is no reason to expect the restructured class hierarchy to be parallel to the restructured metaclass hierarchy. Therefore, further work would be needed to make the class and metaclass hierarchies parallel before they could be used to replace the original hierarchies.

IHR may result in hierarchies which include multiple inheritance. Some

[1]Smalltalk classes may have 'class instance variables', but this feature is rarely used.

object-oriented programming languages, notably Smalltalk, have only single inheritance. Hence, for such a language, a technique for implementing the equivalent effect to multiple inheritance would be required.

4.11 Comparison with previous work

Note that the inheritance hierarchy structure is only one aspect of the design of an object-oriented system, and other work ([Casais90, Casais92, Casais94, Casais95, Chae96, Hoeck93, Opdyke92, Lieberherr88] as well as parts of [Lieberherr91]) has investigated (semi) automatic restructuring of object-oriented systems with regard to other aspects of design. The main focus of this section will be on those aspects which are most closely related to the work described in this chapter.

4.11.1 Casais

Casais [Casais90] describes issues and possible solutions related to the evolution of classes. He summaries and comments on work by Johnson et al [Johnson88] and Lieberherr [Lieberherr88]. He proposes an algorithm for restructuring hierarchies, so that they include only 'legitimate' uses of inheritance. For example, cancellation of inheritance is eliminated.

In [Casais92], Casais describes an incremental algorithm for restructuring inheritance hierarchies, which uncovers design flaws when new classes are added to an existing inheritance hierarchy. An inheritance hierarchy is restructured when a class is added which has no class from which it can inherit the features that it requires without inheriting unwanted features, which have to be *explicitly rejected*. The algorithm removes explicitly rejected features from a hierarchy by creating new abstract classes and moving features 'up' the inheritance hierarchy into these new classes. The algorithm is not guaranteed to remove all duplication of features, as it only examines the hierarchy local to the insertion of each class.

In [Casais92, Casais94], Casais describes the application of two incremental restructuring algorithms to the standard Eiffel [Meyer92] class libraries. His algorithms are applied only to the interfaces of classes, and not the actual source code of methods. Therefore, he does not discuss the problems identified in this chapter of applying such an algorithm to real programs while ensuring preserving their behaviour.

The first algorithm (called *decomposing class interfaces* in [Casais94]) separates classes into smaller classes to make explicit different uses of inheritance. For example, inheritance of implementation is split into a separate class from inheritance of interface (subtyping).

The second algorithm (called *factorizing object definitions* in [Casais94]) aims to correct errors in the inheritance structure of classes, such as when features are overridden too often and when common features have not been factored into abstract classes.

He analyses the results of applying both algorithms to Eiffel classes to demonstrate how they can correct or identify errors in design.

4.11.2 Chae

Chae [Chae96] proposes two restructuring approaches. The first is for restructuring classes to improve their cohesion, that is, splitting a class into components if methods lack cohesion, as measured by a metric based on the instance variables referenced by methods.

Another analysis is proposed for ensuring interface inclusion in inheritance hierarchies, that is not allowing inheritance to omit any methods. A related analysis ensures that overriding methods 'behaviourally conform' to the methods they override; that is, they have certain similarities which suggest that they are indeed specialisations of the methods they override, rather than just have the same name but represent different 'method abstractions'.

No experimental results of applying the suggested restructurings are reported.

4.11.3 Hoeck

Hoeck [Hoeck93] describes a representation of object oriented programs and some semi-automatic restructurings which can be applied to them. He further describes a tool based on this representation which implements some of the restructurings for Smalltalk programs. The tool identifies candidates for the restructurings, and the user specifies which restructurings to perform; the tool then makes the appropriate alterations to the Smalltalk program. His representation of programs is based on an extension of the *class dictionary graphs* of Lieberherr et al [Lieberherr91], which he calls *GEOID*. In addition to some 'primitive transformations' similar to those presented by Opdyke [Opdyke92], he presents four more sophisticated restructurings: *split classes with little cohesion*, *group arguments*, *remove redundancies* and *remove empty abstractions*.

Split classes with little cohesion means splitting up a class if methods lack cohesion, based on whether separate groups of instance variables are referenced by distinct groups of methods. This is different in detail, but similar in intent, to the approach taken by Chae [Chae96] mentioned in Section 4.11.2. Hoeck suggests alternative arrangements for the restructured classes resulting from splitting up a class.

Group arguments means that if the equivalent arguments appear together as arguments to many methods then this indicates that a new class should be created which groups the arguments together as single objects. Appropriate methods must then be altered to replace occurrences of these two arguments with a single argument of this new class.

Remove redundancies means that a variable or method occurring in more than one class should be factored into a single variable or method to remove the duplication. His comparison of methods does not include the case where the methods contain equivalent code, but rather cases which indicate that they *might* be equivalent. Hoeck identifies three different situations and corresponding restructurings:

- where the variables or methods occur in classes which have a common parent, which has no other children (called an *exclusive common parent*) as shown in Figure 4.40. The variable or method is simply moved to the common parent and removed from the original classes.

- where the variables or methods occur in classes which have a common parent, which does have other children (called an *non-exclusive common parent*) as shown in Figure 4.41. A new class is created as a subclass of the non-exclusive common parent, for factoring the variable or method into.

- where the variables or methods occur in classes which do not have a common parent as shown in Figure 4.42. A new class is created for sharing the variable or method using (multiple) inheritance from the original objects.

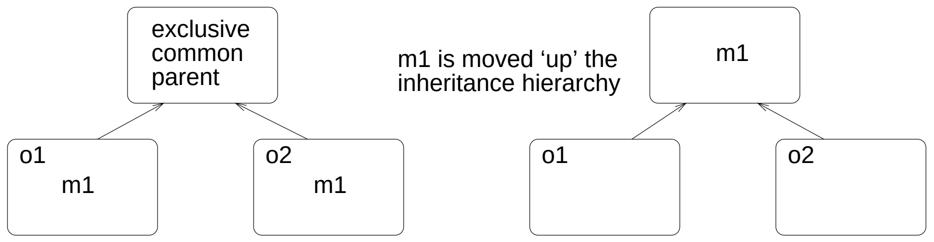


Figure 4.40: Exclusive common parents

Only immediate parents are considered.

Further mention of Hoeck's 'exclusive common parent' restructuring is made in Section 7.1.4, as it has some similarities with a restructuring described in Section 7.1.

Remove empty abstractions means to remove classes which do not define any variables or methods. This is very similar to the restructuring described in Section 3.5.3, but is simpler because Hoeck does not consider removing transitively unnecessary inheritance which may result in hierarchies which have multiple inheritance.

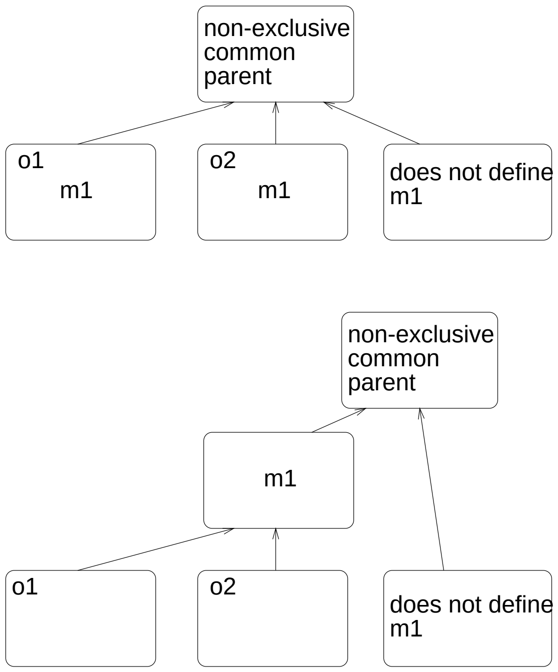


Figure 4.41: Non-exclusive common parents

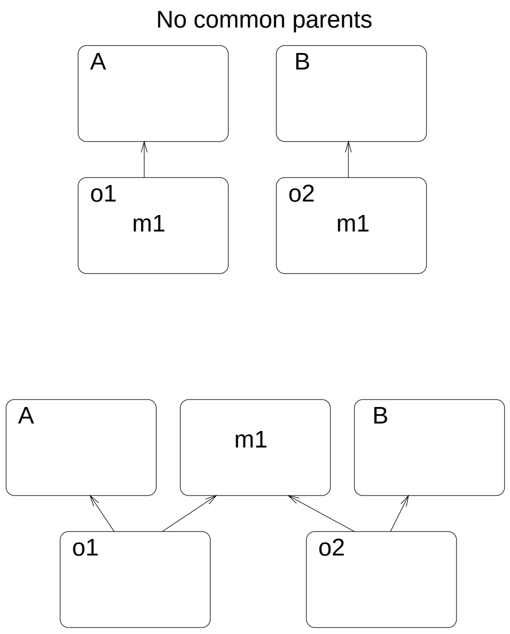


Figure 4.42: No common parents

4.11.4 Lieberherr and the Law of Demeter

The Law of Demeter [Lieberherr88, Lieberherr89] has been proposed as a way of decoupling classes. Informally, the Law of Demeter says that an object should only send messages to a limited set of objects: its instance variables, the arguments of methods and the receiver of the message which invoked the currently executing method (in Self, self). This reduces the coupling between co-operating classes. Restructuring techniques are described in [Lieberherr88] to make programs conform to the Law of Demeter. A critique of the Law of Demeter is presented in [Sakkinen88].

4.11.5 Opdyke

Opdyke [Opdyke92, Opdyke93] presents a collection of 26 simple program restructurings (he uses the word *refactorings*), which he shows to preserve the behaviour of programs. Two examples of these restructurings are to rename a class, and to move a variable to a subclass. From these simple restructurings, three more sophisticated restructurings are defined; *refactoring to generalize*, *refactoring to specialize* and *refactoring to capture aggregations and components.*

Refactoring to generalize means factoring commonalities into an abstract superclass. Opdyke proposes ways of manually modifying the classes to be generalised so that similarities which cannot automatically be refactored can be made compatible. While it is not an insurmountable limitation, he only describes refactoring *two* classes into an (abstract) immediate superclass. The classes to be generalised are chosen by the user. This part of Opdyke's work is further discussed in Section 5.8.1 to compare it to the way that Guru has been extended to refactor shared expressions from methods, described earlier in the same chapter.

Refactoring to specialize means splitting classes which effectively include subclasses encoded according to flags and conditional statements into separate classes. Then, the flags and conditional statements can be replaced by creating objects of the appropriate subclasses and by using polymorphism to obtain the correct behaviour. This involves splitting each existing method which contains a relevant conditional statement into several methods, one for each branch of the conditional statement, as methods of the same name in the appropriate subclasses.

Refactoring to capture aggregations and components means modifying the representation of objects and their components. For example, moving a method from the object's class to the component's class. (In some circumstances this will be similar to the restructuring advocated by the Law of Demeter [Lieberherr88] mentioned in Section 4.11.4.) Furthermore, restructuring a relationship modelled using inheritance into an aggregation is described.

4.11.6 The Smalltalk Refactory

The Smalltalk Refactory [Brant1] or *refactoring browser* implements many of the simple refactorings proposed by Opdyke [Opdyke92]. The refactoring browser is further discussed in Section 7.1.4, as it provides a user interface for applying restructurings.

4.11.7 Pedersen

Pedersen's [Pedersen89] work is not directly restructuring inheritance hierarchies, but is mentioned here because it is an alternative approach to one of the main problems motivating the restructuring of inheritance hierarchies. He recognises that the conventional inheritance mechanism of object-oriented programming languages restricts how inheritance hierarchies can be extended, by only allowing specialisations of existing classes to be created (by creating subclasses). He notes that in many cases, it is more useful to be able to create a generalisation from several classes (by creating a superclass for existing classes), as generalisations are often discovered from examples. He argues that languages should include a generalisation inheritance mechanism, and shows how such a mechanism can co-exist with the specialisation approach of the conventional inheritance mechanism.

4.11.8 Zimmer

Zimmer [Zimmer95] describes manual reorganisation of object oriented programs based on design patterns [Gamma94]. He suggests that code should be examined to discover cases where it is similar to a design pattern, and then restructured to make it conform to the guidelines in [Gamma94]. Also, design errors which are addressed by the patterns in [Gamma94] should be identified and the code restructured to solve the design errors using the appropriate pattern. He concludes that using design patterns is a promising approach, and could benefit from tool support.

4.11.9 Other related work

Other work which deals with issues similar to those related to automatic restructuring of hierarchies includes work on discovering abstract data types (ADTs) [Canfora93], and objects and classes [Canfora96, Ong93] in conventional programming languages. Also relevant to (manual) program restructuring are [Gibbs90, Anderson90, Meyer90]. Furthermore, work on program understanding and design recovery [Biggerstaff89] and restructuring [Griswold93] in conventional programming languages has some similarities to the work described in this chapter. However, as conventional programming languages do not have inheritance, the similarities are limited.

As mentioned in Section 3.7, there is some similarity between automatic inheritance hierarchy restructuring, data mining [Holsheimer94] and conceptual clustering [Fisher87] on small sets of data, as indicated by two papers by Mineau et al [Mineau90, Mineau95]. However, such applications typically deal with very large amounts of data and hence use heuristics to find approximations of optimal hierarchies. Another area of database research which shares some similarities to IHR is that of *schema evolution*, as shown by [Hürsch93] which is co-authored by Lieberherr [Lieberherr91], whose work is discussed in Sections 3.8.1 and 4.11.4.

However, many of the important issues in schema evolution arise because of details irrelevant in non-persistent programming languages[2].

4.12 Summary

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 and error prone.

Guru tackles the problem of automatically restructuring an inheritance hierarchy, while preserving the behaviour 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 and a structure which reflects the inherent structure of the objects. The replacement hierarchy can then be used to replace the original hierarchy.

[2]Self has a limited form of persistence due to saving entire images, but does not have persistence on the granularity of individual objects.

Chapter 5

Refactoring expressions from methods

This chapter describes how Guru has been extended, from automatically restructuring an inheritance hierarchy (as described in Chapter 4) to automatically refactoring shared expressions from methods [Moore96b] at the same time. In the resulting inheritance hierarchies, none of the methods and none of the expressions that can be factored out are duplicated.

Factoring shared methods into traits objects 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 (and restructuring) inheritance hierarchies and methods which maximise 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, or refactor methods, 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 and method refactoring is also error prone and, while a system works, however badly structured it is, the temptation is to leave it alone. Automatic restructuring of inheritance hierarchies and refactoring of methods can improve the factoring of methods into traits objects and the factoring of shared code into methods.

5.1 Introduction

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 chapter is that expressions which are factored out are called 'factored expressions' and the methods created to share factored expressions are called 'factoring methods'.

The behaviour 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 objects 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 object (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. As described in Section 2.3, 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 this chapter this condition is assumed to be valid, and Section 5.5 will describe how this condition is guaranteed by the refactoring system. For conciseness, some refactorings are shown for expressions occurring only once (as above).

5.2 The expressions which could be factored out

Not all expressions may be factored out of a method. For example, a block with a non-local return (see Section 2.3.5) 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, as explained in Section 2.3.7.

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, measured as the number of potential message sends (measured statically, rather than the number of actual message sends that will occur at run-time), to determine whether to factor it out, so that cases such as this do not occur. Section 6.3 includes a discussion of whether this metric is adequate in practice.

Passing a reference to an argument or local variable of a method to another method does not allow the argument or local variable reference to be altered. A message can be sent to an argument or local variable, which may change the state of the object that it refers to (thus changing the object), but the identity of the object referred to by an argument or local variable cannot be altered by passing a reference to the argument or local variable to another method. For example, consider the following code:

```
m1: a = (| t |
         t: 1.
         a: 1.
         m2: t.
         m2: a.
         self t: 3)
m2: t = (t: 2)
```

The statement t: 1 in method m1: makes the local variable t refer to the object 1. The statement a: 1 in method m1: does not alter the object referred to by argument a, but sends the message a: to self. The statement m2: t results in the method m2: being executed, with the object t as the argument. The expression t: 2 in method m2: does not alter the object referred to by t; rather the message t: is sent to self with the argument 2. Similarly, for the statement m2: a. The statement self t: 3 does not assign to the local variable t, but rather sends the message t: 3 to self.

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. 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 (see Section 2.3.5). This restriction is very 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.

5.3 The expressions which are factored out

A limitation of the current implementation of Guru is that only expressions or subexpressions which evaluate to an object are factored out. Such expressions or subexpressions will be called *complete expressions* (whether they are expressions or subexpressions). For example, in the expression `a b: c d` (this is the same as `a b: (c d)`), the expressions `a`, `c`, `c d`, `a b: c d` evaluate to an object. In the expression `a b: c d`, the subexpressions `b: c`, `b: c d`, `d`, and `a b: c` do not evaluate to an object. The only expressions which are parameterised out of an expression are references to arguments or local variables of their enclosing method or block. For example, the expressions `a b: c` and `a b: d` can be refactored as `newMethod1: c` and `newMethod1: d` using `newMethod1: c = (a b: c)` only if `c` and `d` are arguments of their enclosing method. This limitation means that expressions such as `a b: (x y z)` and `a b: (p q r)` cannot be refactored as `newMethod1: (x y z)` and `newMethod1: (p q r)` using `newMethod1: c = (a b: c)`.

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 using `newMethod1 = (a b)`, so that `a b c` would become `newMethod1 c` and `a b d` would become `newMethod1 d`.

Another example, including factoring of repeated subexpressions within expressions and methods, and within factoring methods is shown below:

```
m1 = (a b c: a b d)
m2 = (c: (f g e: a b d).
        f g h)
```

produces the following methods[1]:

```
m1 = (newMethod3 c: newMethod2)
m2 = (c: (newMethod1 e: newMethod2).
        newMethod1 h)
newMethod1 = (f g)
newMethod2 = (newMethod3 d)
newMethod3 = (a b)
```

Guru can also refactor expressions such as `a b c` and `x b c`, but only in limited

[1]The formatting has been improved by hand.

circumstances. This is a limitation of the implementation of Guru rather than this method of refactoring. A factoring method newMethod1: a = (a b c) can be created, and the expressions a b c and x b c would then become newMethod1: a and newMethod1: x respectively. The implementation of Guru only allows for expressions to be parameterised for messages referring to arguments or local variables. Therefore, this refactoring is performed by Guru only when a and x are references to arguments of their enclosing methods. A similar refactoring is performed if the expression a b c appears more than once, and a refers to a local variable of the enclosing method(s) of the expressions, as explained in Section 5.6.

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. The names of arguments are ignored when comparing unrefactored methods, as well as factoring methods and expressions. (A current implementation anomaly is that for comparing factoring methods, argument names are only ignored if they *were also* arguments, rather than temporary slots, of the enclosing methods of the expressions that they factor out.)

The primary reason that Guru performs factoring in the way that it does is so that it can perform factoring before inheritance hierarchy restructuring, independently of the inheritance hierarchy that will result. Also the comparison of expressions is independent of the methods that they occur in. The computational complexity of the approach taken is relatively low. Other forms of factoring are discussed in the following section.

5.4 Alternative ways of factoring out common code

Alternative approaches to factoring out common code, which are not implemented by Guru, are discussed in this section.

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 objects which have a common parent, then a factoring method `newMethod1 = (newMethod2 b c)` could be created in the parent, and the methods `newMethod2 = (a)` and `newMethod2 = (x)` could be created in the appropriate (child) objects. The original expressions (`a b c` and `x b c`) could then be replaced by `newMethod1` in both cases. This refactoring could result in making further refactorings possible, by making methods which were previously different, now similar. Consider the objects in Figure 5.1.

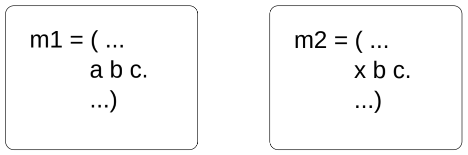


Figure 5.1: Two objects with methods sharing partial expressions

Using Guru's refactoring approach would lead to the results shown in Figure 5.2. The alternative approach described earlier in this section would produce the results shown in Figure 5.3.

Alternatively, the expressions could be factored out by creating factoring method `newMethod1 = (b c)` in the objects which may result from the message sends `a` and `x` (or their parent objects), and replacing the expressions with `a newMethod1` and `x newMethod1` respectively. If the message sends `a` and `x` always result in the same object, or objects with a shared parent object, then this would be a very good solution, as shown by Figure 5.4. Furthermore, this form

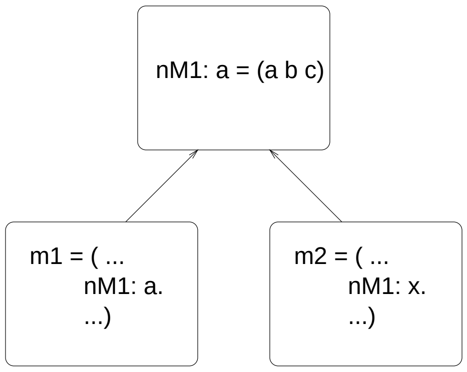


Figure 5.2: Guru's refactoring (ignoring implementation limitations)

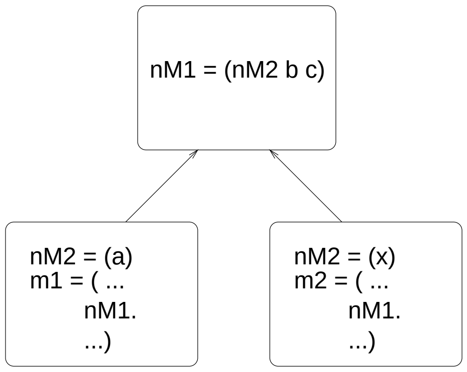


Figure 5.3: An alternative way of refactoring

of restructuring would conform to the 'Law of Demeter' [Lieberherr88].

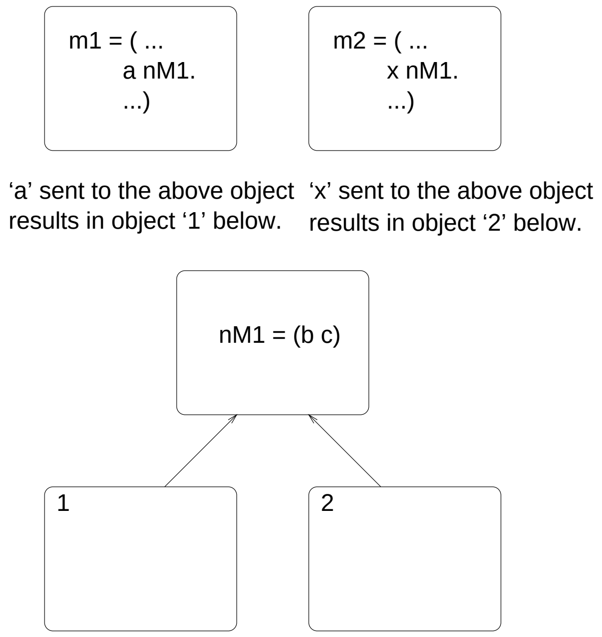

m1 = (...
a nM1.
...)
m2 = (...
x nM1.
...)
'a' sent to the above object results in object '1' below.
'x' sent to the above object results in object '2' below.
nM1 = (b c)
1
2

Figure 5.4: Another way of refactoring ('Law of Demeter' [Lieberherr88] compatible)

However, determining 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 many different objects (with many different, i.e. not shared, parent objects). If these objects were not restructured, then the refactoring would increase the number of methods in the system, which would be counter-productive.

There are many other ways that methods and expressions could be refactored, in addition to the ways discussed above. Consider the following methods:

```
m1 = (some: (complicated: [ code like this ]))
m2 = (some: (complicated: [ code is this ]))
```

These two methods are similar, but it is not straightforward to factor out the commonality.

One way to do this would be:

```
nM1: msg = (some: (complicated: [ (msg sendTo: code) this ]))
m1 = (nM1: 'like')
m2 = (nM1: 'is')
```

Or alternatively:

```
nM1: block = (some: (complicated: [ (block value: code) this ]))
m1 = (nM1: [ | :e | e like])
m2 = (nM1: [ | :e | e is])
```

Code such as:

```
m = (| temp |
        temp: set copy.
        temp add: item1.
        temp add: item2.
        temp add: item3.
        temp add: item4.
        temp add: item5.
        temp add: item6.
        temp)
```

could be 'refactored'[2] into:

```
m = (| temp |
        temp: set copy.
        6 do: [ | :n |
            temp add: (('item',((n + 1) printString)) sendTo: self) ].
        temp)
```

(6 do: aBlock results in aBlock being evaluated with arguments 0 to 5.) This example is artificial and demonstrates that in some cases improving the amount of 'factoring' does not necessarily lead to easily understood code.

5.4.1 Refactorings applicable to complete methods

The effect that refactoring has on complete methods rather than individual expressions is considered in this subsection.

It is possible to factor out either the similarities or the differences between

[2]Using the word 'refactored' in a general sense.

methods. While these two approaches may seem to be 'opposites', they produce similar, but subtly different, results. Consider the example shown in Figure 5.5.

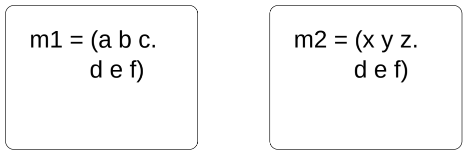


Figure 5.5: Example objects

If the similarities are factored out, as done by Guru, the result would be as shown in Figure 5.6.

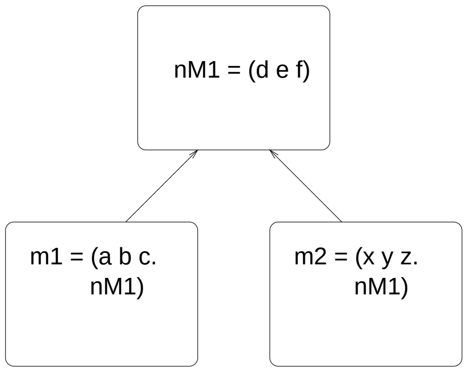


Figure 5.6: Factoring similarities

If the differences are factored out, the result would be as shown in Figure 5.7.

While this does not appear to be very neat, consider what the result would be if instead of m1 and m2, the two methods to be refactored have the same name, for example m. The result of factoring out the differences would be as shown in Figure 5.8.

This can be argued to be the best form of refactoring in the case where the methods have the same name, as it reflects the idea of creating a common abstraction of a method, and the two child objects only define the specialisations required (that is, the differences) to use this common abstraction.

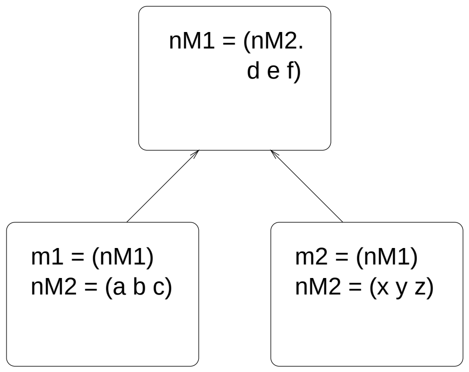


Figure 5.7: Factoring differences

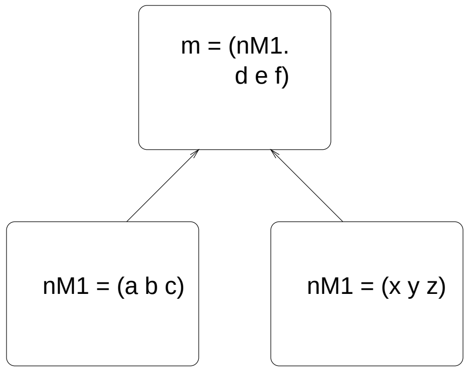


Figure 5.8: Factoring differences of methods with the same name

Consider another example:

```
m1 = (a b.
      a c.
      a d)
m2 = (x b.
      x c.
      x d)
```

could be refactored into:

```
nM1: a = (a b.
          a c.
          a d)
m1 = (nM1: a)
m2 = (nM1: x)
```

These examples are intended to illustrate that there are many different ways to refactor methods. Some of them are only applicable in limited circumstances.

Section 8.1 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 6.3.

If many different forms of refactoring were available, then the system would have to decide which form to use in cases where more than one form of refactoring could be applied.

5.5 Combining method refactoring with inheritance hierarchy restructuring

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 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, as shown in Figure 5.6). Secondly, the refactoring of methods can discover traits objects and inheritance relationships which would not necessarily otherwise exist. That is, if two objects 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 6.3.

Restructuring a collection of objects including refactoring methods is the same process as that described in Chapter 4, except the flattened preserved objects have had their methods refactored using the *method refactoring algorithm*. The method refactoring algorithm is applied to all the methods in all the flattened preserved objects included in the restructuring. This algorithm 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. Then 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, (start + end) > 0, 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, with a unique name, is created for each factored expression. The system needs to check that the name of any factoring method is not used anywhere else in the system, and that no message send could cause a factoring method to execute inappropriately, in order to ensure that the behaviour of the system is not altered by refactoring. The check that no other method has the same name as a factoring method is straightforward. Section 4.4.4 explains the limitations of checking whether any message sends exist which could execute a particular method inappropriately.

In the example above, the following methods are created:

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

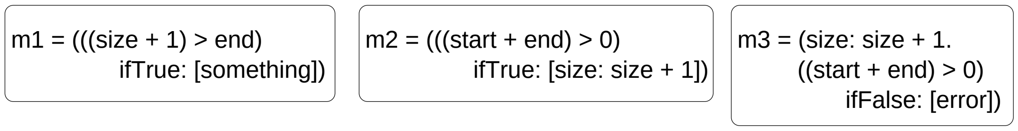
A replacement method is made for each method which includes any factored expressions. These replacement methods are modified such that each factored expression is replaced by the appropriate message send to invoke the appropriate factoring method. (This replacement is also done for factoring methods.)

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 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 flattened preserved objects, with their refactored and factoring methods, and methods which have not been refactored (because they do not contain refactored expressions), are then restructured as described in Chapter 4. The extended IHI algorithm is applied to these flattened objects (as described in Section 4.5) as if there had not been any refactoring of methods. In this example, if the three methods were in three different objects, as shown in Figure 5.9, then they would be restructured with method refactoring into the hierarchy shown in Figure 5.10. Note that there is no requirement for the names of the methods to be different in order for them to be refactored; different names were used in this section simply to make the explanation clearer. Of course, if the methods have the same names then they must be in different objects. The methods are shown in different objects in this example, but they could equally be in the same object (if they have different names), in which case the method refactoring would be unaffected, but the structure of the resulting hierarchy would be different from the example shown.



```
m1 = (((size + 1) > end)
          ifTrue: [something])
```

```
m2 = (((start + end) > 0)
          ifTrue: [size: size + 1])
```

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

Figure 5.9: Original objects.

As the appropriate factoring methods are included in the flattened objects which include methods containing their factored expressions, the restructuring ensures that the appropriate factoring methods will be in the appropriate restructured objects. A factoring method will be located in the object from which all methods which included its factored expression inherit. In other words, if an expression is factored out of methods from only one object, then the factoring

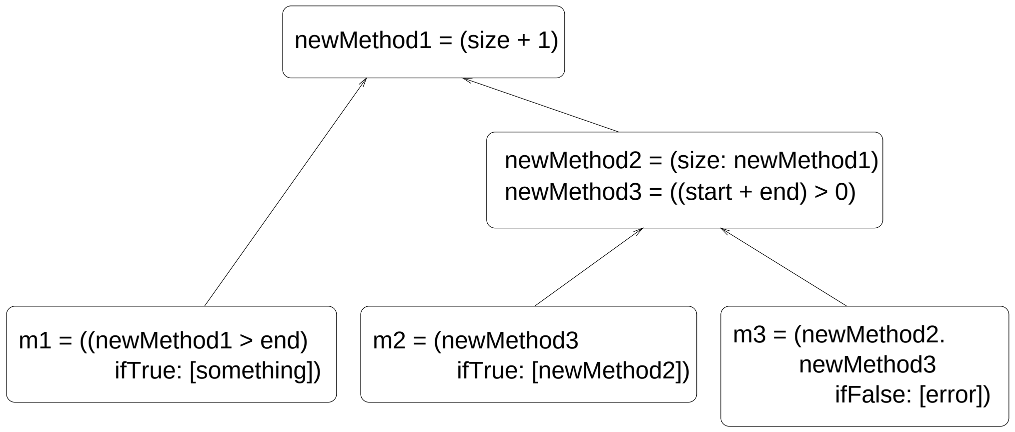

newMethod1 = (size + 1)
newMethod2 = (size: newMethod1)
newMethod3 = ((start + end) > 0)
m1 = ((newMethod1 > end) ifTrue: [something])
m2 = (newMethod3 ifTrue: [newMethod2])
m3 = (newMethod2. newMethod3 ifFalse: [error])

Figure 5.10: Method refactoring combined with IHR.

method will be in the same object as those methods. Alternatively, if an expression is factored out of methods from several different objects, then it will be in an object from which all those objects 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 object which includes a method which contains such a message send.

5.6 Limitations and problems

The names of factoring methods are unique system generated names which have no inherent meaning. These methods can be renamed by the user, and Guru will then rename all sends of the appropriate message, but it can be difficult to invent a short and meaningful name for many of the factoring methods.

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 a factoring method.

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].

Rather than creating a new factoring method for a factored expression, Guru could try to find an existing method which implements the factored expression. Consider the following methods:

```
m = (a b c)
n = (a b d)
p = (a b)
```

They will be refactored into:

```
m = (newMethod1 c)
n = (newMethod1 d)
p = (newMethod1)
newMethod1 = (a b)
```

A better solution may be to use the existing method which defines the factored expression a b, as shown below:

```
m = (p c)
n = (p d)
p = (a b)
```

However, due to polymorphism it is not possible to be sure that the message send p will execute the method p shown above. An object which inherits methods m and n may override method p, resulting in an error if methods m and n are refactored as above. If it were possible to determine that the method p above is executed in response to the message p by all objects inheriting methods m and n, then this would be a good way of factoring these methods. (Note that it is only possible to be certain that the method newMethod1 above is executed in response to the message newMethod1 because it will be the *only* method of this name in the system, and will be inherited by all objects which inherit from objects defining the methods m and n.)

Objects will be slightly changed by the refactoring version of Guru, because they will understand the messages implemented by the factoring methods they inherit. Furthermore, the reintroduction of overriding described in Section 3.5.2 has been implemented to allow factoring methods to be moved higher in the inheritance hierarchy than strictly correct. That is, the reintroduction of overriding allows preserved objects to inherit factoring methods even if they do not contain any methods which include the corresponding factored expression. This is only allowed if it is necessary for removing an anomalous traits object (see Section 3.5.2). This can result in objects inheriting some factoring methods unnecessarily. However, within the same limitations as discussed in Section 4.4.4, the system can check that no message sends will cause factoring methods to be executed inappropriately.

If an RR method includes factored expressions, and the RI method which it replaces does not (because it is in an object not included in the restructuring), then the reintroduction of resends (see Section 4.4.4) will fail to replace the RR send with a resend. This is because to test whether an RR send can be replaced by a resend involves testing whether the RR method is the same as the RI method. If the RR method has been altered because of method refactoring, and the RI method has not, then they will be different. Another minor implementation problem is that RI methods included in a restructuring are always refactored if they contain any expressions which can be factored out. This is because the RR method created to replace an RI method is a copy of the RI method. If the RR method is removed by reintroduction of resends, then any expressions which occur only in the RI method will be unnecessarily factored out. (Method refactoring which has already been done will not be 'undone'.) This would result in more refactoring methods than necessary, and more potential message sends, than if those expressions were not factored out.

The reasons for the restrictions that expressions can only be parameterised according to arguments or local variables, and that comparison of arguments does not rely on the names of the arguments, are related to the details of the implementation of expression comparison. In particular, method arguments referred to in expressions are equivalent if they are in the same position in each method's list of arguments. For example; in methods m: a = ... and n: b = ..., any implicit self receiver message sends 'a' and 'b' inside m: and n: respectively are treated as equivalent. While it is clearly advantageous that the names of arguments should not matter for expression comparison, the reason for relying on the ordering of arguments is more subtle, and directly related to implementation concerns rather than any theoretical concern. When comparing methods with more than one argument, the equivalent arguments have to be compared consistently. For example, consider the two methods (in two different objects):

```
m: a M: b = (a c: b.
             a d: b)
m: x M: y = (x c: y.
             y d: x)
```

(This is an artificial example, as the refactorings below would not be used as they are no improvement over the original methods). The expressions a c: b and x c: y can be considered equivalent, if the factoring method newMethod1: a P: b = (a c: b) is created, and the expressions a c: b and x c: y are replaced by newMethod1: a P: b and newMethod1: x P: y respectively. Similarly, a d: b and y d: x can be replaced by the factoring method newMethod2: a P: b = (a d: b) and the expressions newMethod2: a P: b and newMethod2: y P: x respectively. However, when comparing the complete methods, it must be recognised that they are different as the order of the arguments in the two expressions is different. The comparison of expressions has been implemented so that it can be performed independently of other expressions within methods. The comparison of methods has been implemented simply as the combined comparison of all of their statements. The benefits of comparing methods as complete entities (also see Section 5.4.1) rather than as a collection of separate statements is small and not considered to justify further implementation refinements; the current implementation is sufficient to demonstrate the feasibility and results of refactoring shared expressions from methods. A disadvantage of more sophisticated techniques of comparing methods and expressions to overcome these limitation is that they would be computationally more expensive, as more possible variations would have to be checked to test equality of methods and expressions.

Equality of local variables depends on their names, rather than using the ordering of local variables, as the ordering of local variables is essentially arbitrary and would not benefit the comparison of equivalent methods or expressions.

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, or if the expressions are the same and e (or f) refers to a temporary variable of each of the enclosing methods. 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. This is a minor implementation limitation as, in practice, most factoring of shared expressions occurs for expressions smaller than or equal to a single statement.

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.

A possible criticism of method factoring is that it will degrade the performance of the code by creating many small methods with consequently more (run time) message sends. However, this is a weak argument, as sophisticated compilers, such as the Self system used for this work, are able to automatically inline code [Hölzle91] so that the amount of factoring at the source level does not affect the performance of compiled code.

5.7 Applying method refactoring to other languages

Some of the issues considered in Section 4.10 about implementing IHR for other languages are also relevant for implementing method refactoring for other languages. For example, in order to implement method refactoring for a statically typed object-oriented programming language, the types of arguments, temporary variables, return values of methods as well as the types of expressions, would have to be considered when comparing expressions.

Many of the issues addressed in this chapter are relevant for implementing method refactoring for any other language. For example, no matter which language is used, those expressions which may be factored out of methods need to be identified. Having identified these expressions, the method refactoring algorithm described in Section 5.5 can be used to determine the factoring methods to create. The details of actually implementing this are specific to each language, and so are beyond the scope of this thesis.

5.8 Comparison with previous work

Previous work on automatic and semi-automatic restructuring of object-oriented systems [Lieberherr88, Pun90, Casais92, Dicky96, Godin93, Hoeck93, Moore95, Opdyke92, Lieberherr91] has concentrated on areas other than refactoring expressions from methods; such as restructuring inheritance hierarchies considering methods as indivisible. However, there is some previous work which is sufficiently closely related to the method refactoring described in this chapter to be mentioned here.

5.8.1 Opdyke

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. 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.

5.8.2 Other related work

Work on refactoring expressions from functions or procedures in conventional (non object-oriented) programming languages [Griswold93, Lano93, Ward95] 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.

Griswold [Griswold93] and Ward (FermaT) [Ward95] describe restructuring/-reverse engineering tools for conventional languages which can refactor common code into procedures (as well as performing many other transformations). In contrast to Guru, rather than the common code being identified automatically, it is identified by the user. Both systems can make a procedure or function to replace user identified common code, and replace occurrences of the code with a call of that procedure or function.

Ideas related to factoring (of inheritance hierarchies and methods/procedures/-functions) as a feature of good design are explored in [Wolff94].

5.9 Summary

Improved factoring of methods makes a system more compact, improves consistency and increases code reuse. Manually refactoring methods is difficult, time consuming and error prone. Guru automatically improves the factoring of methods, while simultaneously restructuring an inheritance hierarchy. In hierarchies restructured with method refactoring, no methods, and none of the expressions that can be factored out, are duplicated.

The expressions which may be factored out of each method are identified. Then, a simple algorithm discovers new factoring methods, which are created by the system. The system creates replacement methods, refactored using the new factoring methods, and restructures the hierarchy as described in Chapter 4.

Chapter 6

Results

This chapter describes the results of applying Guru's restructuring, with and without refactoring of methods. The results are analysed in order to evaluate the approaches taken.

6.1 Experiments

The results of applying Guru to five inheritance hierarchies are presented in this section. The five inheritance hierarchies, which will be called the indexables, orderedOddballs, polygons, sendishNodes and samplers hierarchies respectively, were restructured using Guru, both with and without refactoring of methods. Unless stated otherwise, Guru was used automatically.

Three of the hierarchies, the indexables, orderedOddballs and polygons, 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. 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, and less so the polygons, are used extensively during the running of the Self system, as the programming environment is written in Self. Some of the objects 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 behaviour of the system. While the successful replacement of such fundamental objects is not a formal proof that the restructurings are correct, it gives *substantial* evidence.

The sendishNodes and samplers hierarchies were chosen because they are known to be imperfect, and as they are part of the Guru system itself, their 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 6.3.

In the figures, restructured objects are labelled either with the name of the object they replace, or with the name of the traits object they *most closely* replace. What is meant by this is that, for example, the object labelled traits string in Figure 6.5 does not necessarily define the same behaviour as the original object traits string. Only the behaviour of certain objects is preserved, as explained in Section 4.2.1, and traits string is not one of them. The label traits string in the restructured hierarchy is used only for convenience, and reflects the fact that this object is inherited by the equivalent objects in the original hierarchy. Restructured objects which cannot be labelled in this way are not named. All objects are shown with the number of non-inheritance slots they define.

Tables of simple metrics are presented. The entries labelled 'Message sends' refer to the total number of potential message sends in all the methods in all the objects in the appropriate hierarchy. Notice that this is not the same as the number of times a message will actually be sent at run-time. The table below gives some examples to explain this metric:

Expression	->	'Message sends'
a	->	1
self a	->	1
1	->	0
a b: c	->	3

Literals, explicit self and messages referring or assigning to local variables of methods and blocks are not included. 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 labelled 'Overriding methods' are the number of methods which override other methods both from inside and outside the hierarchies restructured. The hierarchies produced before using the automatic reintroduction of overriding only include overriding of methods from outside the objects restructured. Having too much overriding, or methods overridden too often, is an indication of poor design [Johnson88].

The entries labelled 'Time(1)' are the length of time it took to perform the appropriate restructuring on a Sun Sparc 20, before reintroducing overriding. The machine used was shared with other users which makes measuring the times less accurate than possible, but as only CPU time was measured the times are accurate enough for the purpose here. Times are in the format 'hours:minutes:seconds', 'minutes:seconds' or 'seconds' as appropriate. The entries labelled 'Time(2)' are the sum of the times to perform the reintroduction of overriding and resends, and any further reintroduction of overriding, elimination of empty objects or 'cracking' of leaves as appropriate. These times are included to give an indication of the performance of Guru, which is only a prototype implementation. A better implementation should be expected to be considerably faster. In particular, the performance of the reintroduction of overriding is currently very poor. The machine used had 128 Mbytes of real memory (RAM), and as it was shared with other users only part of this was available (approximately 64 Mbytes). In some cases, the times could be considerably improved by using a machine with more real memory. Also, the performance of Self 4.0 code is dependent upon how often it has been run, as only code which has been executed frequently is optimised [Hölzle94]. To reduce the effects of this feature, the restructuring and refactoring code was executed several times before timings were measured.

The entries labelled 'Reduction in size' are the percentage reduction in the number of potential message sends, which indicates the reduction in the total amount of code in the hierarchies. In cases where some manual intervention was needed due to limitations of the current implementation of Guru, the figures include such alterations.

6.2 Results

6.2.1 The indexables hierarchy

This hierarchy includes strings, vectors and sequences. A very similar hierarchy was restructured by an early version of Guru, and the results are described in [Moore95]. The differences between the hierarchy restructured by Guru, shown in Figure 6.1, and the hierarchy restructured in [Moore95] are because a newer version of the Self system has been used. Also, there are minor differences between the hierarchy shown in Figure 6.1 and that in [Moore96b], because of minor additions and modifications to the hierarchy during the development of Guru, since the results in [Moore96b] were collected.

The result of restructuring without refactoring, before reintroducing resends or overriding, is shown in Figure 6.2. This hierarchy is not exactly the same as in [Moore95], because of the differences in the original hierarchy. The leaf objects are preserved (see Section 4.2.1), and also their immediate parent objects

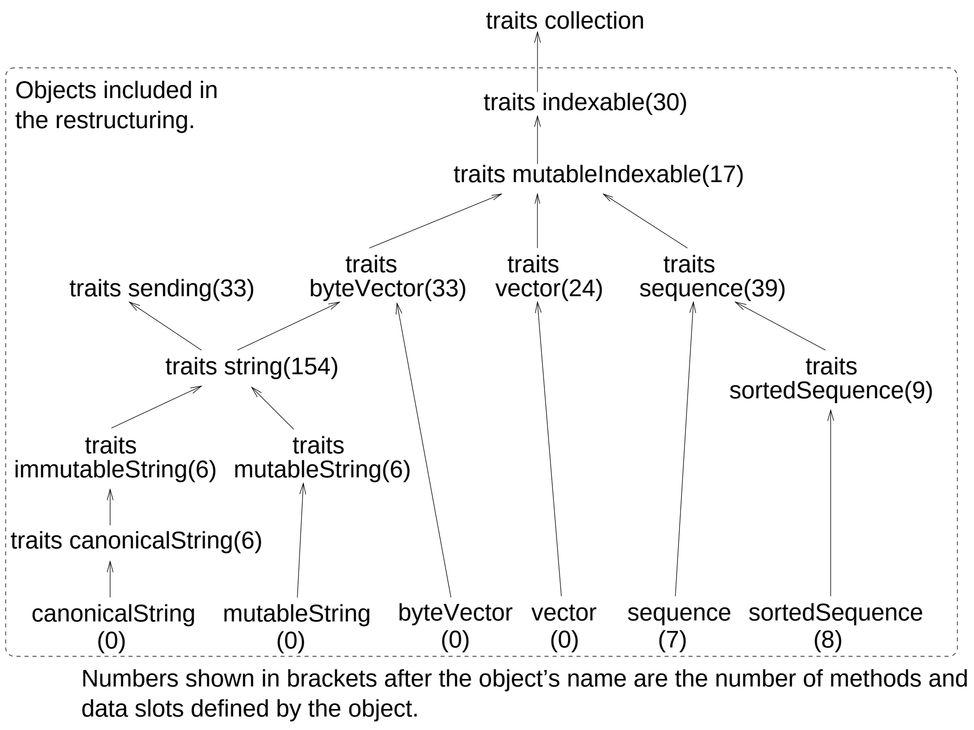


Figure 6.1: The original indexables hierarchy.

should be preserved because they should be expected to have other children, which would not be included in the restructuring (see Section 4.3). However, traits sequence and traits sortedSequence did not have any children other than the prototype objects in the image restructured. In order to make the restructuring as it should be expected to be, copies of sequence and sortedSequence were made, and consequently traits sequence and traits sortedSequence had children not included in the restructuring, and hence were identified as preserved objects.

This initially appears badly structured. However, close inspection reveals that a small number of slots have caused the creation of 7 small traits objects. These small traits objects represent anomalies in the structure of the hierarchy caused by the lack of overriding in the hierarchy. Using the automatic reintroduction of overriding described in Section 3.5.2, and automatically removing 7 methods created to replace resends as described in Section 4.7, the hierarchy is transformed into the one shown in Figure 6.3. (The reintroduction of overriding was applied

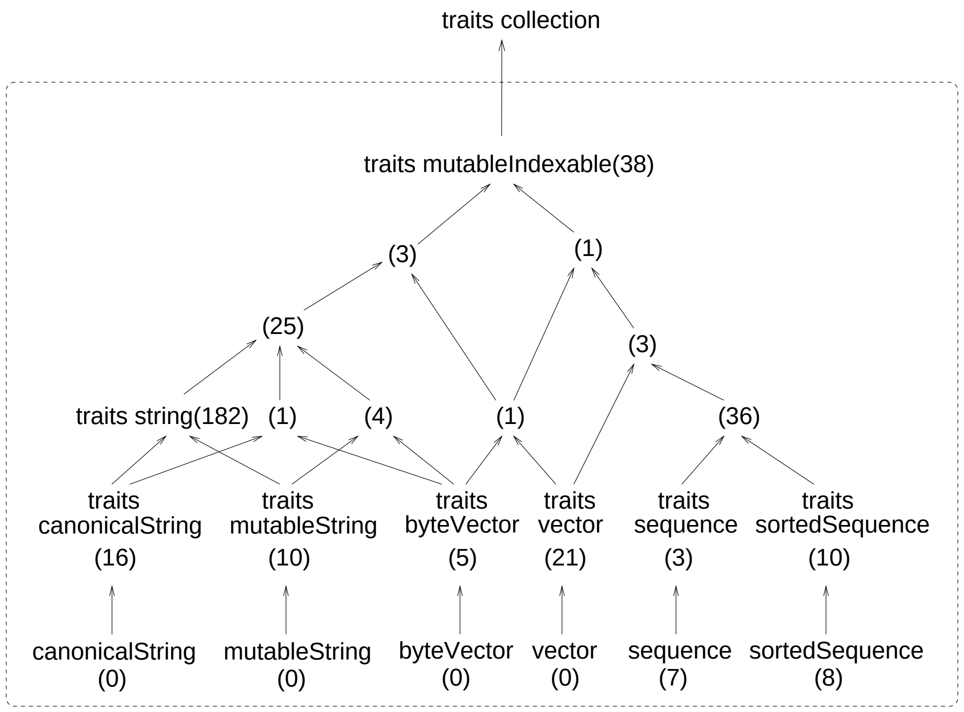


Figure 6.2: The restructured indexables hierarchy, before reintroducing overriding.

both before and after the reintroduction of resends, as this allowed one more traits object to be removed than otherwise).

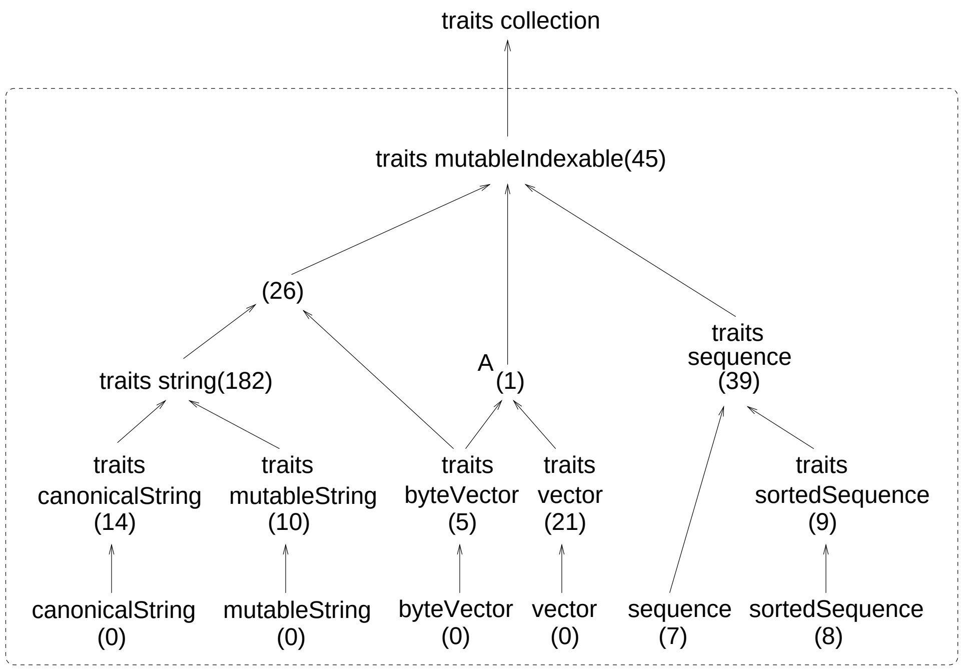


Figure 6.3: The restructured indexables hierarchy, after reintroducing overriding.

The single method defined in the object labelled 'A' was not moved to the object labelled 'traits mutableIndexable' because of the check whether all replacement offspring can implement all the messages sent to self in the method to be moved. Having manually moved this method, the (now) empty object 'A' was automatically removed (see Section 3.5.3). First, the parent slots of the children of 'A' are changed to refer to 'traits mutableIndexable', as shown in Figure 6.4. Then, the transitively unnecessary inheritance is removed, resulting in the hierarchy shown in Figure 6.5.

Guru without automatic reintroduction of overriding ensures that no object will inherit from the Self equivalent of a concrete class, which is a widely accepted design guideline [Johnson88]. The automatic reintroduction of overriding

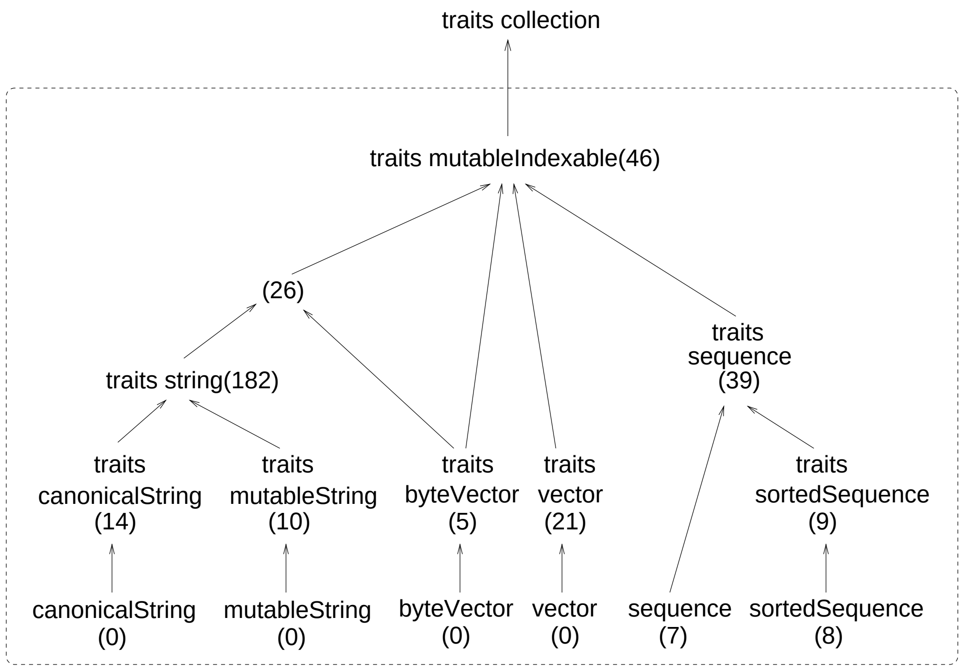


Figure 6.4: The restructured indexables hierarchy, after removing 'A'.

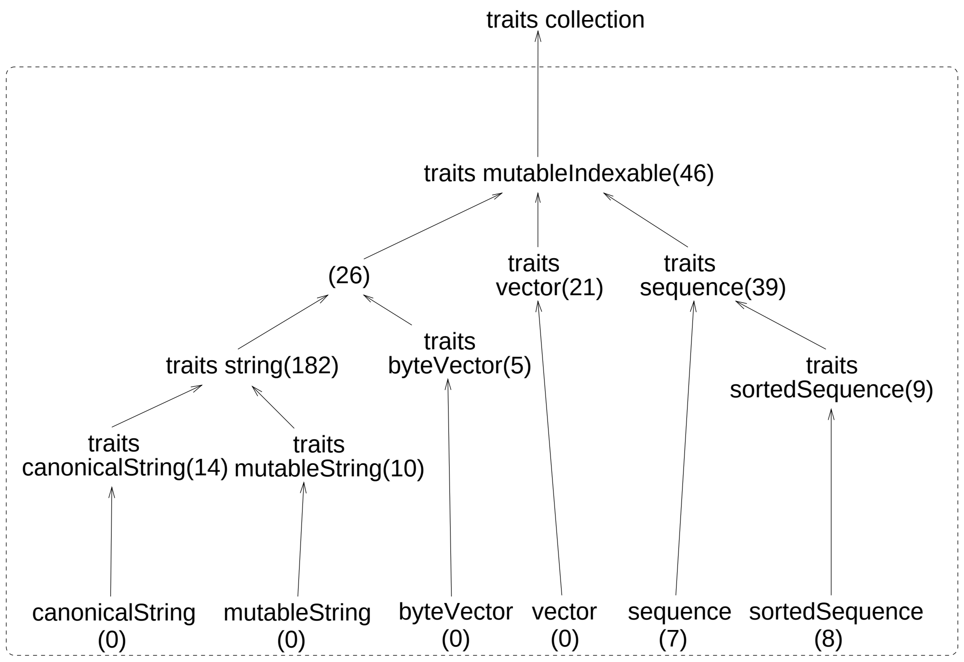


Figure 6.5: The final restructured indexables hierarchy.

has resulted in the replacement for traits sortedSequence inheriting from the replacement for traits sequence. If it is required that objects do not inherit from the Self equivalent of concrete classes, then this could be ensured by a minor modification to the automatic reintroduction of overriding.

The original hierarchy was then restructured with refactoring, producing the hierarchy shown in Figure 6.6, before reintroduction of overriding or resends.

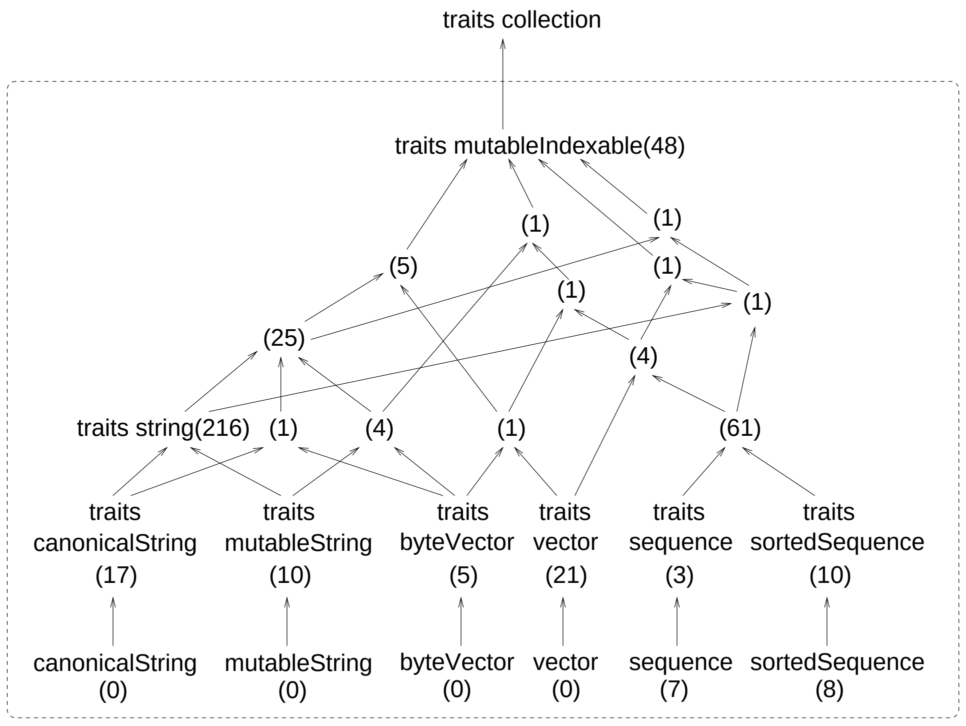


Figure 6.6: The restructured and refactored indexables hierarchy, before reintroducing overriding.

The hierarchy produced after reintroducing overriding and resends, (and then reintroducing overriding again) without any manual alterations is that shown in Figure 6.7.

It is interesting to observe that the hierarchy of Figure 6.7 is slightly different to the one produced by the restructuring without refactoring. An additional traits object has been discovered (labelled 'X') because of the factoring methods introduced. Furthermore, this additional traits object is very easy to understand; it is

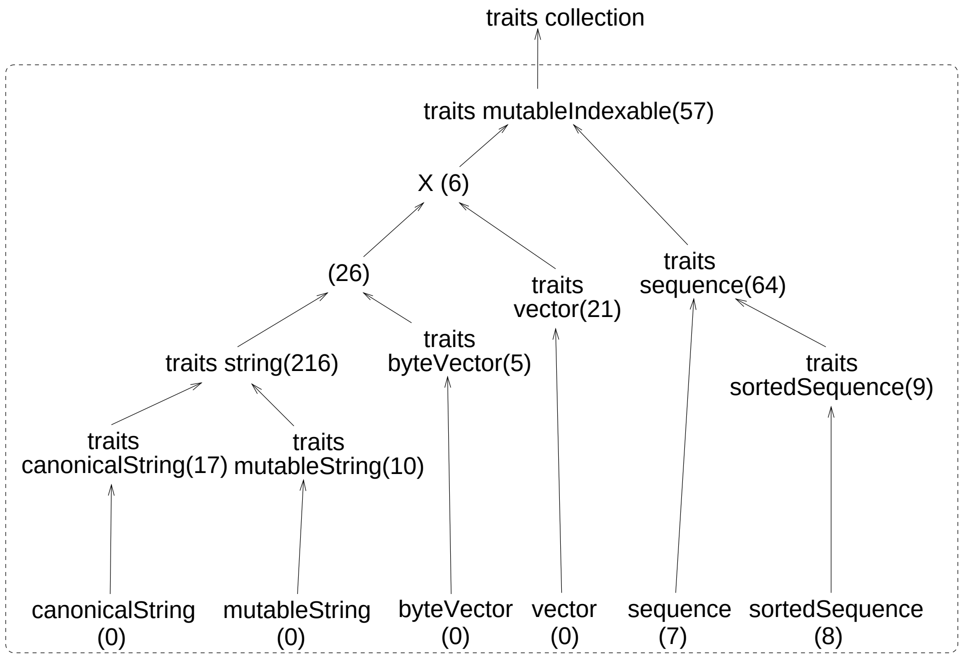


Figure 6.7: The restructured and refactored indexables hierarchy after reintroducing overriding.

the shared behaviour of vector-like objects, as opposed to sequence-like objects.

The following table gives some simple metrics concerning the refactoring and restructuring results.

	Original	Restructured	Restructured with method refactoring
Objects	17	15	16
Methods	322	317	396
Message sends(1)	2167	2194	2072
Message sends(2)	2167	2154	2041
Overriding methods	86	72	69
Reduction in size	-	0.6%	5.8%
Time(1)	-	13:28	30:43
Time(2)	-	5:46	6:53

The entry labelled 'Message sends(1)' includes a minor failing of the automatic reintroduction of resends, and consequent elimination of methods which have been introduced to replace the behaviour of the resend. In the restructuring without refactoring, the system was not able to remove one of these introduced methods, which unfortunately was a very large method. In the restructuring including refactoring, the system failed to remove the same very large method, and two smaller methods. It is important to stress that these minor imperfections do not affect the behaviour of the system, but can occasionally result in the existence of methods which can be removed with some minor modifications to other methods. The entry labelled 'Message sends(2)' is the result after manually performing the appropriate modifications, which requires a very small amount of programmer effort.

A small problem with replacing the original objects with the restructured ones was that, because vector, byteVector, mutableString and canonicalString have different sorts of mirrors than normal objects (see Section 2.3.7), only their traits 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 traits object. This point is further mentioned in Section 4.6.

6.2.2 The orderedOddballs hierarchy

This hierarchy includes the numbers and boolean traits objects. Although Guru was initially intended to restructure hierarchies including concrete objects, it can equally be used when given only traits objects, as in this case. The preserved objects are correctly identified as the leaf objects of the hierarchy to be restructured; it does not matter that these objects are not leaf objects of the Self system as a whole. Only traits objects 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. Also, in practice it would not be possible to include all the number objects in the Self image in a restructuring.

The original hierarchy is shown in Figure 6.8. The results of restructuring with and without refactoring, and before reintroducing overriding or resends, are both shown in Figure 6.9. Note that in this example, the hierarchies with and without refactoring have the same structure and so are shown together. The first numbers refer to the restructuring without refactoring, and the second numbers include method refactoring.

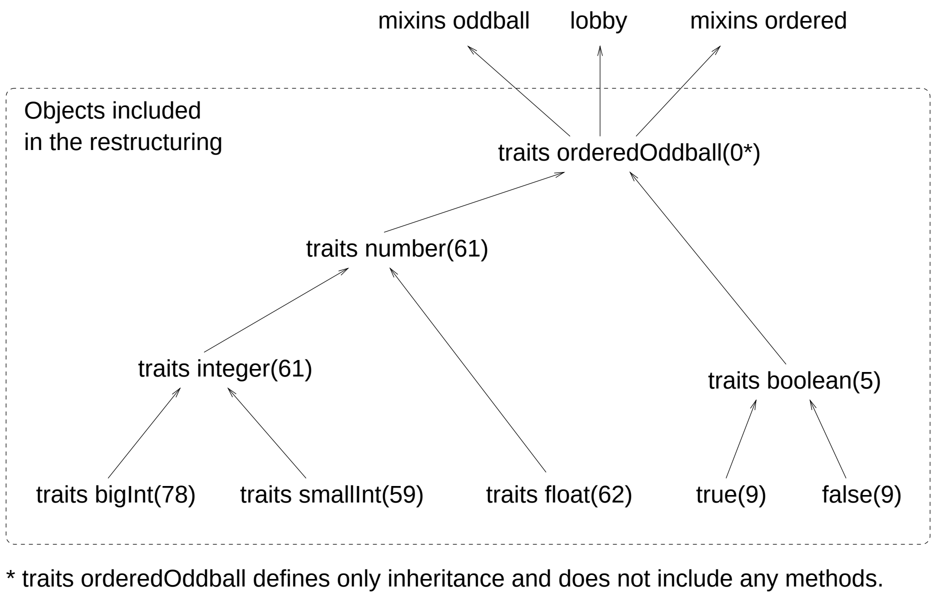


Figure 6.8: The original orderedOddballs hierarchy.

The results, with and without method refactoring, after reintroducing overriding and resends, are shown in Figure 6.10.

In the hierarchies (with and without refactoring) which do not include reintroduction of overriding, a traits object has been created (labelled traits orderedOddball) which *appears* not to define any slots. However, figures are labelled only with the number of *non-parent* slots, and traits orderedOddball defines only parent slots. The IHI algorithm does not create traits objects which define no features, and parent slots are not normally included as 'features' (see Section 4.4). However, the three parent slots defined by traits orderedOddball refer to parent objects outside the object to be restructured and so were included as 'features' (see Section 4.4). Therefore traits orderedOddball simply exists to share those three parent slots. In this example, the creation of this object is fortunate, as if

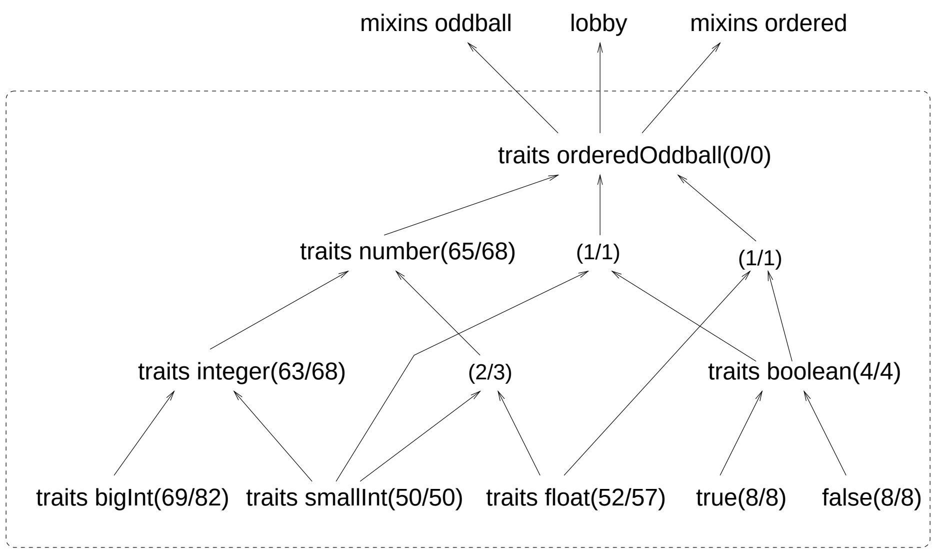


Figure 6.9: The restructured orderedOddballs hierarchy, with and without refactoring, before reintroduction of overriding.

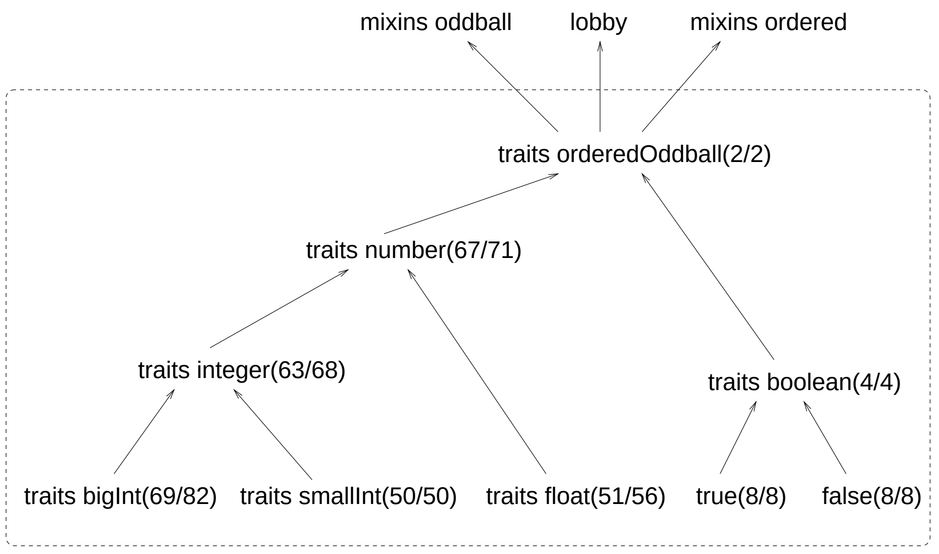


Figure 6.10: The restructured orderedOddballs hierarchy, with and without refactoring, after reintroduction of overriding.

it did not exist then the reintroduction of overriding would not be able to remove the two objects below it that define only one slot.

Notice that there is no difference in the structure of the final hierarchies (with and without refactoring) and the original hierarchy, 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 well designed initially, as such hierarchies are well understood, and it is a small hierarchy, in terms of the number of objects.

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.

	Original	Restructured	Restructured with method refactoring
Objects	9	9	9
Methods	324	302	329
Message sends	1528	1491	1449
Overriding methods	35	29	29
Reduction in size	-	2.4%	5.2%
Time(1)	-	2:59	6:09
Time(2)	-	1:25	1:19

6.2.3 The polygons hierarchy

The objects in the polygons hierarchy shown in Figure 6.11 are morphs used in the standard user interface and in a tool called the *structure editor*. The user interface representation of objects, called outliners, use expanderMorphs as toggles to display or hide slots. The user's hand is represented by a handMorph. In the figures, the following abbreviations are used: asRLPM for abstractSyntax rectiLinearPolygonMorph, asESFM for abstractSyntax structureEditorFrameMorph and EM for expanderMorph.

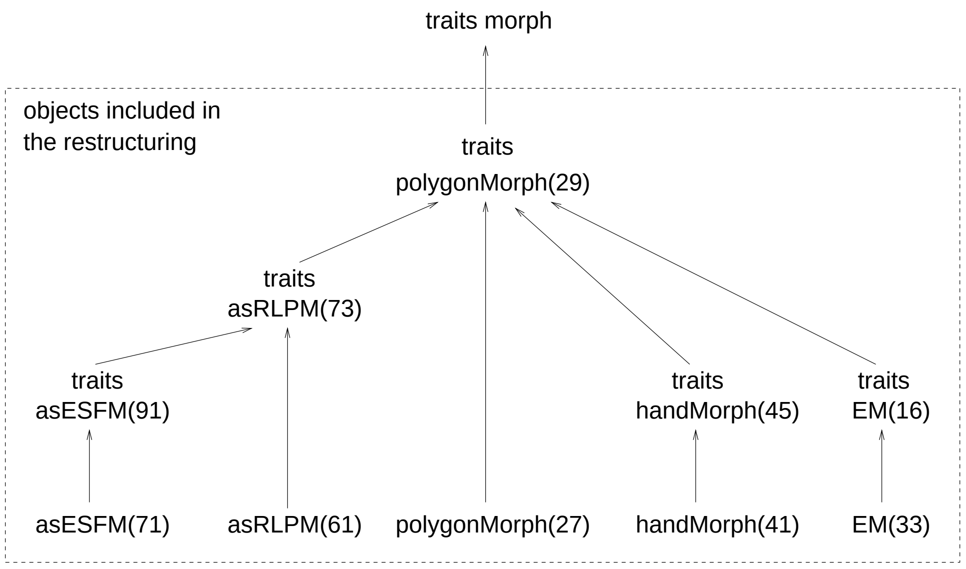


Figure 6.11: The original polygons hierarchy.

The results of restructuring, before and after reintroducing overriding and resends, are shown in Figures 6.12 and 6.13 respectively. The preserved objects were specified manually as only the leaf objects of the hierarchy.

The reintroduction of overriding failed to remove two objects, labelled A and B, so two slots were manually moved to object C, resulting in the hierarchy shown in Figure 6.14. This was then 'cracked' (see Section 4.3.3), producing the hierarchy of Figure 6.15.

Then it was discovered that 5 RR methods (see Section 4.4.4) had not been

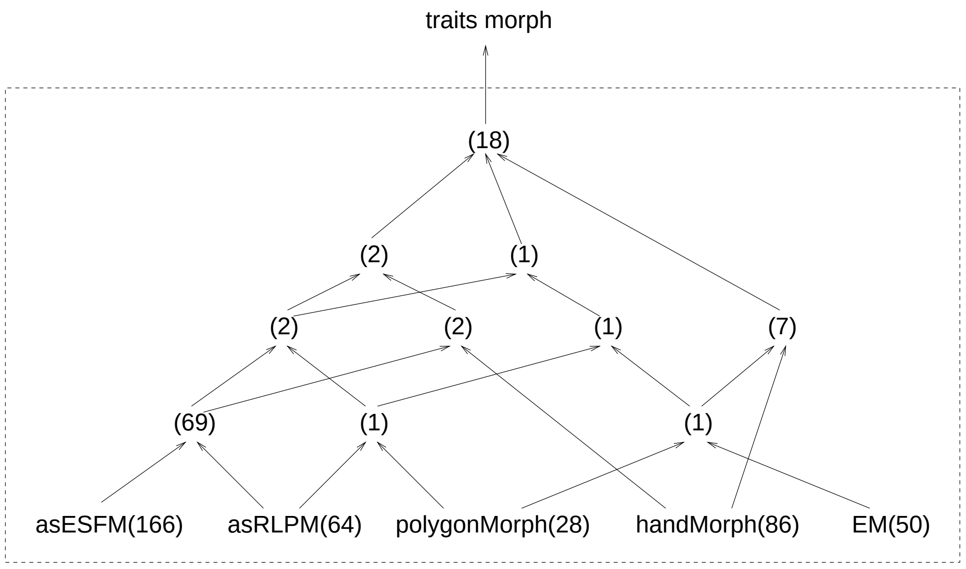


Figure 6.12: The restructured polygons hierarchy, before reintroduction of overriding.

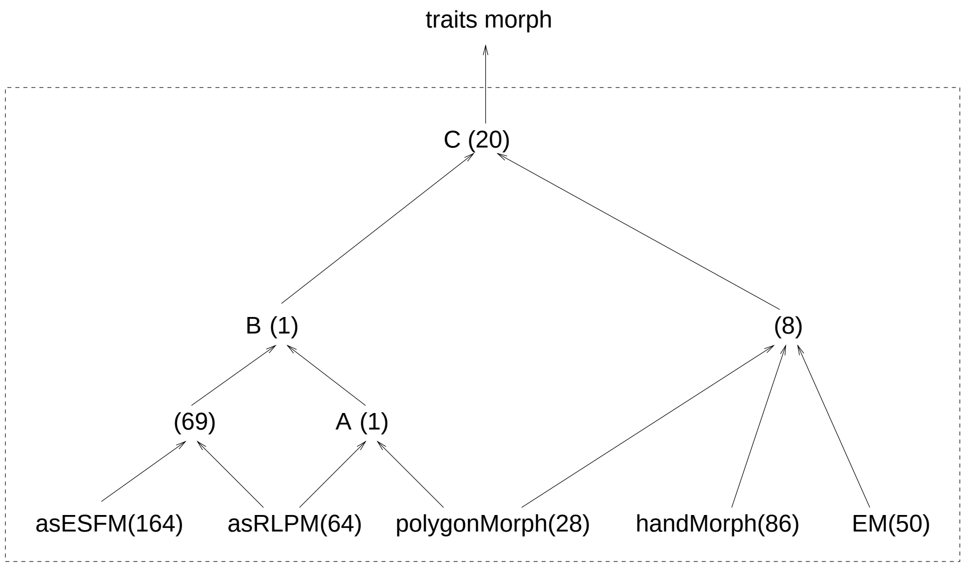


Figure 6.13: The restructured polygons hierarchy, after reintroduction of overriding.

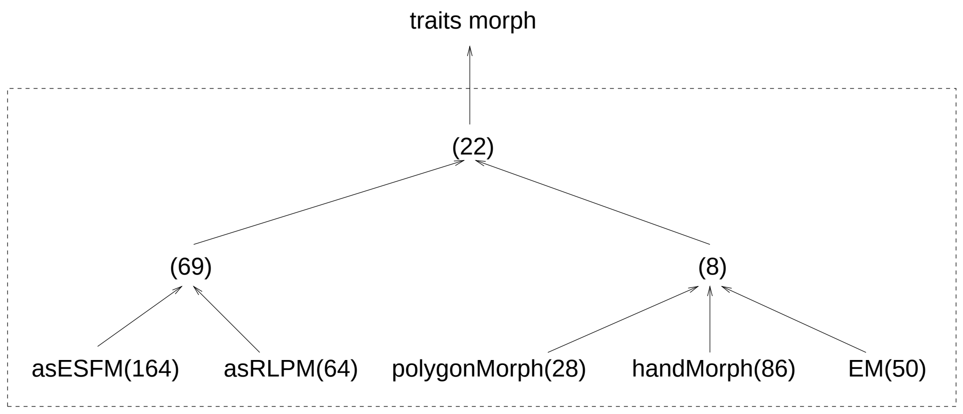


Figure 6.14: The restructured polygons hierarchy, including manual alteration.

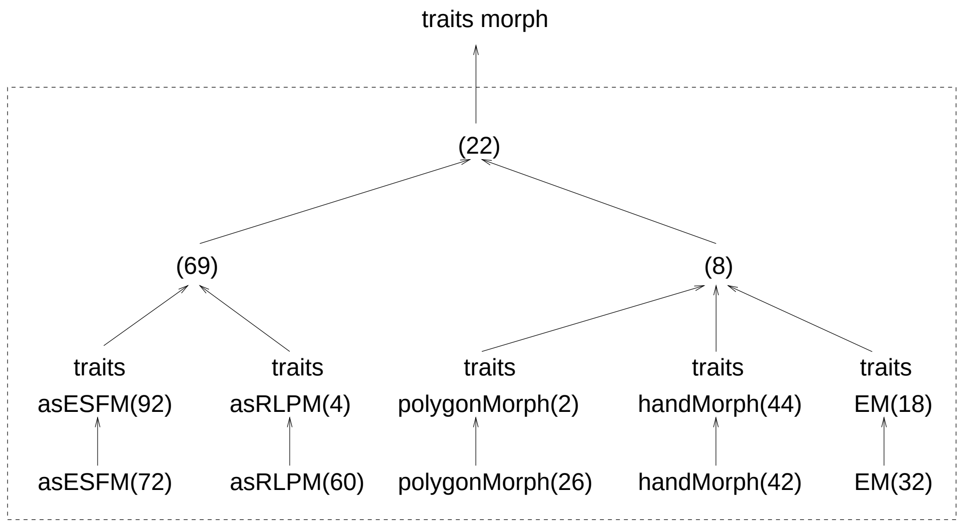


Figure 6.15: The restructured polygons hierarchy, with leaves ‘cracked’.

removed by the reintroduction of resends (see Section 4.7), so these were removed manually, including a small amount of alteration to other methods which included corresponding RR sends.

The manual alteration included moving 2 slots to a parent object. The reason for the problem with removing RR methods, and the corresponding alteration, is indicative of a general problem with the way that resends are handled. Consider the hierarchy shown in Figure 6.16. Hierarchies similar to this can easily result from automatic restructuring. The manually altered hierarchy in Figure 6.17 shows a better way of structuring the hierarchy. This alteration could be implemented automatically, but there may be better solutions which involve keeping track of which methods include RR sends, as discussed in Section 8.1.

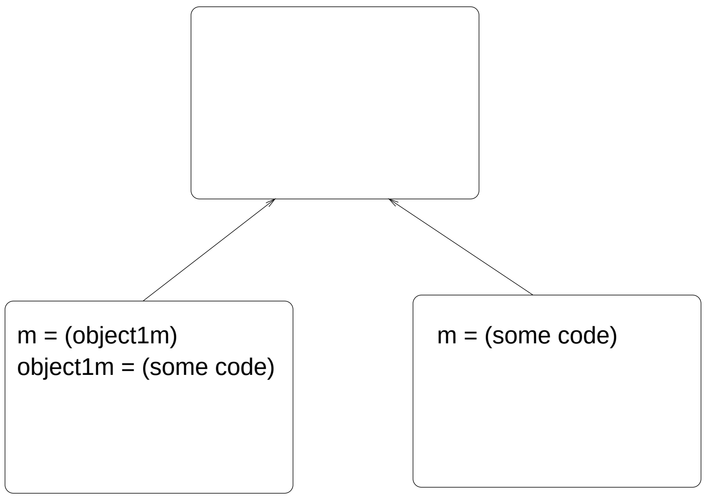


Figure 6.16: Hierarchy including RR method which cannot currently be removed automatically.

The results of applying restructuring with refactoring to the polygons hierarchy of Figure 6.11, before and after reintroducing overriding and resends, are shown in Figures 6.18 and 6.19 respectively.

Manually moving one slot because of the conservative nature of the reintroduction of overriding results in Figure 6.20. ‘Cracking’ the leaves of the hierarchy,

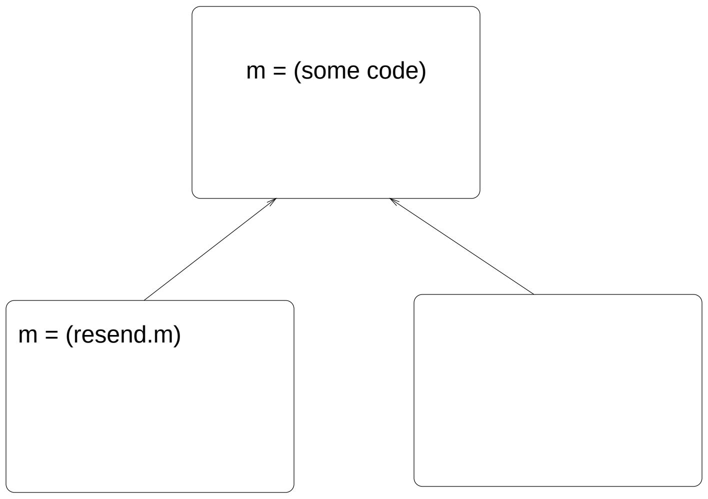


Figure 6.17: Hierarchy of Figure 6.16 after manual alteration.

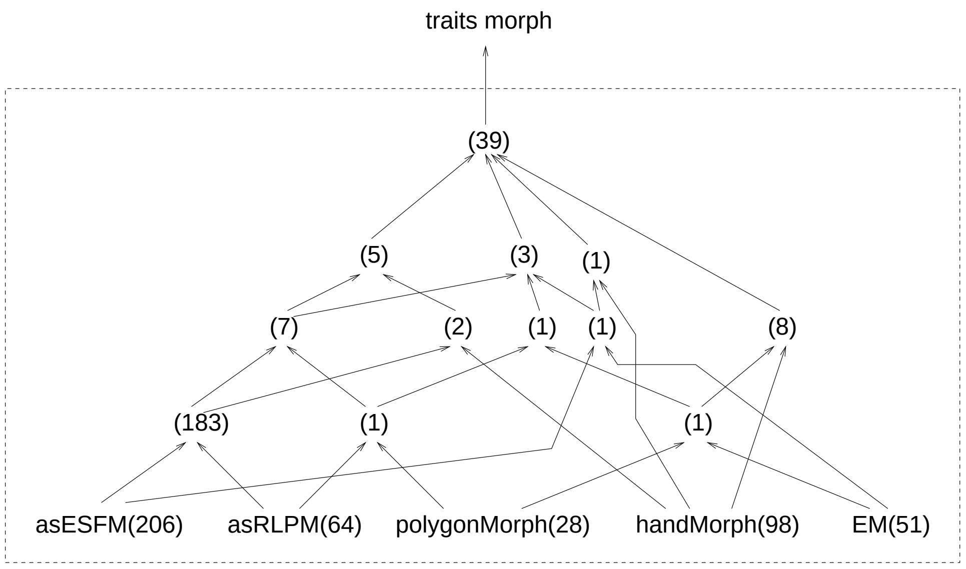


Figure 6.18: The restructured and refactored polygons hierarchy, before reintroduction of overriding.

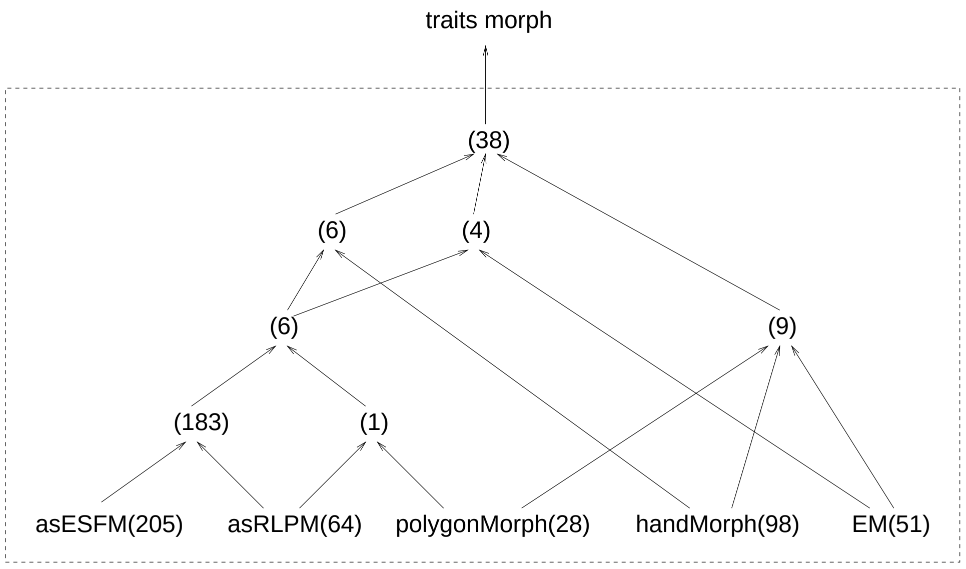


Figure 6.19: The restructured and refactored polygons hierarchy, after reintroduction of overriding.

and automatically creating 22 disambiguating methods to resolve introduced ambiguities (see Section 4.3.1), the hierarchy becomes that of Figure 6.21. In this case, the hierarchy is not very understandable, and more manual alterations would be needed to create a better hierarchy. If the slots in B, C and D are moved to A, then the structure (but not the number of slots defined in each object) would be the same as in Figure 6.15, which would be a large improvement.

The following table presents simple metrics for the original polygons hierarchy, and the hierarchies resulting from restructuring with and without refactoring.

	Original	Restructured	Restructured with method refactoring
Objects	10	13	16
Methods	236	240	465
Message sends(1)	3894	3972	3610
Message sends(2)	3894	3893	3577
Overriding methods	75	68	90
Reduction in size	-	0.02%	8.1%
Time(1)	-	6:02	1:28:24
Time(2)	-	8:14	11:18

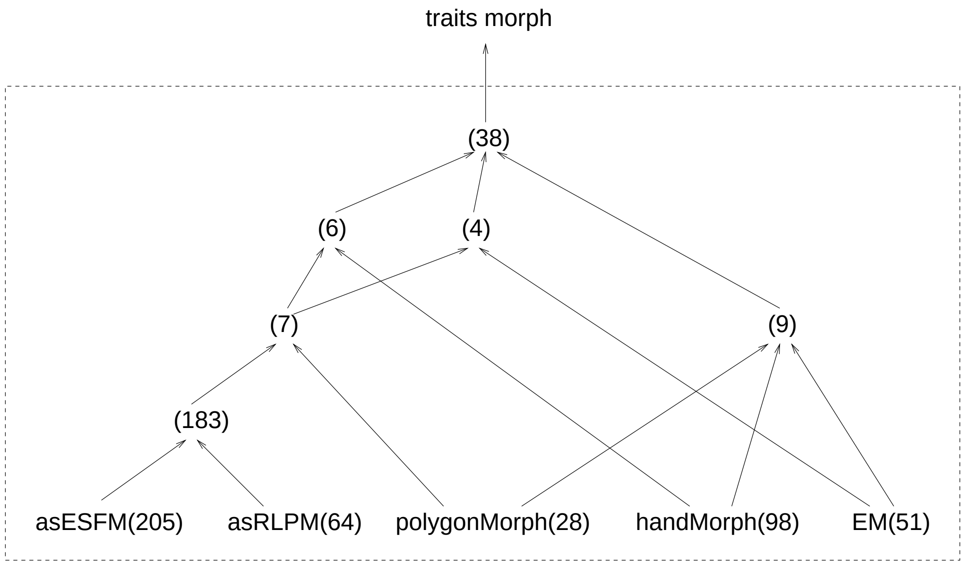


Figure 6.20: The restructured and refactored polygons hierarchy, after manually moving one slot.

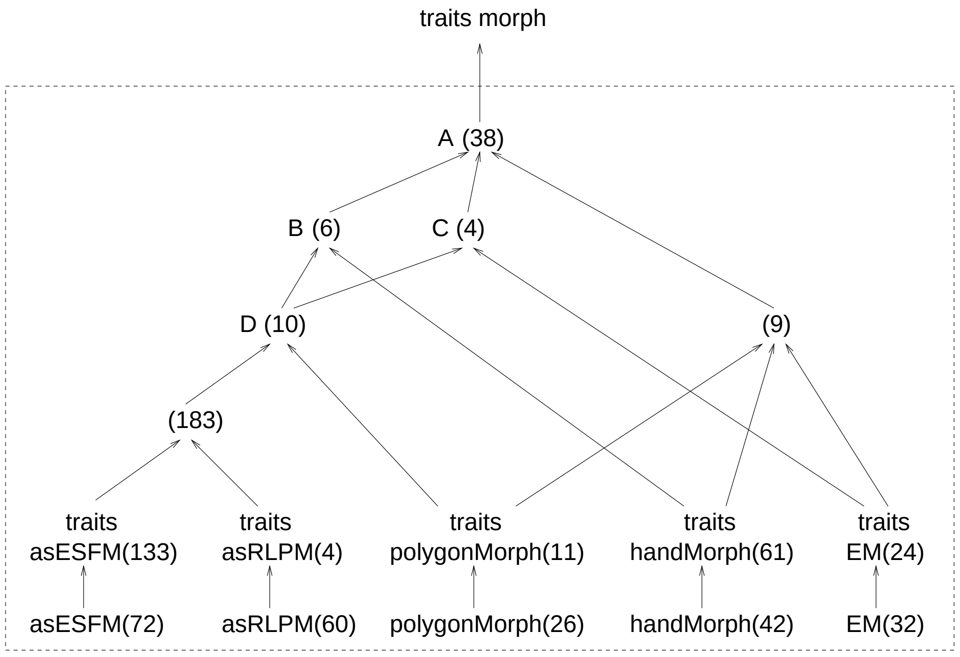


Figure 6.21: The restructured and refactored polygons hierarchy, after ‘cracking’.

The restructuring with refactoring took so long due to insufficient real memory in the machine used.

In this example, the amount of method refactoring is very high because many methods were much larger than usual for Self code.

6.2.4 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 6.22, the restructured hierarchy without refactoring and before reintroduction of overriding or resends is shown in Figure 6.23. The preserved objects were identified as the leaf objects and their immediate parents. The restructured hierarchy after reintroduction of overriding and resends is shown in Figure 6.24.

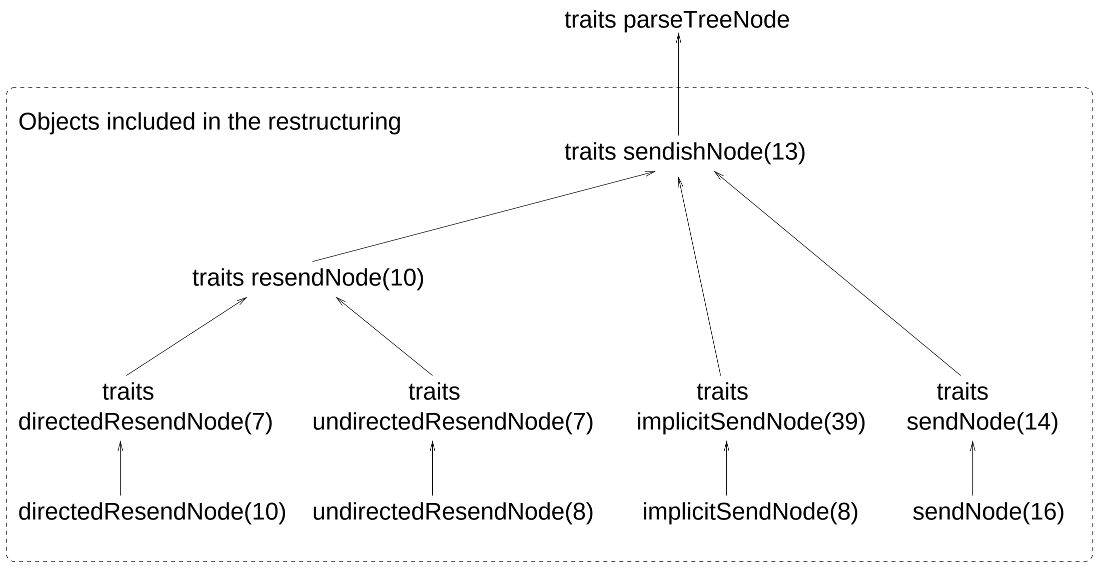


Figure 6.22: The original sendishNodes hierarchy.

The results including method refactoring are shown in Figures 6.25 and 6.26. The reintroduction of overriding failed to remove the object labelled 'X', because of being too conservative. Manually moving the two slots in this object to traits

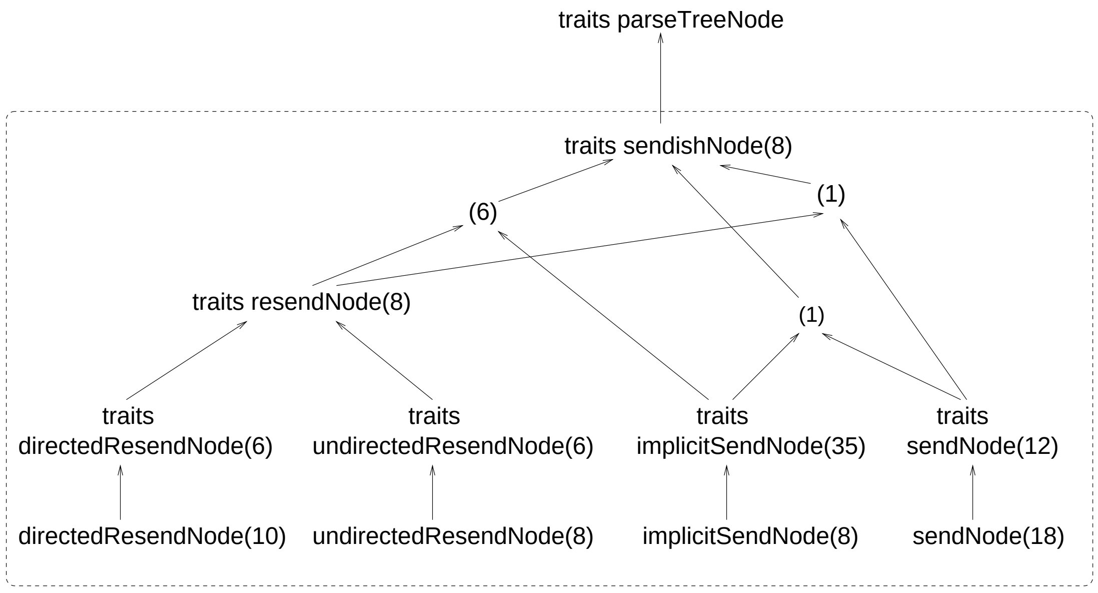


Figure 6.23: The restructured sendishNodes hierarchy, before reintroduction of overriding.

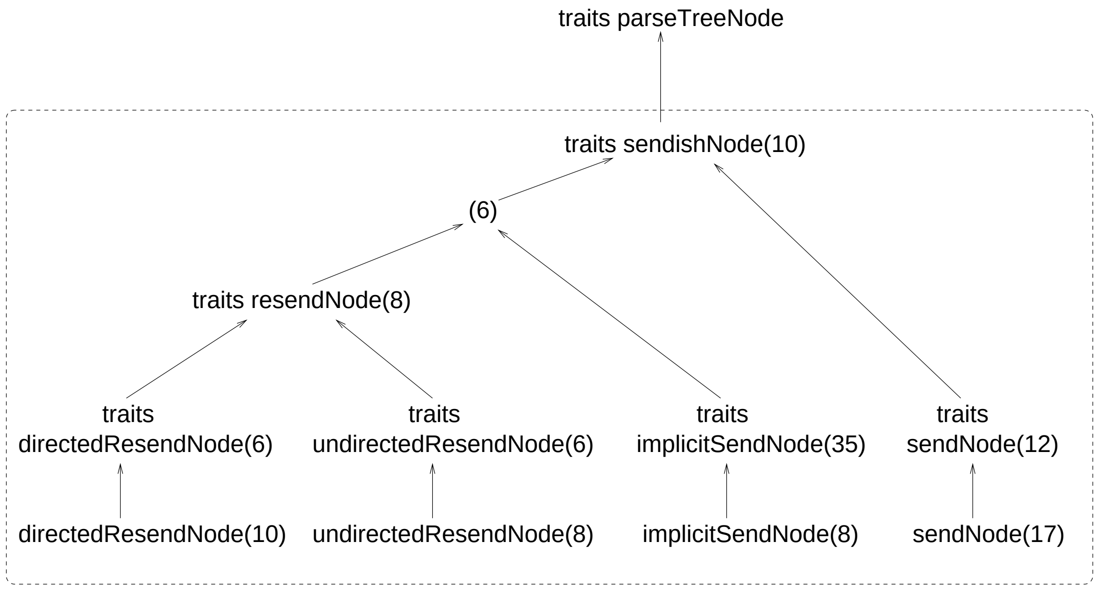


Figure 6.24: The restructured sendishNodes hierarchy, without refactoring of methods but with reintroduction of overriding and resends.

sendishNode, and automatically removing the resulting empty object, produced the hierarchy shown in Figure 6.27.

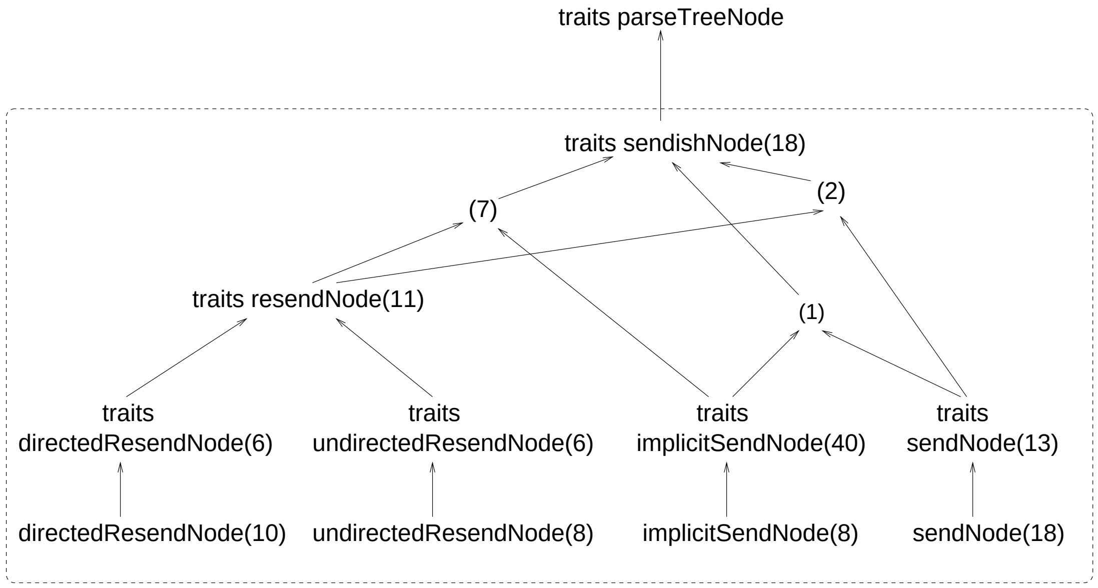


Figure 6.25: The restructured and refactored sendishNodes hierarchy, before reintroduction of overriding.

The restructured hierarchies are similar to the original hierarchy, with the difference in structure due to the discovery of a new traits object. This traits object can easily be understood as the shared behaviour 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 hierarchy, and the hierarchy restructured with and without refactoring.

	Original	Restructured	Restructured with method refactoring
Objects	10	11	11
Methods	80	75	96
Message sends	440	434	380
Overriding methods	41	33	33
Reduction in size	-	1.4%	13.6%
Time(1)	-	1:57	1:50
Time(2)	-	6:02	3:06

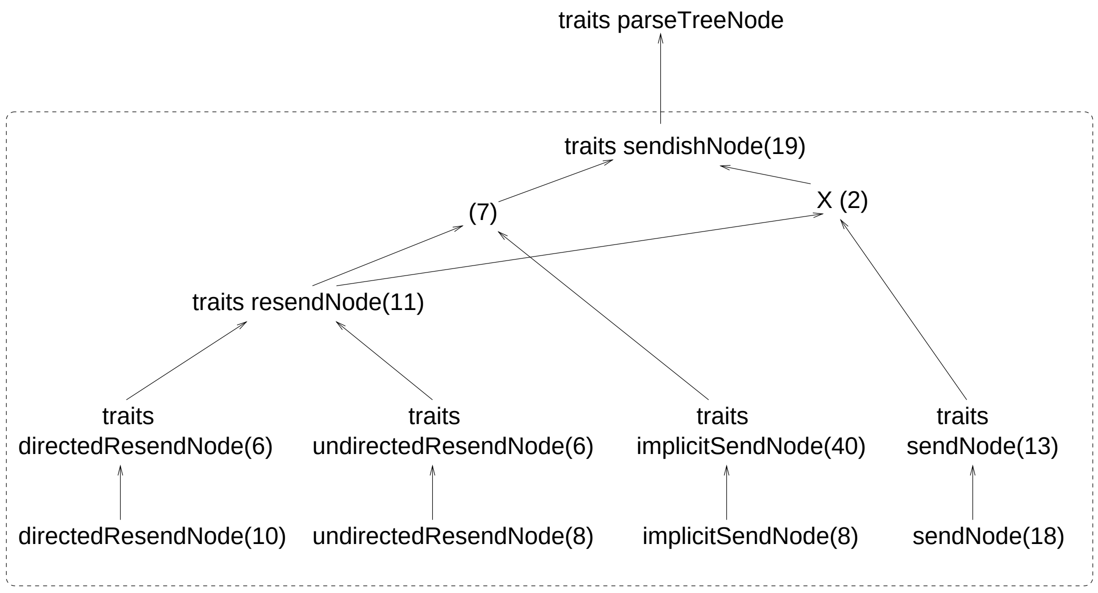


Figure 6.26: The restructured and refactored sendishNodes hierarchy, after reintroduction of overriding.

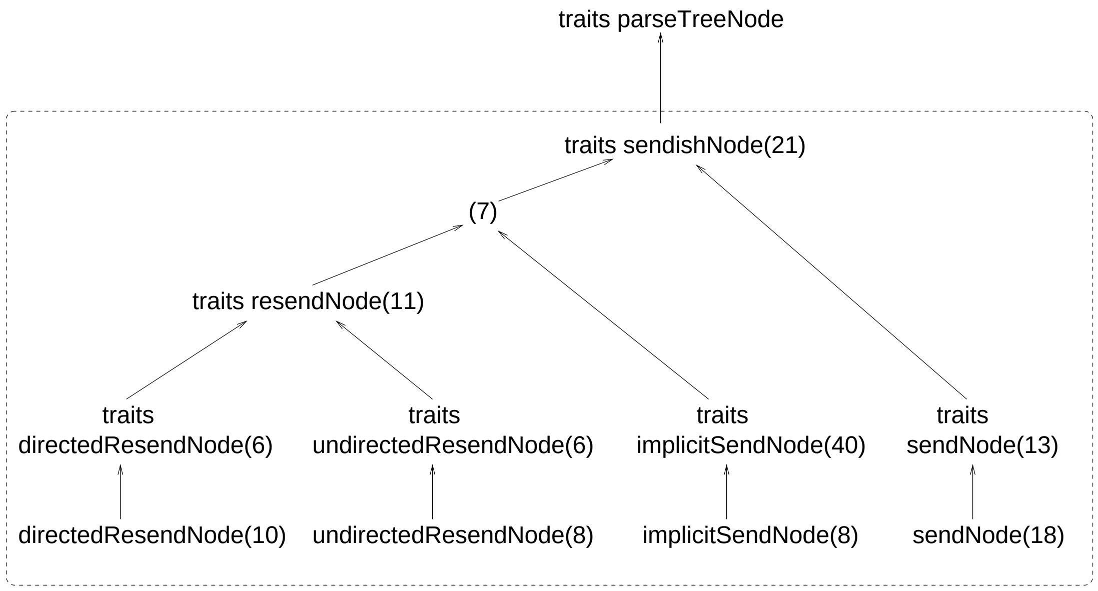


Figure 6.27: The restructured and refactored sendishNodes hierarchy, after manually moving two slots.

6.2.5 The samplers hierarchy

This hierarchy, shown in Figure 6.28, contains the prototypes of the different analysis button morphs described in Section 7.1.2. The following abbreviations are used in the figures: MSLM for mirrorSamplerListMorph, MSM for mirrorSamplerMorph, SM for samplerMorph, GM for getterMorph and MGM for mirrorGetterMorph. This hierarchy contains relatively few methods, but each prototype contains a large number of assignable slots. In this example, the preserved objects were manually specified as the leaves of the hierarchy.

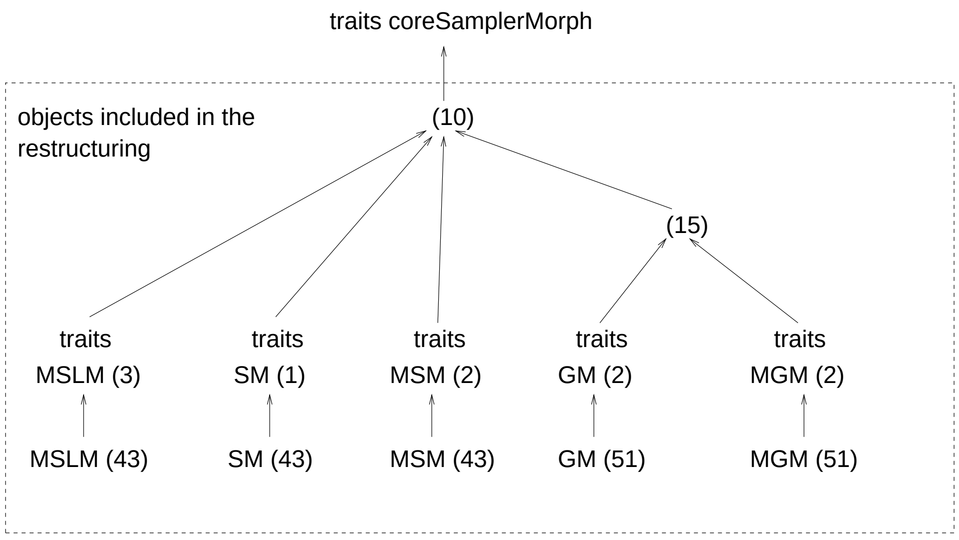


Figure 6.28: The original samplers hierarchy.

The results of restructuring with and without method refactoring, before reintroduction of overriding or resends, are shown in Figure 6.29. After reintroduction of overriding and resends, both with and without refactoring, the hierarchy becomes that shown in Figure 6.30. In these figures, the structure of the hierarchies is the same with and without refactoring, so these hierarchies are shown together. The first number refers to the result without method refactoring, and the second number includes refactoring.

'Cracking' the leaf objects (see Section 4.3.3) results in the hierarchy of Figure 6.31.

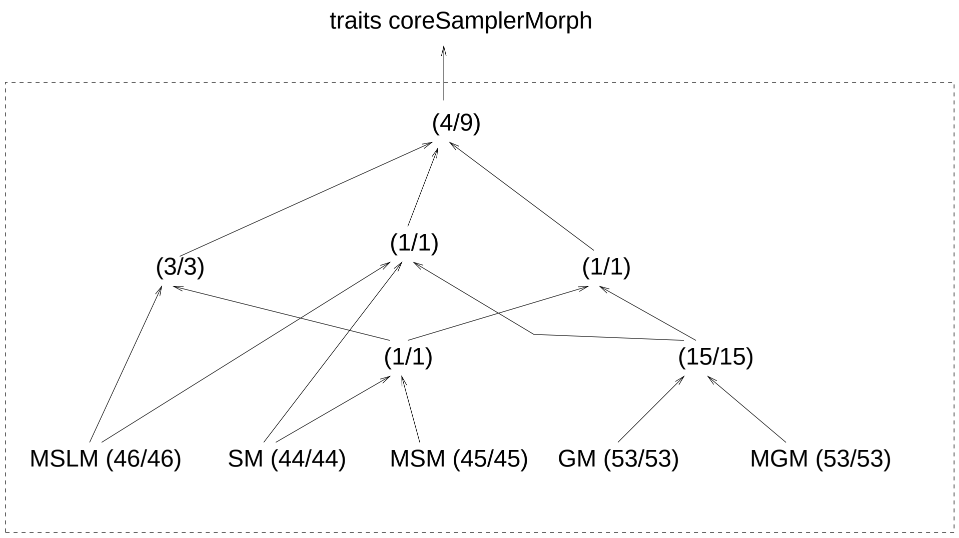


Figure 6.29: The restructured samplers hierarchy, before reintroduction of overriding.

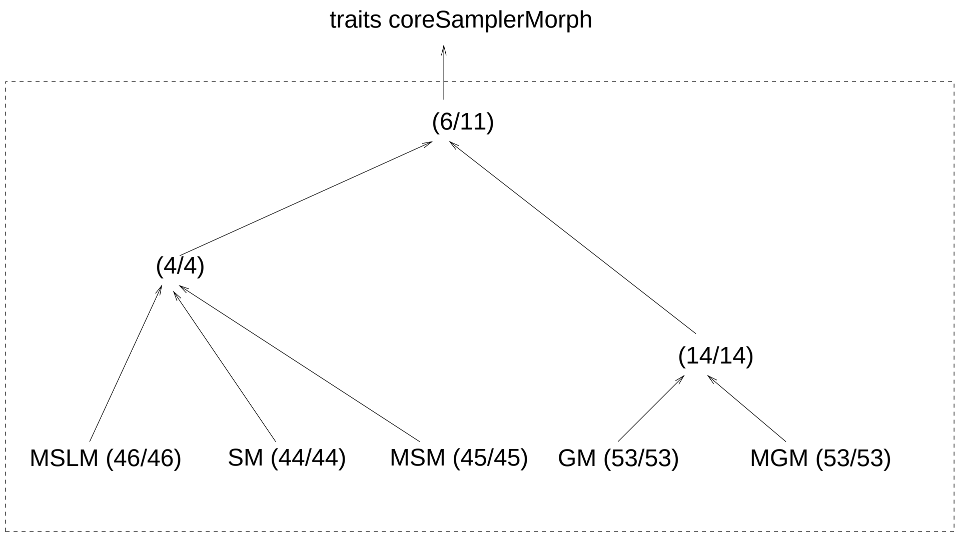


Figure 6.30: The restructured samplers hierarchy, after reintroduction of overriding.

In this example, there is very little method refactoring, because there are few methods, and the methods are small.

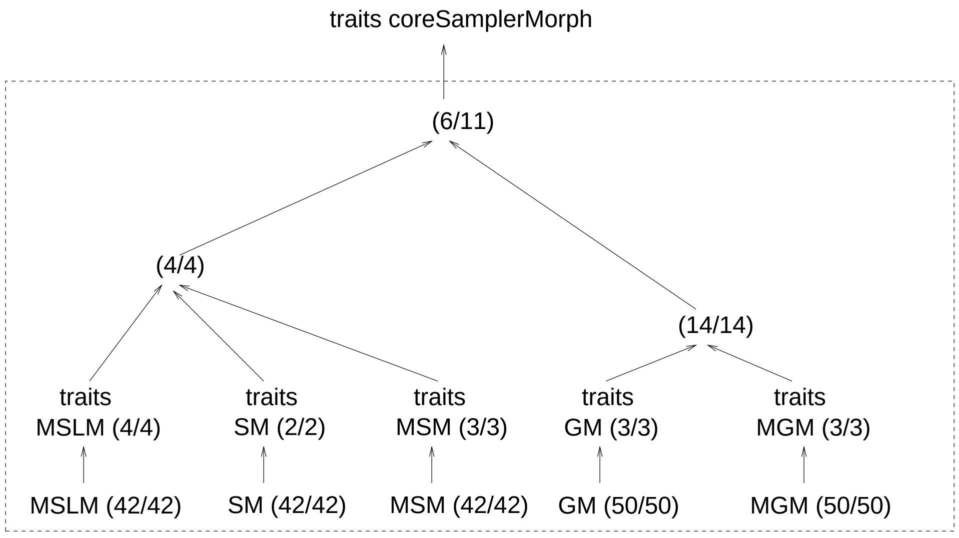


Figure 6.31: The restructured samplers hierarchy, after 'cracking' of leaves.

The following table is for the original samplers hierarchy, and the restructured hierarchies after 'cracking' of leaves.

	Original	Restructured	Restructured with method refactoring
Objects	12	13	13
Methods	35	34	39
Message sends	172	165	150
Overriding methods	31	26	26
Reduction in size	-	4.1%	12.8%
Time(1)	-	33	55
Time(2)	-	56	50

6.3 Analysis 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, in all except one case (polygons with method refactoring). 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 maximising sharing and minimising duplication of the features (mostly methods) of objects. Any other hierarchies which defined the same features for their objects, 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 all except one of the restructurings, single inheritance was sufficient to ensure no duplication of methods or factored expressions.

The results show the effectiveness of the reintroduction of overriding described in Section 3.5.2 at improving the structure of inferred inheritance hierarchies. Similarly, these results are evidence that at least a small amount of overriding is desirable for producing good inheritance hierarchy structures. Notwithstanding, the amount of overriding has been reduced compared to the original hierarchies. Note that it is the reintroduction of overriding which has caused the removal of nearly all occurrences of multiple inheritance.

It is interesting that method refactoring has not had more effect on the structure of the inheritance hierarchies produced. Partly, this is because of the effectiveness of the automatic reintroduction of overriding in removing very small traits objects; it must also be due to the details of the hierarchies restructured. While the approach taken maximises the amount of factoring of methods (within the limitations of Guru), the benefit of inferring new traits objects from factoring methods would appear, on the evidence presented, 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 objects, with *relatively* few factoring methods sharing expressions amongst methods of more than one object. 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 object will tend to have more similarities between them than methods in different objects.

The restructuring with method refactoring of the polygons hierarchy revealed a limitation of the current implementation. In resulting hierarchies, refactoring methods can sometimes be effectively duplicated, differing only in the names of arguments, because of the treatment of equality of local (non argument) slots of methods. If the two expressions, t1 b: c and t2 b: c are both repeated, where t1 and t2 both refer to local slots (not arguments) of their enclosing method, then two factoring methods will be created; newMethod1: t1 = (t1 b: c) and newMethod2: t2 = (t2 b: c). Note that this also means that expressions which should be replaced by factoring methods will not be replaced in the equivalent circumstances. For example, if the expression t1 b: c is repeated, but the expression t2 b: c is not repeated, then a factoring method for t1 b: c will be created, and occurrences of that expression replaced by the appropriate send to execute the factoring method. However, the expression t2 b: c will not be replaced in the same way, because the equality test for local (non argument) slots of methods relies on equality of name, which is more restrictive than necessary in many cases. Fortunately, such circumstances occur very infrequently in these examples, and do not have a large affect on the results of the experiments. In the case where t1 and t2 are both arguments of their enclosing methods, and are both in the same position in the list of arguments, there is no duplication.

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. The 'reduction in size' metric is smaller for the hierarchies which were expected to be well designed, and so it may be useful as a quality metric. The total 'reduction in size' figures for all the example hierarchies are; 0.8% reduction without refactoring and 7.4% reduction with refactoring. Therefore, method refactoring considerably improves the results, as measured by the 'reduction in size' metric. 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 in this chapter show that, despite the limitations described in Section 5.3, the system performs a considerable amount of refactoring. In the case of the indexables hierarchy, 77 expressions were factored out, and in the polygons hierarchy, over 200 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. To give an indication of the sort of expressions which were factored out from the orderedOddballs, indexables and polygons 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 8.1):

```
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))

newMethod1644 = (nextPos: (((newMethod1495) - borderwidth) @
                                                  (newMethod1497)))
newMethod1495 = (xList at: urlnx)
newMethod1497 = (yList at: urlnx)
```

```
newMethod1466 = (labelMorph copy)
newMethod1577P: vp1 P: vp2 = ((vp1 < 0) && [vp2 > 0]) ||
                              [(vp1 > 0) && [vp2 < 0]]))

newMethod1467P: p P: p1 P: pb = ((pb - p) crossProduct: (p1 - p))
newMethod1468P: p P: pa P: pn = ((pa - p) crossProduct: (pn - p))
```

The last two methods, newMethod1467P:P:P: and newMethod1468P:P:P:, give an example where different ordering of arguments makes a difference to the meaning of two otherwise similar methods. A possibly better way of refactoring these methods, which has not been implemented, would be to define newMethod1468P:P:P: as:

```
newMethod1468P: p P: pa P: pn = (newMethod1467P: p P: pn P: pa)
```

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 potential message sends (measured statically, rather than the number of actual message sends that will occur at run-time), 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 potential message sends); hence the total number of potential message sends will be reduced by 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.

In the sendishNodes hierarchy, some methods were written in a particular way in order to maximise 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.

6.4 Summary

This chapter has presented the results of applying Guru to realistic examples. These results show that the inheritance hierarchy structures produced are those expected of well designed hierarchies. Furthermore, the refactoring of shared expressions from methods has improved the amount of code reuse, while having no negative effect on the inheritance hierarchies produced (in all except one example). The behaviour of programs has been shown to be preserved by replacing objects fundamental to the correct running of the Self programming environment with their restructured equivalents (with and without refactoring of methods), with no change in the behaviour of the Self system.

Chapter 7

Complementary Tools

This chapter describes tools which are useful in addition to the hierarchy restructuring and method refactoring described in Chapters 4 and 5.

Restructuring tools will only be useful if they are easy to use. Section 7.1 describes three different approaches to user interaction with restructuring tools. A user interface has been developed for using the restructurings described in Chapters 4 and 5, as well as for other simpler restructurings which require different user interaction. Furthermore, simple static analyses are also considered because, even if the inheritance hierarchy structure and the factoring of methods in a system are satisfactory, a programmer may want to know about aspects of a system without actually altering the system, for example, to evaluate the results of a restructuring, to assist in program understanding, or to help in manual restructuring.

A system may be well structured, but nevertheless include code which is not necessary for a particular application. Section 7.2 describes an approach to removing code which is unnecessary for an application, that is, separating application programs from the rest of the Self programming environment.

7.1 User interaction with restructuring and analysis tools

This section describes three alternative approaches to using restructuring and analysis tools in the Self programming environment. The approaches can be applied to similar programming environments for other languages. Each approach is suitable for different restructurings and analyses; similarly, each restructuring or analysis may be most suited to a particular approach to user interaction, or may be suitable for more than one approach.

The three approaches are:

- Having as little user interaction as possible.
- Requiring the user to specify the restructurings and analyses to perform, and the objects on which to perform them.
- Integrating restructuring and analyses into the programming environment, so that the system becomes proactive rather than reacting to user requests.

The analyses and restructurings considered are described below. The approaches to integrating these into the programming environment could be applied to other sorts of analyses and restructurings; those described are not considered to be comprehensive.

The analyses implemented were:

'not sent' Whether the name of a slot is ever sent as a message. A slot which is 'not sent' may be unnecessary.

'not fully implemented' Whether all of the messages potentially sent in a method have at least one implementor. The presence of such potential message sends may indicate that the method contains an error, is obsolete, or that there is at least one method missing.

'overrides' Whether the slot overrides an inherited slot. This information is provided by some browsers for languages other than Self.

'overrides unnecessarily' Whether the slot overrides an inherited slot with the same definition.

'ambiguous' Messages which are ambiguous for an object. That is, messages which when sent to an object would result in an 'ambiguous selector' run-time error.

'place holders' The automatic identification of 'place holder' methods, using the algorithm described in Section 4.4.2.

The use of type inferencing [Agesen95] would allow more sophisticated analyses to be possible, for example, checking for slots containing message sends which are not implemented for the type of receiver determined.

Some simple metrics were also implemented, but are not described here as the study of metrics is an extensively researched area, and descriptions of metrics and their uses are presented in [Chidamber94, Möller93]. Simple metrics such as the number of slots in an object, the number of statements in a method, and the average number of statements in methods in an object are all straightforward to determine. Some simple metrics were used in Chapter 6 to evaluate the results of the restructuring and refactoring performed by Guru.

The restructurings implemented were:

- The inheritance hierarchy restructuring with and without refactoring of expressions from methods, as described in Chapters 4 and 5.
- Removal of 'overrides unnecessarily' slots.
- Moving slots which have the same definition to a common parent object. This is called the 'moving same sibling slots' restructuring, and is described in the following subsection.

- Splitting an object into its assignable slots and a separate object for its non-assignable slots. This is called 'cracking', and is described in Section 4.3.3 for use on the leaf objects of restructured hierarchies, but can also usefully be applied to many other objects.

The 'moving same sibling slots' restructuring

This restructuring removes slots which have the same definition and which also share a common parent. It is similar to the restructuring described in Section 3.5.1 for reintroducing overriding, but has subtle and important differences. Rather than moving a slot higher in the inheritance hierarchy in order to reflect a 'default' implementation of a slot which is to be overridden by 'non-standard' implementations of the slot, the 'moving same sibling slots' restructuring removes duplicated slots. This is achieved by moving slots which are defined the same in objects which share a common parent into that common parent, and removing those duplicated slots.

Figure 7.1 shows an example of this restructuring. The 'moving same sibling slots' restructuring results in one of the (equivalent) implementations of m in A, B or C being moved to E (and being removed from the other two). The restructuring described in Section 3.5.1 would have moved the implementation of m in D into E, as this is the 'default' implementation; it results in no decrease in the duplication of slots.

In the current implementation of the 'moving same sibling slots' restructuring, only hierarchies two levels deep are considered. Deeper hierarchies could be considered, but in practice most of the benefit of this restructuring is to be found in hierarchies one or two levels deep, and further depth makes the restructuring too slow without removing substantially more slots.

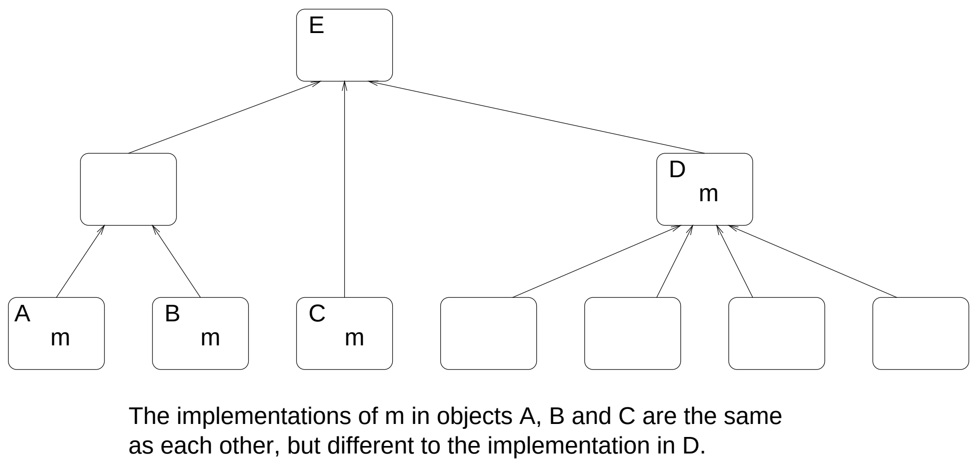

E
D
m
A
m
B
m
C
m
The implementations of m in objects A, B and C are the same as each other, but different to the implementation in D.

Figure 7.1: The most duplicated implementation is to be overridden by the less duplicated implementation(s).

7.1.1 Using as little programmer interaction as possible

Rather than using a programmer's time in specifying which restructurings and analyses to perform, and which objects to perform them on, one approach is to have the system perform certain restructurings and analyses with the minimal programmer involvement. For example, the analyses can be applied to any collection of slots or objects, such as all the objects and slots in a module (see Section 2.3.2), or all the objects and slots in the system.

The application of restructurings with minimal programmer interaction has limitations. In order to apply a restructuring without any programmer involvement, for example on an entire Self image, the restructuring *must* perform exactly the sort of restructuring that a programmer will accept as an improvement, and must not produce any unexpected results. Such restructurings are therefore simple and will have limited benefits. Of the restructurings proposed earlier, only removal of 'overrides unnecessarily' and 'moving same sibling slots' can safely be applied without user interaction. In fact, these restructurings are so 'safe' in terms of result and programmer understanding that they could be applied automatically 'behind the scenes' without even being requested.

More sophisticated and potentially beneficial restructurings, such as those described in Chapters 4 and 5, cannot safely be applied without programmer interaction, that is they cannot be applied globally or on all the objects in a module, as the results cannot be *guaranteed* to be understandable to a programmer. Therefore, the restructurings of Chapters 4 and 5 require the user to manually specify which objects to include in a restructuring. Furthermore, such restructurings may take too long to be feasible to apply on very large numbers of objects. Even applying the much simpler 'overrides unnecessarily' and 'moving same sibling slots' restructurings on an entire Self image took several hours on a Sparc 2.

The analyses which identify potentially unnecessary slots, and the 'overrides unnecessarily' and 'moving same sibling slots' restructurings, were applied to all the objects and slots in the standard Self system; the results are presented below. (The 'moving same sibling slots' figure refers to the number of slots removed by this restructuring.)

Proportion of slots in the system	identified by the analysis
13.5%	Not sent
1.0%	Not fully implemented
0.3%	Not sent and not fully implemented
0.4%	Overrides unnecessarily
0.65%	moving same sibling slots

Only slots which are 'well known' (see Section 2.3.2) were considered.

It might be assumed that 'not sent' slots can be safely removed as being unnecessary. However, it is impossible to determine all the messages sent in a Self system, because of computed selectors (see Section 2.3.7). Furthermore, 'not sent' slots may be necessary if they are used reflectively, rather than through message sends; for example, _Mirror understands: 'aMessage' (see Section 2.3.4). Even if the 'not sent' slots could be determined accurately, many slots exist which are not currently necessary, but are useful for future reuse and so should not be removed. Therefore, identifying 'not sent' slots as potentially removable has limited value.

Similarly, 'not fully implemented' slots cannot safely be removed as, rather than being obsolete, they may simply contain an error which should be corrected. Furthermore, it is quite possible for a method to contain code which is never executed; messages which have no implementor will not cause an error if they are never sent.

Methods which are both 'not fully implemented' and 'not sent' are better candidates for removal, but even these cannot be safely removed automatically, and user interaction is needed.

The removal of 'overrides unnecessarily' slots and 'moving same sibling slots' both provide only a very small improvement. Nevertheless, the number of such slots is larger than had been expected.

Two slots were identified which would have caused an ambiguity if the relevant message had been sent. One of these slots was commented as 'should probably be thrown away'. The percentage of all slots that this represents is insignificant, so this analysis does not appear to be of much value in general.

None of these very simple restructurings are sophisticated enough to have a large impact on program quality, so for better results more user involvement is needed. The following section describes a user interface for providing such user interaction for supporting more sophisticated restructurings, such as those described in Chapters 4 and 5 on user chosen collections of objects.

7.1.2 Programmer driven interaction

The Self user interface is based on direct visual representation and manipulation of objects [Smith95]. Although the user interface is simple and elegant for most programming requirements, certain simple tasks are not well supported.

For example, in order to find the results of the analyses described above for a particular object, the programmer needs to send a message to the reflective representation of that object (called its 'mirror'; see Section 2.3.4). In order to send a message to an object, an *evaluator* can be opened for the object, and a message sent by typing the message into the evaluator and pressing the 'Evaluate' button. The object resulting from the message sent to that object is then attached to the user's hand. Similarly, a message can be sent to an object's mirror either by opening an evaluator on the mirror object, or using an evaluator on the object and sending a message to the object which will result in the object's mirror, such as the _Mirror primitive.

For example, consider finding the slots for an object which override inherited slots. An evaluator can be opened on the object, and the expression '_Mirror overrides' evaluated. The resulting object will be a set of names of the slots which override inherited slots. In the standard Self image, to examine the contents of a set is not very convenient; the easiest way is to evaluate the message 'asVector' for the set object. This vector can be 'opened', giving direct access to all the objects contained in the set.

In order to provide a user interface for the simple restructurings and analyses described earlier, and for the restructurings described in Chapters 4 and 5, analysis button morphs were created. They provide a fast and convenient way to access the results of message sends to objects or their mirror objects. These morphs are parameterised by a message to send to the object (or its mirror) onto which they are dropped or moved. This message will be called the *request* and the object sent the message will be called the *target*. In contrast to the buttons provided in the standard Self user interface, the target of analysis buttons is the object that the button is over, rather than a specific object set using the middle button menu (or in the code of a method). Also, those analysis buttons which display their result (rather than return an object to the user) are not pressed in order to display their result; rather, they display their result whenever they are placed over a target object.

The following types of analysis button morphs have been implemented:

sampler displays the printString of the object resulting from sending the request to the target object.

getter attaches the object resulting from sending the request to the target object to the user's hand.

mirrorSampler the same as sampler, except the request is sent to the mirror of the target object.

mirrorGetter the same as getter, except the request is sent to the mirror of the target object.

mirrorSamplerList displays the printStrings of the objects in the collection resulting from sending the request to the mirror of the target object.

A samplerList analysis button morph (the same as mirrorSamplerList except sending the request to the object rather than its mirror) was not implemented as it was not found to be needed.

Figures 7.2, 7.3, 7.4 and 7.5 show a selection of these analysis button morphs.

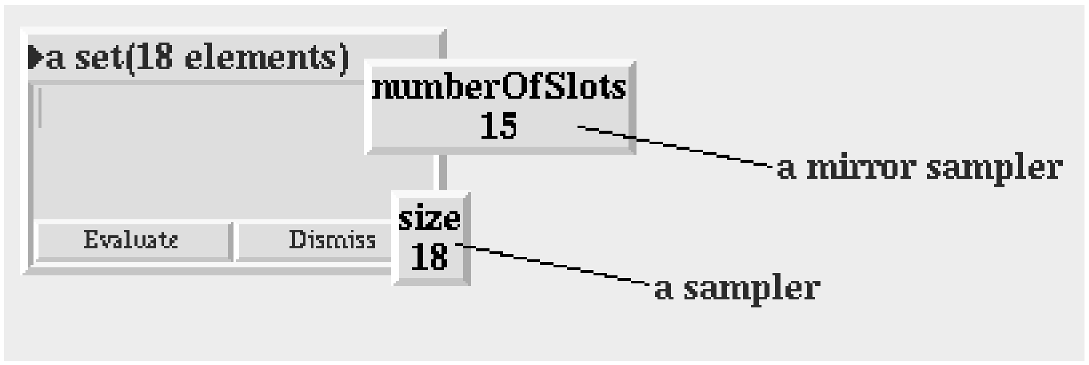


Figure 7.2: sampler analysis button morphs.

The analyses and the simpler of the restructurings described earlier can be used through analysis button morphs.

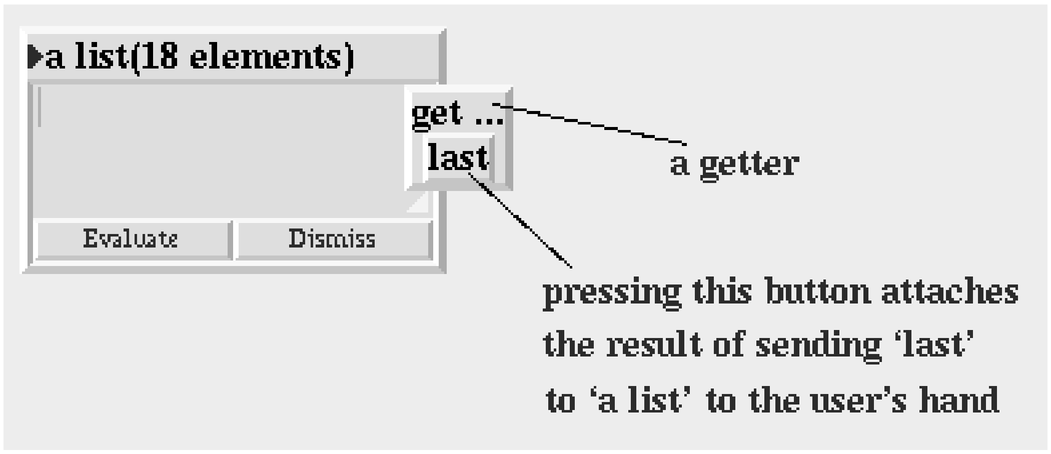

a list(18 elements)
get ...
last
Evaluate
Dismiss
a getter
pressing this button attaches
the result of sending 'last'
to 'a list' to the user's hand

Figure 7.3: getter analysis button morph.

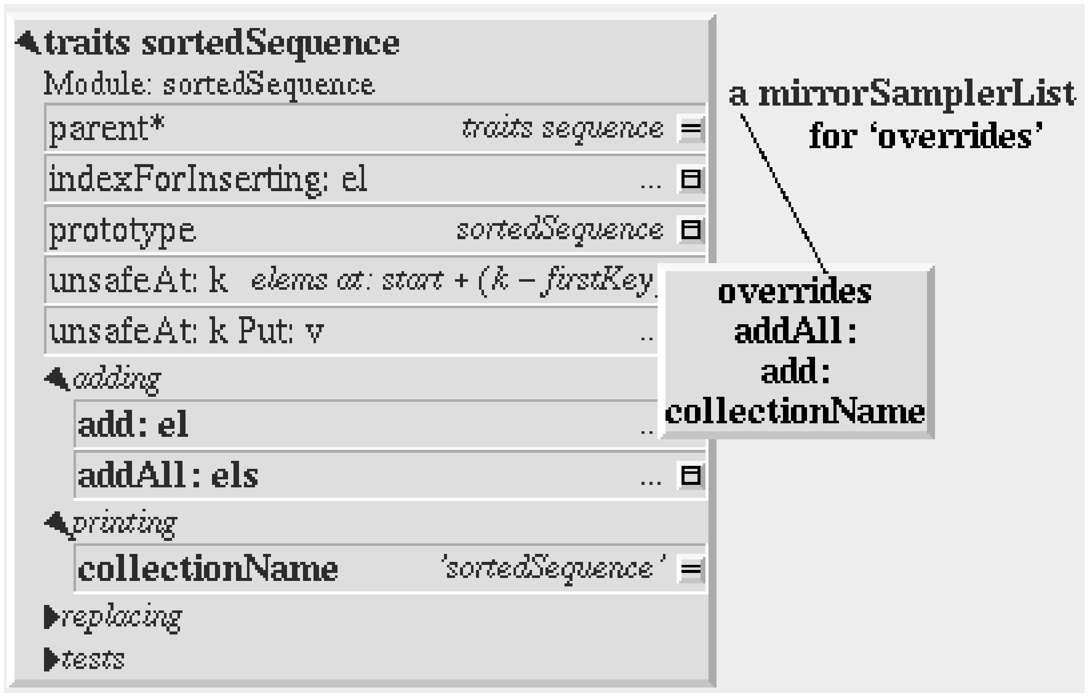


Figure 7.4: mirrorSamplerList analysis button morph.

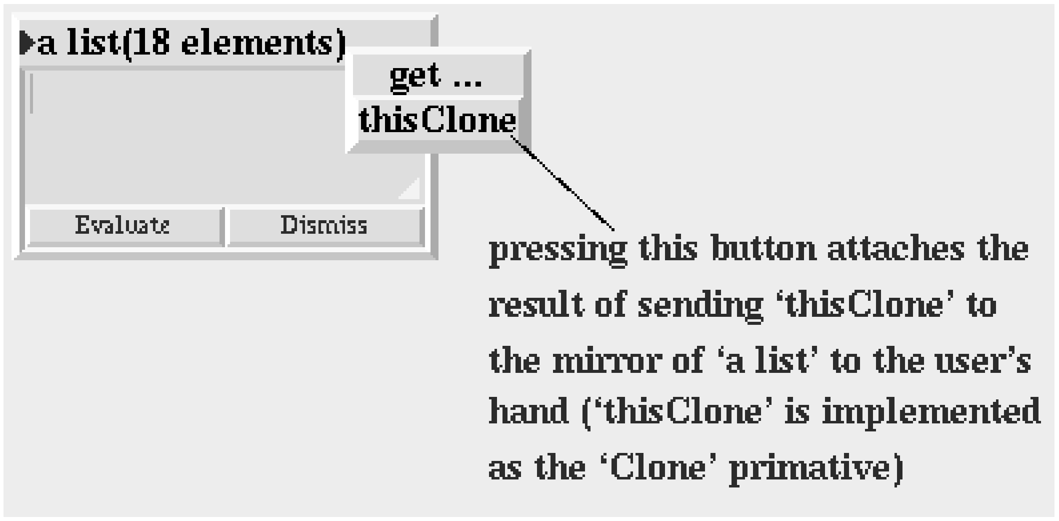

a list(18 elements)
get ...
thisClone
Evaluate
Dismiss
pressing this button attaches the result of sending 'thisClone' to the mirror of 'a list' to the user's hand ('thisClone' is implemented as the 'Clone' primative)

Figure 7.5: mirrorGetter analysis button morph.

If the target does not understand the request then, in the case of sampler morphs, text to indicate that the request is not understood is displayed. Similarly, in the case when a getter morph is dropped onto a target which does not understand the request, the string object indicating that the request is not understood is attached to the user's hand.

New sampler and getter morphs can be created using a creator morph, which is parameterised by the request message of the sampler or getter morph to be created. When a creator morph is pressed, the appropriate new sampler or getter morph is attached to the user's hand.

A special getter morph, called a collectionGetter, has been constructed for fetching objects out of collections. It allows the objects in a collection to be brought onto the screen, one at a time, without actually removing them from the collection. It is more sophisticated than other getter morphs in that each collectionGetter contains a collection of the objects that it has already fetched from a collection, so that it fetches all of the objects out of a collection without fetching the same element more than once. Figures 7.6 and 7.7 show how a collectionGetter morph is used and created. (The safeGetKeys button returns a collection containing the keys of the target, which useful if the target is a dictionary.)

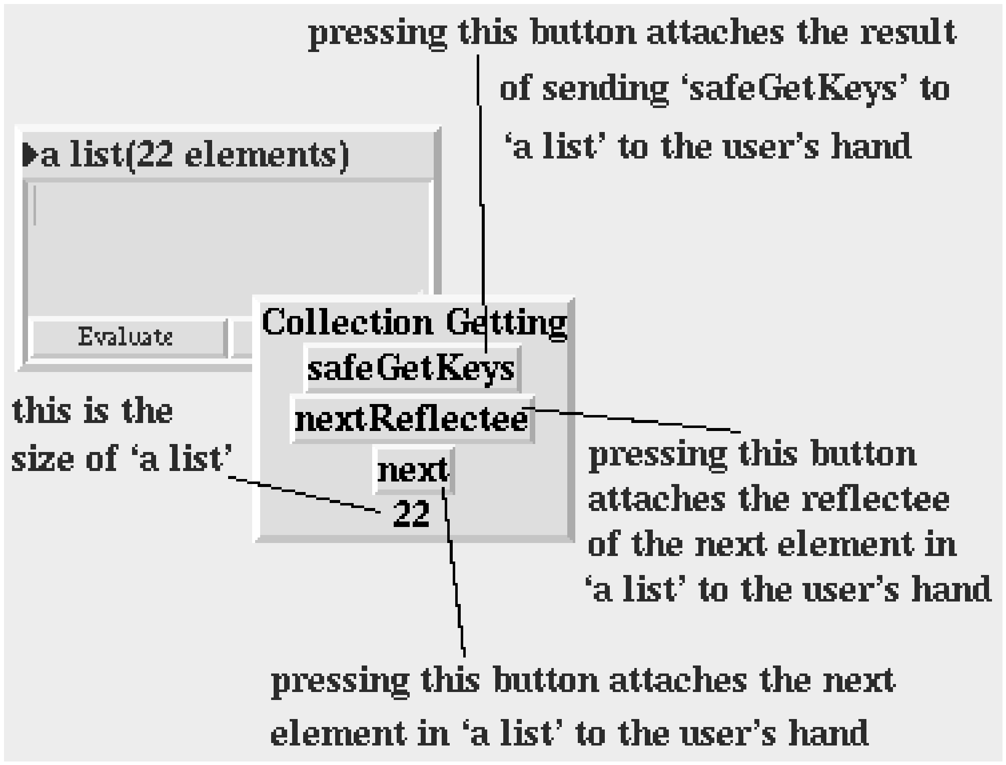

pressing this button attaches the result of sending 'safeGetKeys' to 'a list' to the user's hand
▸a list(22 elements)
Evaluate
Collection Getting
safeGetKeys
nextReflectee
next
22
this is the size of 'a list'
pressing this button attaches the reflectee of the next element in 'a list' to the user's hand
pressing this button attaches the next element in 'a list' to the user's hand

Figure 7.6: User interface for inspecting collections.

Using the restructuring and refactoring described in Chapters 4 and 5 is also inconvenient using the standard Self user interface. In particular, in order to specify the objects to restructure, the user must create a suitable empty collection, and then add each object individually using an evaluator. In order to refer to some objects in an evaluator, long strings of text are often necessary. To provide a user interface for the restructuring and refactoring described in Chapters 4 and 5 another morph based on analysis button morphs, called an ihr collector, was constructed for specifying the objects to be included in a restructuring. An ihr collector is created and attached to the user's 'hand' by clicking on a ihr collector creator morph. The ihr collector is then dropped on an object to add that object to the collection of objects to be restructured. The ihr collector can then be 'picked

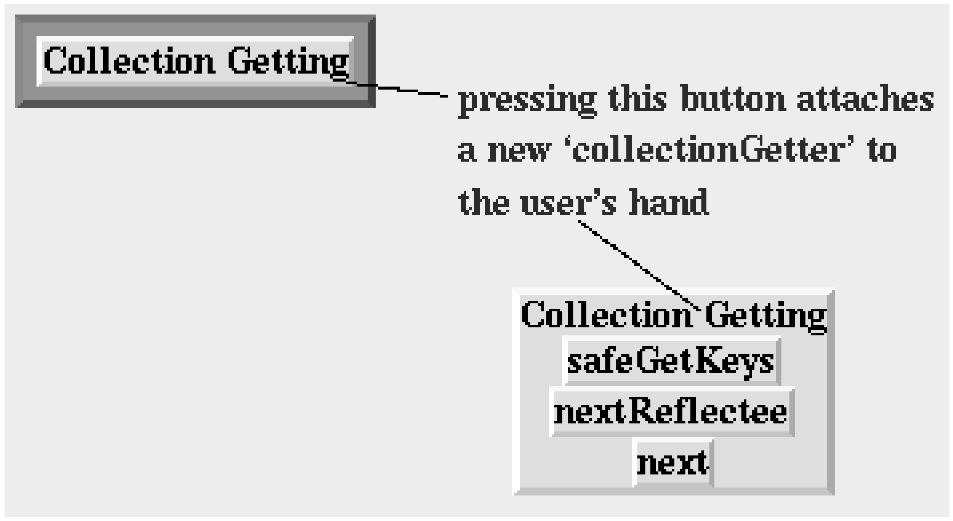

Collection Getting
pressing this button attaches a new 'collectionGetter' to the user's hand
Collection Getting
safeGetKeys
nextReflectee
next

Figure 7.7: User interface for creating collectionGetters.

up' by the user's 'hand' and moved over all the objects to be included in the restructuring. Figures 7.8 and 7.9 illustrate this part of the user interface.

To perform the 'standard' restructurings (Chapters 4 and 5, with or without refactoring of methods) is very straightforward. Having specified the objects to be included in the restructuring, the button labelled 'ihr' is pressed, and the object representing the collection of objects resulting from the restructuring is attached to the user's 'hand'. Restructurings can be performed in stages, with buttons for performing part of the restructuring (with or without refactoring), and then for reintroducing overriding (Section 3.5), removing empty objects and transitively unnecessary inheritance (Section 3.5.3), reintroducing resends (Section 4.7), and cracking leaf objects (Section 4.3.3). Other 'non-standard' restructurings can be performed by pressing the button labelled 'self' to attach the object representing the collection of objects gathered by the ihr collector to the user's 'hand'. The relevant messages can be sent to this collection of objects using an 'evaluator', as shown in Figure 7.10.

In order to relate the original objects to the restructured objects, the collection of restructured objects can be sent messages (using the evaluator) of the

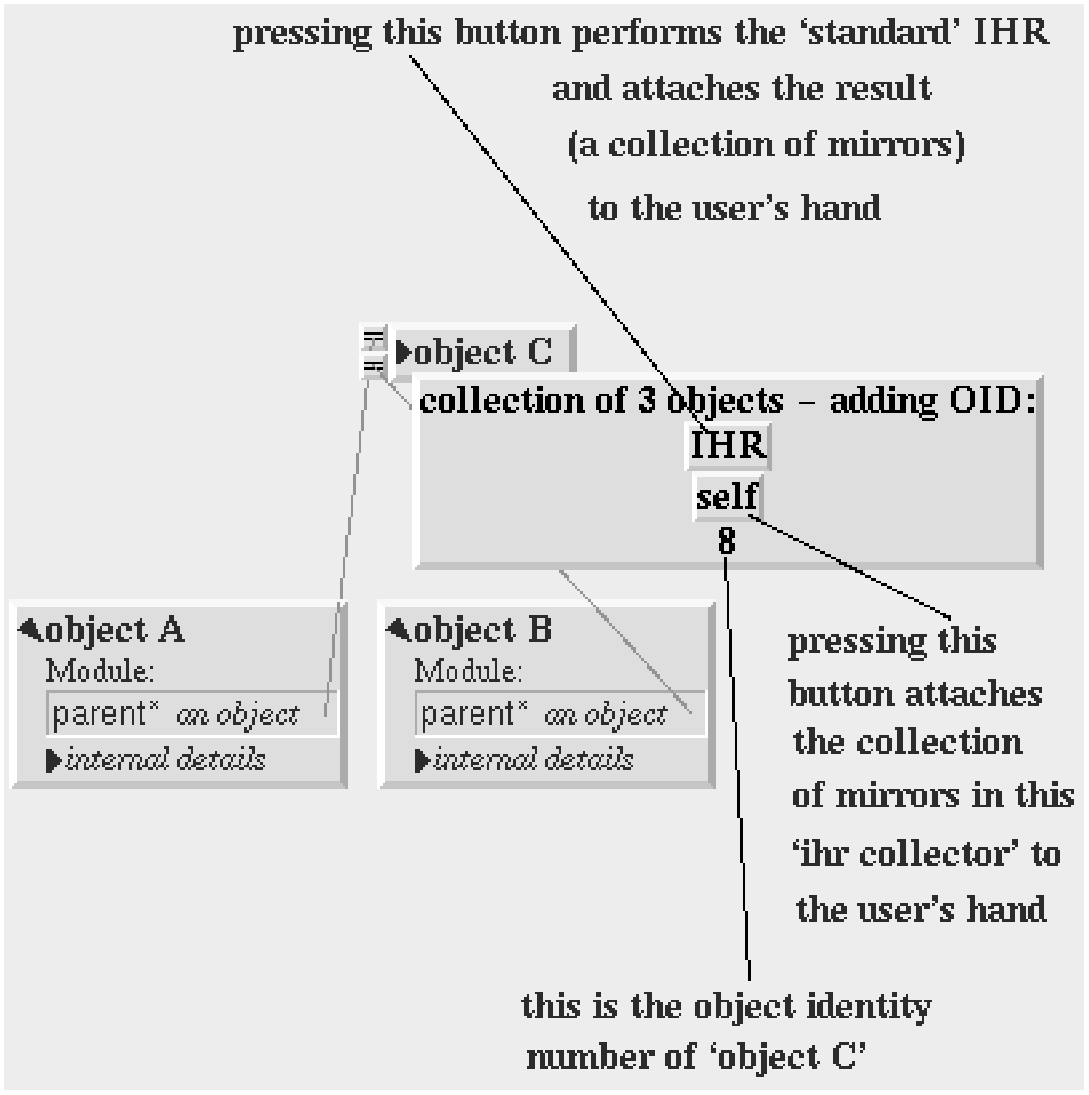


Figure 7.8: User interface to specify objects to be restructured.

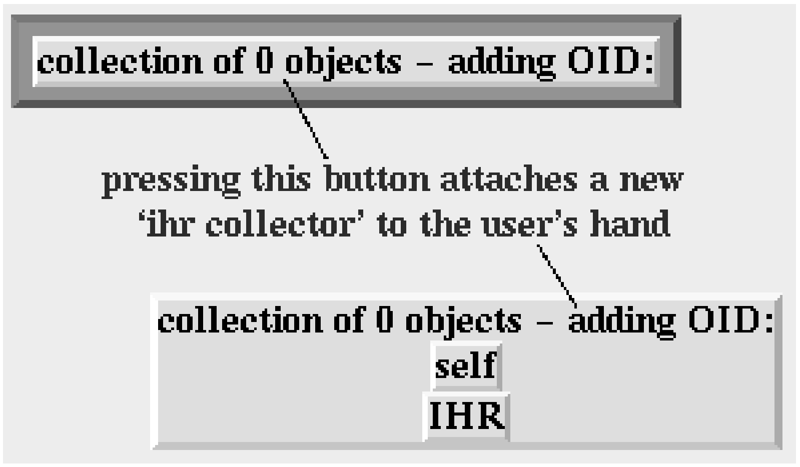

collection of 0 objects – adding OID:
pressing this button attaches a new 'ihr collector' to the user's hand
collection of 0 objects – adding OID:
self
IHR

Figure 7.9: User interface to create new 'ihr collector'.

form replacementFor: anObject, where anObject is the original object, as shown in Figure 7.10. The result of this message send is the appropriate replacement object (if the original object was a preserved object). A more sophisticated user interface for relating the original objects to restructured objects has not been found to be necessary.

Alternatively, objects can be brought out of a collection (that is, they can be brought into the Self world and are not removed from the collection) using the 'next reflectee' button of a collectionGetter, as described earlier.

7.1.3 Programming environment driven interaction

Rather than explicitly requesting certain analysis information or restructurings, the programming environment can be modified to provide certain information, and suggest restructurings, in an unobtrusive way without it being explicitly requested.

The slots of an object outliner can be coloured according to whether they satisfy certain of the simple analyses described earlier. For example, a slot which overrides an inherited slot can be coloured orange, and if it overrides the slot unnecessarily, that is with the same definition as the inherited slot, it can be

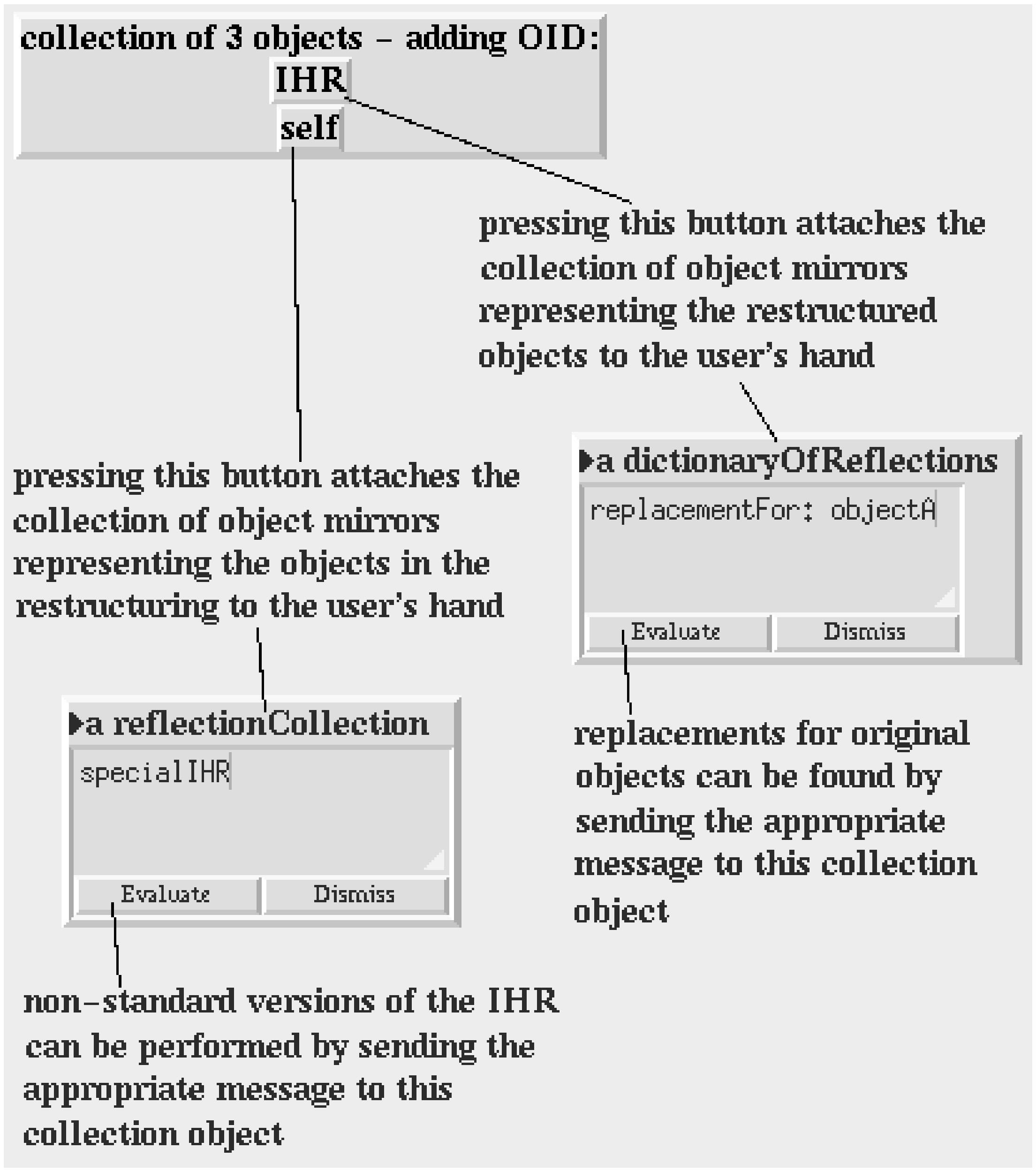

collection of 3 objects – adding OID:
IHR
self
pressing this button attaches the collection of object mirrors representing the restructured objects to the user's hand
pressing this button attaches the collection of object mirrors representing the objects in the restructuring to the user's hand
a dictionaryOfReflections
replacementFor: objectA
Evaluate
Dismiss
replacements for original objects can be found by sending the appropriate message to this collection object
a reflectionCollection
specialIHR
Evaluate
Dismiss
non-standard versions of the IHR can be performed by sending the appropriate message to this collection object

Figure 7.10: User interface for inspecting results of IHR.

coloured red. Similarly, other analysis can be indicated using different colours. This idea is an extension of the way that 'copied-down' slots (see Section 2.3) are indicated.

Similarly, when methods are added to a system, an interactive restructuring system could perform various checks on the code to be added. It could then suggest restructurings to the user, and perform them if required.

Unfortunately, while this would be an unobtrusive way of displaying simple analysis information, and performing appropriate restructurings, it is intrusive to the performance of the Self user interface. To perform even the simplest analysis or restructuring is currently too slow for an interactive environment. Furthermore, to display many different analyses using different colours, the colours have to be chosen carefully so that they are sufficiently easy to distinguish from each other, and a key has to be provided to explain the meaning of different colours to novice users.

The sophisticated programming environments of Self and Smalltalk allow programmers to incrementally modify a system, without having a long 'edit-compile-link-run' cycle. Using an incremental restructuring algorithm, such as [Dicky96], would seem to complement Self's programming environment. Unfortunately, incremental restructuring is not currently practical, because it would significantly affect the responsiveness of such environments, which is one of their most important features. Given more computational power, such an approach may be feasible. Nevertheless, if an interactive system only checks code when it is modified or added to a system, then it will not check legacy code. Also, it can be confusing to a user to do radical rearrangements to a system while it is being used. Large scale restructuring should only be done in response to an explicit request from a programmer. Further discussion of the incremental restructuring algorithm described in [Dicky96] can be found in Section 3.8.3.

7.1.4 Comparison with previous work

The ParcPlace-Digitalk Smalltalk [ParcPlace] programming environment contains a check for messages not implemented when a method is added or modified. It uses a string comparison algorithm to suggest a correction, based on the messages which are implemented in the system. It is implemented in such a way that the interactive performance of the system is not significantly affected. ParcPlace also provides a browser which indicates which methods are inherited and which are overridden.

The Refactoring Browser [Brant1] provides simple user directed low-level restructurings for Smalltalk. An associated tool, called SmallLint [Brant2], checks for some common Smalltalk coding errors. The restructurings are limited and not as sophisticated as those described in Chapters 4 and 5.

ENVY/QA [ENVY/QA] is a commercial product for analysis of Smalltalk programs. It gathers metrics, discovers potential coding errors, provides a 'code coverage tool' (see Section 7.2.2), a tool for automatically producing documentation for applications and a source formatting tool.

One of the restructurings described by Hoeck [Hoeck93], called the 'Remove redundancies' restructuring with 'exclusive common parents' (see Section 4.11.3) is similar to the 'moving same sibling slots' restructuring described in Section 7.1. Hoeck's restructuring will be called 'ecp', and the 'moving same sibling slots' restructuring will be called 'mss'. The ecp considers only immediate parents of classes, whereas the mss considers two level hierarchies. Furthermore, the ecp only considers moving methods if they are the equivalent in all of the children of the common parent class, whereas the mss only requires that all the children understand the message implemented by the method. Hence, the mss includes ecp as well as allowing other cases which the ecp does not.

7.2 Removing unnecessary code

One of the advantages of prototyping systems, such as Self and Smalltalk, compared to more conventional systems, is that they allow programmers to rapidly build and modify programs, and to experiment with different implementations and designs. This is particularly useful for evolving a prototype system to discover a user's requirements if they are not initially well understood, and when developing systems for new and unusual applications. However, a consequence of prototyping is that often old versions of implementations and designs remain in a system. Even if a system is perfectly structured, a particular application may use only a small proportion of the entire system. Therefore, a system for extracting an application program from a system could be necessary, even if a perfect restructuring system existed.

7.2.1 Dynamic Analysis Stripper

In order to remove slots which are unnecessary for an application program, previous work [Agesen94, Moore94] has used static analysis to predict which methods will be needed. While this approach can be very successful in many cases, for languages such as Self there are some problems caused by the dynamic nature of the language. In particular, it is impossible to predict which slots are necessary when computed selector names are used (see Section 2.3).

An alternative approach has been investigated, by implementing a tool for dynamic analysis of a program to determine which slots are necessary. Similar analysis is performed by 'code coverage' tools for determining whether test suites fully 'exercise' an application.

In the dynamic analysis stripper, the slots to be considered for inclusion in the delivered application are modified so that they notify a monitoring object when they are accessed. Note that although all slots could be considered, in most cases only those slots which will be needed in addition to slots which will already exist in a standard system should be considered.

The slots needed by an application are then identified by running the application so that all slots in the application are accessed. While this cannot easily be guaranteed, a dynamic analysis stripper can be left collecting this data for many test runs of an application with minimal affect on its performance, so if left for long enough and run on a sufficient number of test cases it can accurately identify which slots are needed for the application.

Once the slots used by an application have been identified, they can be moved into a module and 'filed-out' (see Section 2.3.2).

In contrast to work which uses static analysis, it is impossible for a dynamic analysis stripper to identify more slots than are needed. However, it is possible for it to fail to identify all necessary slots, because of incomplete exercising of an application, which is a more dangerous failing. However, it does identify slots which are necessary because of computed selectors. Furthermore, it does not require a 'main' program or expression to be specified from which to start a static analysis.

Unfortunately, even the dynamic analysis stripper fails to identify slots referred to using only reflection, for example '_Mirror understands: 'aMessage'' (see Section 2.3.4). Also parent slots are not identified, because they exist for inheritance without being directly accessed. Many of the necessary parent slots can be identified automatically by looking for any parent slots referring to an object which includes any of the identified slots.

An additional use of a dynamic analysis stripper is that it can be used for profiling.

The dynamic analysis stripper was used for delivering (and profiling) Guru. As Guru was developed experimentally, considerable code was written which exists only for testing Guru, and different strategies have been implemented and evaluated for solving the same problems. The necessary code for the released version of Guru [Guru], identified by the dynamic analysis stripper, was less than half of the code written.

7.2.2 Comparison with previous work

Agesen [Agesen94, Agesen96] describes using static type inference to extract application programs from the Self programming environment. He reports results of using his application extractor on several example applications. His technique requires the programmer to specify a message send which starts the application to be extracted (similar to a 'main' function in the language 'C'). His type inference system determines which slots in the system are accessible starting from this message send. Then those slots identified are extracted and made into an executable file. Agesen's extractor can be used automatically in most circumstances, and preserves the behaviour of extracted programs. However, he admits that his extractor has some limitations, in particular it cannot automatically handle computed selectors or reflective code.

ENVY/QA [ENVY/QA] provides a code coverage tool, the intended use of which is to check that test suites of programs/data fully use all the methods in an application. If methods are not executed as a result of running a set of test programs/data then this indicates that the test suite is not comprehensive, and hence needs additional tests.

In [Moore94] the author describes a static analysis tool for Smalltalk, called ProgramFinder, which aims to determine which methods are required for an application. This tool was integrated into a system for translating Smalltalk into CLOS [Keene89] in order to produce stand-alone executables for Smalltalk applications. The aim of the ProgramFinder was to reduce the size of translated applications. The ProgramFinder requires a method to be specified as the starting method of the application to be extracted. From this, a transitive closure of all the methods implementing all the message sends in the starting method, and all the methods implementing all the message sends in those methods etc. is calculated. This is very conservative; it does not use any type inferencing to reduce the number of implementors considered for each message send and so identifies more methods than necessary. A simple mechanism of being able to specify that only certain classes should be considered for implementing any message send was included. The ProgramFinder was found to be useful, but much less accurate than a system such as Agesen's [Agesen94, Agesen96].

ParcPlace-Digitalk Smalltalk [ParcPlace] includes a tool called the 'Stripper' which is used to extract an application from the Smalltalk development environment. The Stripper eliminates certain classes specific to program development, such as the compiler classes, without any analysis as to whether they are needed, but rather due to built-in assumptions.

7.3 Summary

This chapter has described tools which provide program analyses and restructurings, useful both in combination with and separate from the restructurings described in Chapters 4 and 5. The way that a user can interact with such tools is considered, and restructurings which require different forms of interaction (than those of Chapters 4 and 5) are described. Furthermore, user interface tools for making the restructurings described in Chapters 4 and 5 easy to use are described.

The tools described include ones for removing unnecessary slots from a Self system. Those slots identified as unnecessary include slots which override an equivalent slot, slots which have equivalent definitions in 'sibling' objects, and slots which are not executed for a particular application. These slots can be identified independently of the restructurings described in Chapters 4 and 5, providing a collection of complementary tools for program improvement.

Chapter 8

Conclusions

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

Hierarchies created by Guru have 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. The hierarchies produced by Guru do not depend upon the structure of the original inheritance hierarchies. Guru will produce the same inheritance hierarchies for any inheritance hierarchies which define the same methods for the preserved objects.

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 allows programmers to concentrate on ensuring that objects define the correct methods, while leaving the inheritance hierarchy, and factoring methods, to be created automatically.

8.1 Critique and suggestions for further work

The extended IHI algorithm described in Chapter 3 appears to produce well designed hierarchies. However, while the IHI algorithm without overriding produces hierarchies satisfying well justified criteria, the reintroduction of overriding is a less formally justified addition to the algorithm. Rather than introducing overriding into the hierarchies produced by the IHI algorithm, it would be possible to define a set of criteria which includes overriding. This may then lead to an algorithm which infers better hierarchies (which include overriding) than those created by the extended IHI algorithm.

Guru handles only two aspects of the design of object oriented programs: inheritance hierarchy structure and method factoring. There are many other aspects which Guru does not address, and so Guru cannot be claimed to be the only useful sort of restructuring tool. Guru should be used in conjunction with other restructuring tools for best results. In particular, the traits objects discovered by Guru may be too large and lack cohesion, because Guru aims to include shared slots in as few traits objects as possible.

A possible criticism of the approach taken in this work is that basing inheritance hierarchies only on code rather than on the underlying meaning of the code, or on abstractions defined by humans and based on domain knowledge, does not lead to good hierarchies or methods. However, the results presented in Section 6 show that the hierarchies produced do have the structures expected of well designed hierarchies. Thus, basing the design of hierarchies on maximising code sharing does appear to lead to well designed hierarchies. However, although Guru can be used fully automatically, it cannot replace a programmer's understanding of the underlying meaning of code, and so some user involvement should be expected in designing, and restructuring, hierarchies. 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. Similarly, only a programmer can make informed guesses about the likely future changes and reuse of a system.

Whether Guru should work fully automatically, and then the user modify its results if required, or should be made to work semi-automatically is a more difficult question for which there is no definitive answer. In this thesis, fully automatic restructuring was investigated as it is easier to add user interaction to an automatic system than to make a semi-automatic system fully automatic. Also, an automatic system which produces good results is more valuable than a similar semi-automatic system. Furthermore, there has been much less previous research in fully automatic restructuring.

Some object-oriented programmers believe that inheritance hierarchies should be used to define type hierarchies and should not just be used for code sharing. This view is particularly popular amongst advocates of statically typed languages. However, this then leaves the question of what types (abstractions) should be. Abstractions can be view as either intensional or extensional. The intensional view is that an abstraction is the shared properties of a set of objects. The extensional view is that an abstraction is a set of objects with some shared properties. In nearly all object-oriented programming languages, whether statically or dynamically typed, classes are intensional as they define the methods shared by their instances. Guru is compatible with the intensional view, with the shared properties being discovered automatically.

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.

While the reintroduction of overriding described in Section 3.5.2 improves the structure of inheritance hierarchies, the current implementation is not entirely reliable, due to complications in avoiding the introduction of ambiguities as explained in Section 3.5.1. Also, the reintroduction of overriding does not check whether methods contain resends, which it should in order to check whether the method can safely be moved, as mentioned in Section 3.5.1.

The implementation of the removal of methods introduced to replace resends, described in Section 4.7 contains an error described in Section 6.2.1.

Section 6.2.3 includes a description of a situation in which the reintroduction of overriding can positively affect the reintroduction of resends. If an RI method is moved higher in the inheritance hierarchy by the reintroduction of overriding, then, in some circumstances, this can allow more RR methods to be removed by reintroduction of resends.

In general, the way that resends are handled could be improved. Rather than removing resends by converting them into message sends to invoke uniquely named methods, and then trying to reintroduce resends in order to remove those uniquely named methods, an alternative approach could convert resends into a reference to the method that will be executed. Then, the effect of resends on testing equality of methods (see Section 4.4.1) could still be handled correctly, and these references could be used to modify the reintroduction of overriding so that it moves RI methods such that resends can be reintroduced to invoke the correct methods.

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 just over half an hour on a Sun Sparc 20[1]. 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

[1]In a previous implementation [Moore95], restructuring the indexables hierarchy without refactoring methods took less than one minute on a Sun Sparc 5. However, this implementation contained some simplifications which could cause it to fail in some circumstances.

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 6.3 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. For example, if the original hierarchy is too large or complex for a programmer to understand, then the programmer will not be able to determine whether the restructured hierarchy is an improvement. Furthermore, if the results of restructuring are too complex to be understood by a programmer, then they will not be useful.

The restructuring described in Chapter 4 *could* be implemented to be reasonably fast to perform, but nevertheless it would not be desirable to do this restructuring every time that the system is changed. Programmers get used to the inheritance hierarchy, and so changes should not be made too often. Also, for each restructuring the user would have to specify which objects to include. Therefore other techniques could be employed, which are faster and incremental, but do not necessarily produce optimal hierarchies or handle all situations. Some restructurings which may be suitable are discussed in Chapter 7.

Using an incremental algorithm for adding or modifying objects in a hierarchy, such as the algorithm described in [Dicky96], is not practical as the hierarchy has to start in a particular state. Also, [Dicky96] considers complete hierarchies rather than partial hierarchies (see Section 4.3) so cannot be prevented from restructuring objects which the user wants to remain unchanged.

The question of the validity of automatic method factoring is difficult to answer. The amount of factoring is a matter of personal preference, and so there cannot be claimed to be a ‘correct’ factoring of methods. Section 6.3 discusses whether the factoring methods created by Guru are easily understood in practice, and suggests that more work is needed.

Further work is also required to reduce the limitations of Guru's method 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 Section 5.6. Also, there may be a limit to the amount of refactoring that can be performed in an acceptable amount of time.

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 [Ungar94a]. 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 optimising compilers; a refactoring system could benefit from reusing the analysis of such compilers.

The results described in Chapter 6 from applying Guru to Self inheritance hierarchies are very encouraging. However, further experiments should be undertaken to evaluate Guru more thoroughly. For example, evaluating the effectiveness of Guru on initially very badly designed Self code could provide useful insights into areas which require further improvement. Such an experiment was undertaken with a previous version of Guru [Moore95], and the results indicated the Guru may be useful for assisting in the identification of the code which most requires improvement, rather than actually producing well designed hierarchies. Another useful experiment may be to apply Guru to Smalltalk hierarchies which have been translated into Self using Wolczko's 'Smalltalk in Self' translator [Wolczko96].

Complementary tools have been constructed to provide simple program analyses, such as gathering metrics, and user interface tools to make them easy to use. However, the metrics and analyses are of limited usefulness. The restructurings provided as alternatives to those described in Chapters 4 and 5 are also limited, but do find and remove some unnecessary and incorrect code. Two of the restructurings, 'removing overrides unnecessarily' and 'moving same sibling slots' can be applied globally without user intervention as they clearly improve code. Together, they identified and removed over 1% of the slots in the standard Self image. Although this is a small percentage, it was higher than expected. If the standard image is better quality than average code then this figure should be expected to increase when these restructurings are used on average or lower quality code.

The Dynamic Analysis Stripper allows applications to be extracted from the Self programming environment. However, it does not guarantee to correctly identify all the necessary code, and the current implementation is not very robust. Nevertheless, it has been used successfully for extracting the restructuring tool described in Chapters 4 and 5, and overcomes some of the limitations of similar tools which use static analysis.

Despite advances in software technology, in particular the increasing popularity of object-oriented programming, building software remains difficult and time-consuming; there are still no silver bullets [Brooks87]. Tools should be built which incorporate the restructurings implemented by Guru (see Chapters 4, 5 and 7) with those suggested by others (see Sections 3.8, 4.11 and 5.8). In the future, restructuring tools may become part of the standard programming environments of systems, providing an improvement in productivity by reducing the effort needed to handle implementation details, thus allowing programmers to concentrate on the most important aspect of software construction, that is, defining the correct behaviour of the software.

Bibliography

[Agesen93] Ole Agesen, Jens Palsberg and Michael I. Schwartzbach Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In *Proceedings of ECOOP 93*, (LNCS 707, pages 247-267) Springer-Verlag, 1993.

[Agesen94] Ole Agesen and David Ungar. Sifting Out the Gold: Delivering Compact Applications from an Exploratory Object-Oriented Programming Environment. In *Proceedings of OOPSLA 94*, (SIGPLAN Notices 29(10), pages 355-370) ACM, 1994.

[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.

[Agesen96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University (also published as Sun Microsystems Laboratories Technical Report SMLI TR-96-52, 1996), 1995.

[Anderson90] B. Anderson and S. Gossain. Hierarchy evolution and the software lifecycle. In *Proceedings of TOOLS 90*, pages 41-50, Prentice Hall, 1990.

[Bergstein91] Paul L. Bergstein. *Object-Preserving Class Transformations.* In *Proceedings of OOPSLA 91*, (SIGPLAN Notices 26(11), pages 299-313) ACM, 1991.

[Biggerstaff89] Ted J. Biggerstaff. Design Recovery for Maintenance and Reuse. In *IEEE Computer* 22(7), pages 36-49, IEEE, 1989.

[Brant1] World wide web site for the Refactoring Browser. http://st-www.cs.uiuc.edu/users/brant/Refactory/-RefactoringBrowser.html

[Brant2] World wide web site for SmallLint. http://st-www.cs.uiuc.edu/users/brant/Refactory/Lint.html

[Brooks75] Frederick P. Brooks, Jr. *The Mythical Man-Month.* Addison-Wesley, 1975.

[Brooks87] Frederick P. Brooks, Jr. No Silver Bullet - Essence and Accidents of Software Engineering. In *IEEE Computer* 20(4), pages 10-19, IEEE, 1987.

[Budd91] Timothy A. Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1991.

[Canfora93] G. Canfora, A. Cimitile and M. Munro. A Reverse Engineering Method for Identifying Reuseable Abstract Data Types. In *Proceedings of the Working Conference on Reverse Engineering* pages 73-82, IEEE, 1993.

[Canfora96] G. Canfora, A. Cimitile and M. Munro. An Improved Algorithm for Identifying Objects in Code. In *Software - Practice and Experience* 26(1), pages 25-48, John Wiley & Sons, 1996.

[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.

[Casais94] Eduardo Casais. Automatic reorganization of object-oriented hierarchies: a case study. In *Object Oriented Systems* 1, pages 95-115, Chapman & Hall, 1994.

[Casais95] Eduardo Casais. *Managing Class Evolution in Object-Oriented Systems.* Prentice Hall, 1995.

[Chae96] Heung-Seok Chae. *Restructuring of Classes and Inheritance Hierarchy in Object-Oriented Systems.* MSc thesis, Korea Advanced Institute of Science and Technology, 1996.

[Chambers91] Craig Chambers, David Ungar, Bay-Wei Chang and Urs Hölzle. Parents are Shared Parts of Objects: Inheritance and Encapsulation in Self. In *Lisp and Symbolic Computation*, 4(3), pages 207-222, Kluwer Academic Publishers, 1991.

[Chambers93] Craig Chambers. *The Cecil Language: Specification and Rationale.* Technical Report #93-03-05, Department of Computer Science and Engineering, University of Washington, 1993.

[Cheeseman88] Peter Cheeseman, James Kelly, Matthew Self, John Stutz, Will Taylor and Don Freeman. Autoclass: a Bayesian Classification system. In *Proceedings of the Fifth International Conference on Machine Learning*, 1988.

[Chidamber94] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. In *IEEE International Transactions on Software Engineering*, pages 476-493, IEEE, 1994.

[Cook90] William R. Cook, Walter Hill and Peter Canning. Inheritance is not Subtyping. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1990.

[Cook92] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA 92*, (SIGPLAN Notices 27(10), pages 1-15) ACM, 1992.

[Dicky95] Herve Dicky, Christophe Dony, Marianne Huchard and Therese Libourel. ARES, Adding a class and REStructuring Inheritance Hierarchies. In *11 ièmes journées Bases de Données Avancées*, Nancy (France), 1995.

[Dicky96] Herve Dicky, Christophe Dony, Marianne Huchard and Therese Libourel. On Automatic Class Insertion with Overloading. To appear in *Proceedings of OOPSLA 96*, ACM, 1996.

[Dony92] Christophe Dony, Jacques Malenfant and Pierre Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *Proceedings of OOPSLA 92*, (SIGPLAN Notices 27(10), pages 201-217) ACM, 1992.

[ENVY/QA] *ENVY/QA documentation.* Object Technology International Inc, Ottawa, Canada, 1996.

[Fisher87] Douglas H. Fisher. Knowledge Acquisition Via Incremental Conceptual Clustering. In *Machine Learning* 2, pages 139–172, Kluwer Academic Publishers, 1987.

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Garey79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman, 1979.

[Gibbs90] Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz and Xavier Pintado. Class Management for Software Communities. In *Communications of the ACM* 33(9), pages 90-103, ACM, 1990.

[Godin93] Robert Godin and Hafedh Mili. *Building and Maintaining Analysis-Level Class Hierarchies using Galois Lattices.* In *Proceedings of OOPSLA 93*, (SIGPLAN Notices 28(10), pages 394-410) ACM, 1993.

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

[Griswold93] William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. In *ACM Transactions on Software Engineering and Methodology* 2(3), pages 228-269, ACM, 1993.

[Guru] World wide web site for Guru.
http://www.cs.man.ac.uk/~ivan/guru.html

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

[Holsheimer94] Marcel Holsheimer and Arno Siebes. *Data mining: The search for knowledge in databases.* Technical Report CS-R9406, Centrum voor Wiskunde en Informatica, 1994.

[Hölzle91] Urs Hölzle, Craig Chambers and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *Proceedings of ECOOP 91*, (LNCS 512, pages 21-38) Springer-Verlag, 1991.

[Hölzle94] Urs Hölzle and David Ungar. A Third Generation Self Implementation: Reconciling Responsiveness with Performance. In *Proceedings of OOPSLA 94*, (SIGPLAN Notices 29(10), pages 229-243) ACM, 1994.

[Hürsch93] Walter L. Hürsch, Karl J. Lieberherr and Sougata Mukherjea. Object-Oriented Schema Extension and Abstraction. In *ACM Computer Science Conference, Symposium on Applied Computing*, ACM, 1993.

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

[Johnson91] Ralph E. Johnson and Jonathan M. Zweig. Delegation in C++. In *Journal of Object-Oriented Programming* 4(7), pages 31–35, SIGS Publications, 1991.

[Jones90] Clifford B. Jones. *Systematic software development using VDM.* Prentice Hall, 1990.

[Keene89] Sonya E. Keene *Object-Oriented Programming in Common Lisp: A Programmer's guide to CLOS.* Addison-Wesley, 1989.

[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 M. Holland, and Arthur J. Riel. Object-Oriented Programming: An Objective Sense of Style. In *Proceedings of OOPSLA 88*, (SIGPLAN Notices 23(11), pages 323-334) ACM, 1988.

[Lieberherr89] Karl J. Lieberherr, Ian M. Holland. Assuring Good Style for Object-Oriented Programs. In *IEEE Software* 6(5), pages 38-48, IEEE, 1989.

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

[Lieberman86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA 86*, (SIGPLAN Notices 21(11), pages 214-223) ACM, 1986.

[Light93] Marc Light. *Classification in Feature-based Default Inheritance Hierarchies.* Technical Report 473, The University of Rochester 1993.

[Meyer88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Meyer90] Bertrand Meyer. Tools for the new culture: lessons from the design of the Eiffel libraries. In *Communications of the ACM* 33(9), pages 68-88, ACM, 1990.

[Meyer92] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[Mineau90] Guy W. Mineau, Jan Gecsei and Robert Godin. Structuring Knowledge Bases Using Automatic Learning. In *Proceedings of the sixth International Conference on Data Engineering*, pages 274-280, IEEE, 1990.

[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, IEEE, 1995.

[Möller93] K. H. Möller and D. Paulish. *Software Metrics - A Practitioner's Guide to Improved Software Development.* Prentice Hall, 1993.

[Moore94] Ivan R. Moore, Mario Wolczko, and Trevor Hopkins. Babel - A Translator from Smalltalk into CLOS. In *TOOLS USA 1994*, (TOOLS 14, pages 425-433) Prentice Hall, 1994.

[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.

[Moore96a] 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.

[Moore96b] Ivan R. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. To appear in *Proceedings of OOPSLA 96*, ACM, 1996.

[Ong93] C. L. Ong and W. T. Tsai. Class and object extraction from imperative code. In *Journal of Object-Oriented Programming* 6(1), pages 58-68, SIGS Publications, 1993.

[Opdyke92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign (Technical Report UIUC-DCS-R-92-1759), 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* pages 66-73, ACM, 1993.

[ParcPlace] *Objectworks\ Smalltalk User's Guide (second edition).* ParcPlace (now ParcPlace-Digitalk), Sunnyvale, California, USA, 1992.

[Pedersen89] Claus H. Pedersen. Extending Ordinary Inheritance Schemes to Include Generalization. In *Proceedings of OOPSLA 89*, (SIGPLAN Notices 24(10), pages 407-417) ACM, 1989.

[Plato] Plato. *The Republic.* (Translation by Desmond Lee), Penguin Books, 1987.

[Pun89] Winnie W. Y. Pun and Russel L. Winder. *Automating Class Hierarchy Graph Construction.* Research Note RN/89/23, University College London, 1989.

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

[Rak90] Edward J. Rak. *Two redesign tools for Smalltalk.* MSc thesis, University of Illinois at Urbana-Champaign, 1990.

[Sakkinen88] Markku Sakkinen. Comments on the law of Demeter and C++. In *Proceedings of OOPSLA 88*, (SIGPLAN Notices 23(12), pages 38-44) ACM, 1988.

[Schmidt93] Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs. Discrete Mathematics for Computer Scientists.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1993.

[Self4.0] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, Mario Wolczko. *The Self 4.0 Programmer's Reference Manual.* Sun Microsystems Laboratories Inc. and Stanford University, 1995.

[Smith95] Randall B. Smith, John Maloney and David Ungar. The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility. In *Proceedings of OOPSLA 95*, (SIGPLAN Notices 30(10), pages 47-60) ACM, 1995.

[Stein87] L. A. Stein. Delegation is inheritance. In *Proceedings of OOPSLA 87*, (SIGPLAN Notices 22(12), pages 138-146) ACM, 1987.

[Ungar87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA 87*, (SIGPLAN Notices 22(12), pages 227-241) ACM, 1987.

[Ungar91] David Ungar, Craig Chambers, Bay-Wei Chang and Urs Hölzle. Organizing Programs Without Classes. In *Lisp and Symbolic Computation*, 4(3), pages 223-242, Kluwer Academic Publishers, 1991.

[Ungar94a] David Ungar. Private communication. Sun Microsystems Laboratories Inc, Mountain View, California, 1994.

[Ungar94b] David Ungar. self-interest@self.stanford.edu mail group, 1994.

[Ungar95] David Ungar. Annotating Objects for Transport to Other Worlds. In *Proceedings of OOPSLA 95*, (SIGPLAN Notices 30(10), pages 73-87) ACM, 1995.

[Ward95] Martin P. Ward and Keith H. Bennett. Formal Methods for Legacy Systems. In *Journal of Software Maintenance: Research and Practice* 7(3), pages 203-219, 1995.

[Wolczko96] Mario I. Wolczko. self includes: Smalltalk. ECOOP 96, Prototype-based languages workshop, (not published), 1996.

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

[Zimmer95] Walter Zimmer. Using Design Patterns to Reorganize an Object-Oriented Application. In *Architectures and Processes for Systematic Software Construction*, (FZI-Publication 1/95, pages 171-183), Forschungszentrum Informatik an der Universität Karlrsruhe, 1995.

Appendix A

Formal descriptions of the IHI algorithm

This appendix provides a formal description of the IHI criteria, the IHI algorithm, and the computational complexity of the IHI algorithm. All three sections of the appendix were written by Tim Clement.

A.1 A formal description of the criteria for inferred hierarchies

To supplement the informal description of the criteria satisfied by the inheritance hierarchies produced by the IHI algorithm given in Section 3.2, we provide here a formal definition. It will be presented using the syntax of VDM [Jones90], although a familiarity with standard set notation should be enough to read it.

We need to define some graph theoretic notions. Graphs themselves can be modelled as sets of pairs of nodes, each pair representing an edge.

$Graph = (Node \times Node)\textbf{-set}$

In this model, one graph is a subgraph of another if its edges are a subset of the other set. We can define a graph to be a closure if the set of edges is closed under transitivity.

$$\mathit{Closure} = \mathit{Graph}$$

where

$$\textbf{inv-}\mathit{Closure}(c) \quad \triangleq$$
$$\forall n_1, n_2, n_3 \colon \mathit{Node} \cdot$$
$$(n_1, n_2) \in c \wedge (n_2, n_3) \in c \;\Rightarrow$$
$$(n_1, n_3) \in c$$

The transitive closure of a graph is the smallest closure containing that graph.

$$({}^{+})(g \colon \mathit{Graph})\; c \colon \mathit{Closure}$$

$$\textbf{post}\; g \subseteq c \wedge \forall c' \colon \mathit{Closure} \cdot g \subseteq c' \;\Rightarrow\; c \subseteq c'$$

The reflexive transitive closure adds edges from each node to itself

$$({}^{*})(g) \quad \triangleq \quad g^{+} \cup \{n \mapsto n \mid n \colon \mathit{Node}\}$$

An inheritance hierarchy is a graph where the nodes are traits objects or replacement objects (to be denoted by the sets C and O respectively) and there are no loops, in conjunction with a mapping from the nodes to sets of features (to be denoted by the set F).

$$\mathit{Node} = C \mid O$$

$$\mathit{Hierarchy} = \mathit{Graph} \times (\mathit{Node} \xrightarrow{m} F\textbf{-set})$$

where

$$\textbf{inv-}\mathit{Hierarchy}(i, f) \quad \triangleq \quad (\forall n \colon \mathit{Node} \cdot (n, n) \notin i^{+}) \wedge$$
$$\textbf{dom}\, f = \bigcup \{\{n_1, n_2\} \mid (n_1, n_2) \in \textbf{dom}\, i\}$$

The extra criteria for the inheritance hierarchies produced by the IHI algorithm are as follows:

0. They must represent the given object definitions. Object definitions can be modelled as a map from objects to the sets of features they contain.

$$\mathit{ObjectDef} = O \xrightarrow{m} F\text{-}\mathbf{set}$$

The object definition represented by a hierarchy is found by associating with each object all the features from all its ancestors in the hierarchy (including itself).

$$\mathit{objects} : \mathit{Hierarchy} \to \mathit{ObjectDef}$$

$$\begin{aligned}\mathit{objects}(i,f) \quad &\triangleq \quad \{o \mapsto \bigcup\{f(n) \mid n\colon \mathit{Node} \cdot (n,o) \in i^*\} \mid \\ &\quad o\colon O \cdot o \in \mathbf{dom}\, f\}\end{aligned}$$

1. Features appear at a single node

$$\begin{aligned}&\mathit{unique\_features}(i,f) \quad \triangleq \\ &\qquad \forall fs_1, fs_2 \in \mathbf{rng}\, f \cdot \\ &\qquad\qquad fs_1 \cap fs_2 \neq \{\,\} \;\Rightarrow\; fs_1 = fs_2\end{aligned}$$

2. The number of internal nodes must be as small as possible for the set of objects represented.

$$\begin{aligned}&\mathit{minimal}(i,f) \quad \triangleq \\ &\qquad \forall (i',f')\colon \mathit{Hierarchy} \cdot \\ &\qquad\qquad \mathit{objects}(i,f) = \mathit{objects}(i',f') \;\Rightarrow \\ &\qquad\qquad\qquad \mathbf{card}\, f' \leq \mathbf{card}\, f\end{aligned}$$

3. The hierarchy should contain all inheritance consistent with the objects.

$$\begin{array}{l} \mathit{all\_inheritance}(i, f) \quad \triangleq \\ \quad \forall n_1, n_2 \colon \mathit{Node} \cdot \\ \quad\quad (\forall o \colon O \cdot \\ \quad\quad\quad (n_2, o) \in i^+ \;\Rightarrow\; (n_1, o) \in i^+ \\ \quad\quad) \;\Rightarrow\; (n_1, n_2) \in i^+ \end{array}$$

4. Links implied by transitivity should not be explicit in the graph

$$\begin{array}{l} \mathit{no\_transitivity}(i, f) \quad \triangleq \\ \quad \forall (n_1, n_2) \in i \cdot \\ \quad\quad \neg \exists n_3 \colon \mathit{Node} \cdot (n_1, n_3) \in i \wedge (n_3, n_2) \in i \end{array}$$

5. Objects are leaves

$$\mathit{objects\_are\_leaves}(i, f) \quad \triangleq \quad \neg \exists o \colon O, n \colon \mathit{Node} \cdot (o, n) \in i$$

We can justify the claim that these conditions define the hierarchy for a given set of objects uniquely by considering first the nodes and then the edges of the graph.

The set of nodes must provide all the features that the objects need. Further, since features occur in at most one node, each node must be inherited by all the objects requiring any of its features. This means that nodes can contain more than one feature only if these features appear in all the objects in which any of them appear. Unwanted features must be in uninherited nodes, and if the number of nodes is minimal, there will be none of these. The nodes thus partition the features. Minimality also requires that features which do always appear together in the objects share a node: otherwise, the nodes which contain them can be merged. The partition (and hence the number of nodes and the features associated with them) is thus uniquely defined.

Turning to inheritance, it is clear that objects can only inherit from nodes which provide the features they need. The third condition requires that the transitive closure of the inheritance is the largest graph consistent with this, and hence defines it uniquely. In any transitive closure graph, we can determine which edges are replaceable by paths, so the smallest graph generating that transitive closure is also uniquely defined.

The effect of the fifth condition is to add an extra feature to each object, of "being itself". Its presence or absence does not affect the uniqueness of the result, but does affect the minimum number of nodes required in the inheritance graph.

A.2 A formal definition of the inheritance hierarchy inference algorithm

In giving a formal version of the algorithm described in English and diagrams above, we shall assume that we start with the objects represented as a value of the type *ObjectDef* above.

The first step of the algorithm transforms object definitions to grouping graphs, where a grouping graph is a relation between features and objects, which we can model as a set of (feature,object) pairs.

$$\mathit{GroupingGraph} = (F \times O)\textbf{-set}$$

The transformation associates each object with the features it contains.

$$\mathit{step}_1 : \mathit{ObjectDef} \to \mathit{GroupingGraph}$$

$$\begin{aligned}&\mathit{step}_1(od) \quad \triangleq \\ &\quad \{(f, o) \mid \\ &\quad\ f\colon F,\, o\colon O \cdot o \in \textbf{dom}\ od \wedge f \in od(o)\}\end{aligned}$$

A mapping graph is another way of looking at the same information: it associates features with the sets of replacement objects which contain them. Sets containing more than one replacement object correspond to the traits objects of the informal description.

$$\mathit{MappingGraph} = F \xrightarrow{m} O\textbf{-set}$$

The second step just creates this new representation from the old one:

$\mathit{step}_2 : \mathit{GroupingGraph} \to \mathit{MappingGraph}$

$$\mathit{step}_2(gg) \quad \triangleq \quad \{f \mapsto \{o \mid o\colon O \cdot (f, o) \in gg\} \mid f\colon F\}$$

The object sets of the mapping graph form the nodes of the inheritance graph, and the edges are represented as pairs as above.

$$\mathit{InheritanceGraph} = (O\textbf{-set} \times O\textbf{-set})\textbf{-set}$$

The third step of the algorithm constructs an inheritance graph from a mapping graph by putting all possible inheritance edges into the graph: that is, those linking nodes to nodes which represent any proper subset of their objects.

$\mathit{step}_3 : \mathit{MappingGraph} \to \mathit{InheritanceGraph}$

```
$\mathit{step}_3(mg) \quad \triangleq$
    $\textbf{let } \mathit{nodes} = \textbf{rng}\, mg \cup \{\{o\} \mid o\colon O\} \textbf{ in}$
    $\{(v_1, v_2) \mid v_1, v_2 \in \mathit{nodes} \wedge v_2 \subset v_1\}$
```

(We add a node corresponding to each object so that the objects will be represented by terminal nodes of the final graph even if they have no unique features.)

The final step prunes the inheritance graph of those edges implied by transitivity: since these edges are all in the graph just constructed, they are easy to find.

$$\begin{aligned}
&\mathit{step}_4 : \mathit{InheritanceGraph} \rightarrow \\
&\qquad\qquad \mathit{InheritanceGraph}
\end{aligned}$$

$$\begin{aligned}
&\mathit{step}_4(tg) \;\triangleq \\
&\quad \{(v_1, v_2) \\
&\quad \mid (v_1, v_2) \in tg \wedge \\
&\quad\; \neg\exists v_3\colon O\text{-}\mathbf{set} \cdot (v_1, v_3) \in tg \wedge (v_3, v_2) \in tg \\
&\quad \}
\end{aligned}$$

Our representation of the inheritance graph lacks the information on which features are associated with each vertex of the inheritance graph, which would be needed for defining the traits objects. This can easily be recovered by inverting the mapping graph.

$$\begin{aligned}
&\mathit{invert} : \mathit{MappingGraph} \rightarrow \\
&\qquad\qquad (O\text{-}\mathbf{set} \xrightarrow{m} F\text{-}\mathbf{set})
\end{aligned}$$

$$\begin{aligned}
&\mathit{invert}(mg) \;\triangleq\; \{os \mapsto \{f \mid f \in \mathbf{dom}\, mg \wedge mg(f) = os\} \mid \\
&\qquad os \in \mathbf{rng}\, mg \cup \{\{o\} \mid o\colon O\}\}
\end{aligned}$$

A.3 Complexity of the inheritance hierarchy inference algorithm

If the IHI algorithm is to be put to practical use, its complexity should be estimated. There are three things which together characterise the size of the input: the number of objects to be considered, o; the total number of features they define (that is, the sum of the number of features each object contains), f; and the number of distinct features, d. To see how these affect the running times, the graph manipulations of the abstract algorithm must be described in more detail.

If the input is presented object by object, an ObjectVertex can be created for each. For each feature of the object, the ObjectVertex is added to a list in the corresponding FeatureVertex: a new FeatureVertex is created each time a new feature is encountered. There are f features to be considered. If the FeatureVertices are arranged as a balanced tree ordered by feature, the time for each insertion is $O(\log f)$, so the total time taken to build this structure is $O(f \log f)$. It also takes $O(o)$ time to create the ObjectVertices, and $O(d)$ time to create the FeatureVertices, but since the number of features must be at least as large as the number of objects and the number of distinct features, the sorting time dominates this step. The ObjectVertices will appear in the lists at each FeatureVertex in the order in which they were created.

The mapping graph can be constructed by a pass over this structure which creates and links the TraitsVertices. For each FeatureVertex, the set of ObjectVertices associated with that vertex is compared against those in all previously created TraitsVertices: if one is found to be equal then the FeatureVertex is linked to that TraitsVertex, while if the new set is equal to none of them a new TraitsVertex is created and labelled with the new set, and the FeatureVertex linked to that. (Where only one object has a feature, no TraitsVertex needs to be created and the link should be to the ObjectVertex instead.) Since the ObjectVertices appear in the same order at each FeatureVertex, the set comparison can be done in time proportional to the size of the smaller set by comparing corresponding elements in the two lists. The worst case here is when the sets are equal. There are d sets, and if the set sizes are $n_1, n_2, \ldots, n_d$, the time taken is $\Sigma_{i=1}^{d} \Sigma_{j=i+1}^{d} \min(n_i, n_j)$. But $\Sigma_{i=1}^{d} n_i = f$, the total number of features. This makes the worst case of the sum the case when $n_i = f/d$: making one set i larger makes another set j smaller, and so $\min(n_i, n_j)$ will contribute less to the overall sum. The worst case time is thus $\Sigma_{i=1}^{d} \Sigma_{j=i+1}^{d} f/d = f(d+1)/2$, which is $O(fd)$.

Constructing the InheritanceEdges between the TraitsVertices is straightforward. There are at most d TraitsVertices (since they partition the distinct features), and they must be compared pairwise to see if one is inherited by a subset of the objects that inherit the other. The comparison can be done in time proportional to the size of the proposed superset by comparing the labels, so the total time taken is $\Sigma_{i=1}^{d}\Sigma_{j=i+1}^{d} n_j$ where the n_j are the set sizes as before, and this is readily seen to be $O(fd)$.

To start the next step, the TraitsVertex links in each TraitsVertex should be sorted into some arbitrary order: this can take $O(d.d\log d)$ time, since each of the d vertices may have $O(d)$ links in it. In the algorithm of Section 3.3 for pruning the unwanted InheritanceEdges, all (i.e. $O(d)$) TraitsVertices must be considered as grandparents. Each may have InheritanceEdges from $O(d)$ TraitsVertices and $O(o)$ ObjectVertices. The TraitsVertices may in turn have $O(d)$ TraitsVertices and $O(o)$ ObjectVertices as children. These sets must be subtracted from the sets of children of the grandparent. The ordering of ObjectVertices and TraitsVertices in each TraitsVertex allows the common replacement objects to be marked as deleted in time $O(o)$ and the common traits objects to be marked in $O(d)$, so the total time is $O(o+d)$. Since this must be done $O(d^2)$ times in the worst case, the total time is $O(d^2(o+d))$.

Giving an overall complexity means relating o, f and d. It was remarked above that the number of features f must be at least as great as the number of objects o and the number of distinct features d, but there are no other necessary constraints. However, if each vertex introduces exactly one feature, then the number of features that an object inherits will be given by the depth of the hierarchy. If the number of children from each TraitsVertex in a hierarchy is fixed at b, then the depth of the hierarchy will increase with the logarithm (to base b) of the number of objects. Hence f is $O(o\log o)$ The number of distinct features is the number of vertices, which is $O(o)$. The complexities of the steps then become $O(o(\log o)^2)$, $O(o^2\log o)$, $O(o^2\log o)$ and $O(o^3)$, which suggests that the algorithm as a whole should be $O(o^3)$. However, under these assumptions the number of children of a TraitsVertex is constant rather than $O(d)$, and so the InheritanceEdge removing step is $O(o)$, and the overall complexity is $O(o^2\log o)$.

In practice, the number of children might be expected to grow slowly with the number of objects as new subtrees are grafted on to some point in the existing hierarchy. Experiments with the algorithm on randomly generated sets of objects suggest slightly better than $O(o^3)$ performance, which suggests that it should be practical for reasonably large collections of objects.

The pruning step described in Section 3.8 and required to construct hierarchies meeting the minimum inheritance condition requires time exponential in the number of children at each vertex. Since in the worst case there may be $O(o)$ children at a vertex this makes the algorithm as a whole exponential, and because the algorithm is in effect solving the minimum cover problem at each vertex and this is known to be NP-complete [Garey79], this is probably a lower bound on the problem. [Lieberherr91] show that minimum cover problems can be encoded as minimal inheritance problems to show that no alternative pruning strategy can improve on this in the worst case, and as a result adopt an algorithm that is not optimal by their own criteria. However, the arguments above suggest that even this pruning may well be feasible in practice if it is deemed necessary after the arguments to the contrary in Section 3.8.