A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies

Ivan Moore[†], **Tim Clement**[‡]

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

[‡]Adelard,
Coborn House, 3 Coborn Road,
London E3 2DA, England

Abstract

This paper[1] describes an algorithm for inferring inheritance hierarchies. It is simple both to understand and to implement. It is also efficient enough for use with realistically sized problems. The solutions produced meet a set of criteria, which are justified as producing the inheritance hierarchy which most clearly reflects the inherent structure of the objects to which it is applied. The motivation for such an algorithm is discussed, and a comparison is made with two similarly motivated algorithms. An example of an application using the algorithm is presented.

1 Introduction

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. For example, the object 3 is not only an integer but also a number, hence shares abstract behaviour with other numbers, and shares more specific behaviour with other integers. The structure of the inheritance hierarchy reflects the abstractions shared between objects, by sharing behaviour (methods), structure (instance variable structure) and data (class variables). (Some object-oriented programming languages do not share all of these different kinds of features using inheritance.) In this paper methods, structural information and data will be called features. Using inheritance to share features makes initial definition and subsequent maintenance of a system easier.

However, designing inheritance hierarchies is hard, precisely because it requires the identification of appropriate abstractions. Texts on object oriented design such as [Meyer 88] provide guidelines, but it remains an art rather than a science. Even if the initial hierarchy is well designed, subsequent extensions and modifications may turn out to be best handled by a radical rearrangement of the hierarchy, and this seldom happens. Even in the basic libraries of object oriented languages there is often scope for improvement.

An alternative approach to constructing a hierarchy by hand is to infer one from the features of the objects that a program creates. (Where objects are created as instances of certain, *concrete* classes, the features of these classes should be considered. Where they are created as copies of other objects, only a set of prototype objects need to be considered. Where there is an existing hierarchy, this can be "flattened" to determine the sets of features that objects contain, as explained in [Moore 95].) This paper will describe an algorithm (the *inheritance hierarchy inference (IHI) algorithm*) which infers an inheritance hierarchy from a set of objects and their features. An example of its action is shown below: from the objects of Figure 1 it will infer the hierarchy of Figure 2. The arrows represent inheritance of features from a parent in the hierarchy, and throughout this paper the direction of the arrows is from parents to

[1]Published in TOOLS Europe 1996, Prentice-Hall.

their children, rather than the more conventional direction from children to their parents, in order to simplify the description of the algorithm. In the inferred hierarchy, object A defines feature f3 for itself, inherits feature f2 from its immediate parent and inherits feature f1 from its parent's parent.

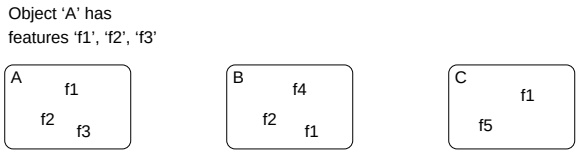

Figure 1: A collection of objects with their features

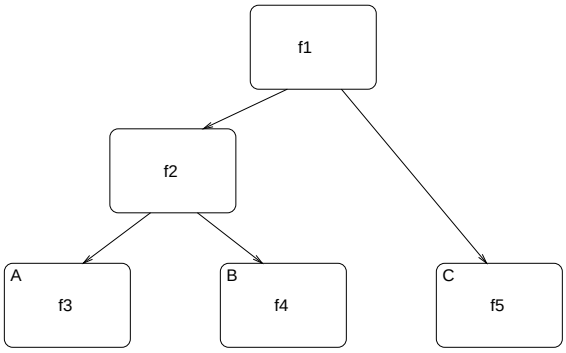

Figure 2: The inferred inheritance hierarchy

In general, the algorithm may construct hierarchies with multiple inheritance, in which case the diagram will be a graph rather than a tree.

Different definitions of what is meant by a 'feature', and the equality of features, can be used in the algorithm for different results. 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. Alternatively, for inferring a 'type hierarchy' for a strongly 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[Pun 90]. The IHI algorithm has been used in the implementation of a re-engineering tool[Moore 95] for the dynamically typed language Self[Ungar 87]; defining features by their name and their 'meaning', where the meaning of a method is a parsed version of it. For re-engineering a statically typed language, method features would also have to include the types of arguments and return values.

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 objects, so each original object must have a corresponding node in the hierarchy which inherits or defines exactly the set of features that the original defines. The structure of nodes above the objects (which correspond to abstract classes in a class based language) and their inheritance links, being new, can take any form, so many hierarchies will satisfy this correctness condition, including the one which leaves the objects unchanged. Further criteria must be met if the hierarchy is to be a representation of a structure that might naturally be inferred from 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 at exactly one node in the hierarchy. (It must appear at least once if it appears in any of the objects to meet the correctness condition.) The motivation for this criterion is the motivation for having inheritance in the language in the first place: it makes a system more compact, and easier to maintain. It is used elsewhere [Pun 89] 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 internal nodes (referred to as *classes* below) should be used in the hierarchy. This will mean fewer individual definitions to be understood. Since for correctness every feature must appear in some class or object and no feature can appear in a class that will be inherited from by an object that does not contain it, classes will contain more than one feature if and only if that combination of features is always found together in the objects.

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 the set of objects which inherits from some class C also inherits from class D, then C should itself inherit (directly or indirectly) from class D. As a consequence, we

prefer the hierarchy of Figure 3 to that of Figure 4.

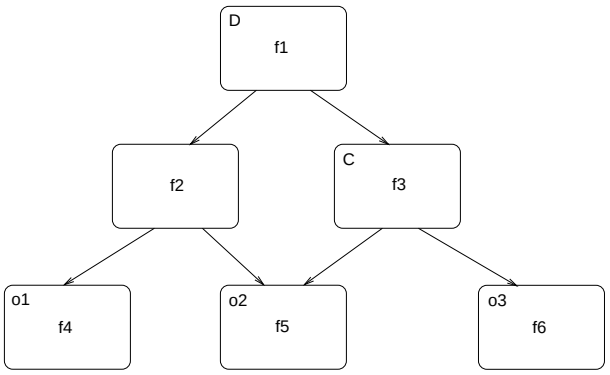

Figure 3: Hierarchy with all inheritance consistent with the objects

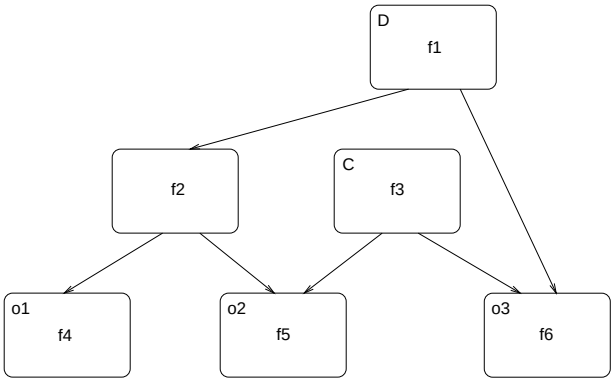

Figure 4: Hierarchy which does not satisfy the third criterion

The motivation is that if the class of objects containing the features of class C (f3 in the example) is really not a subclass of D (those containing f1), a representative selection of objects should include at least one where f3 does not occur with f1 just as there are objects where f1 occurs without f3. If the objects are not representative, then there is no reason to believe anything about the hierarchy inferred.

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 definitions of the objects and classes more complex and sends the readers of the code to look immediately at classes that they will encounter eventually anyway.

A final criterion is that the original objects should correspond to 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 [Johnson 88]. This is seen as less important than the other criteria, and may be relaxed to allow objects as internal nodes, by a small modification to the algorithm. The five criteria together are sufficient to uniquely define the hierarchy inferred from any set of objects (see Appendix B for a brief justification).

3 The IHI 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 ClassVertices, and two types of edges, 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 ClassVertices in which a feature is defined. ClassVertices represent the inferred classes, 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 [Lieberherr 91, Hoeck 93] there are similarities which make comparison easier.

Consider the objects in Figure 5, shown with their features inside them.

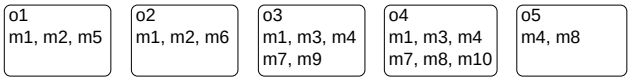

Figure 5: Example problem objects

The first step of the algorithm creates a bipartite graph with a unique FeatureVertex for each feature, and FeatureEdges to the objects they appear in. For the example, this would produce the

graph shown in Figure 6. If each FeatureVertex were used to define a class and each FeatureEdge were used to define an inheritance link, then 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 classes than necessary and the second criterion is not satisfied.

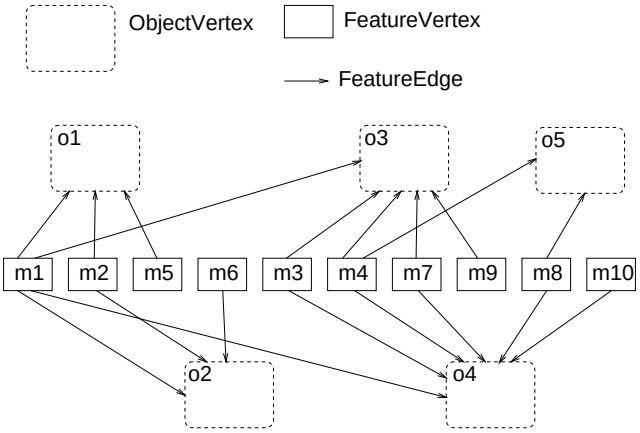

Figure 6: Initial grouping graph

To minimise the number of classes in the hierarchy, ClassVertices are introduced. The next step of the algorithm creates a ClassVertex 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.) The FeatureEdges from the FeatureVertices to their objects are then replaced by a single FeatureEdge to the ClassVertex with those objects as its label. The effect on the example is shown in Figure 7.

This graph now represents all the new nodes that will appear in the inferred hierarchy and the features that they will have. It is called the *mapping graph*. By construction, the ClassVertices and ObjectVertices partition the features amongst themselves (each FeatureVertex has exactly one FeatureEdge) so there will be no duplication in any resulting hierarchy. The label of every ClassVertex identifies exactly the set of objects

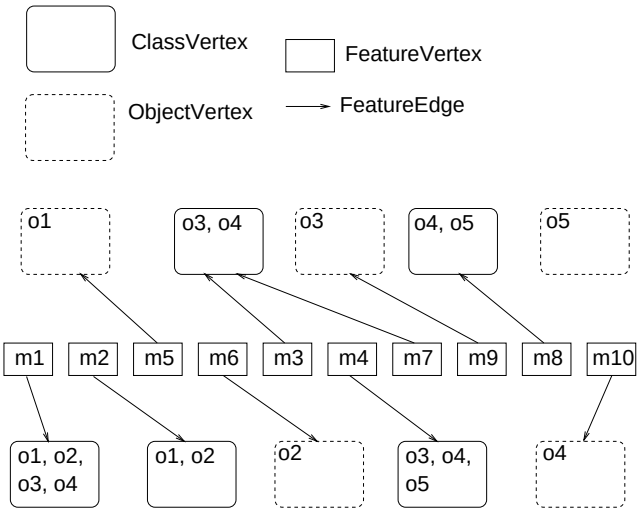

Figure 7: Mapping graph with FeatureEdges and FeatureVertices

containing the features given by the FeatureEdges and no two ClassVertices have the same label so the number of classes is as small as possible. The mapping graph is useful in its own right because it shows what inherent classifications exist, and what 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 ClassVertex or ObjectVertex, as in Figure 8, which represents the same mapping graph as Figure 7.

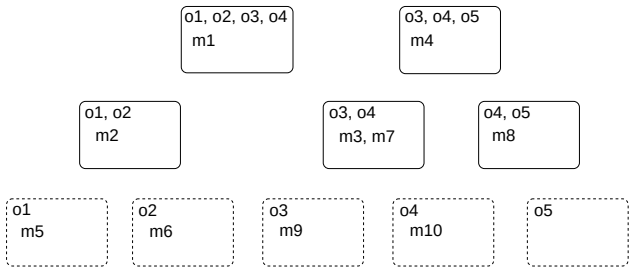

Figure 8: The mapping graph as labelled nodes

The inheritance edges which may be needed in the hierarchy are those connecting each ClassVertex (let its label be the set *os*) to every other

ClassVertex with a subset of *os* as its label, and to every object that appears in *os*. (If we consider the labels alone, then, the resulting graph is a subgraph of the subset inclusion lattice.) For example, an InheritanceEdge should be added from 'o1, o2, o3, o4' to 'o1, o2' because 'o1, o2' is a proper subset of 'o1, o2, o3, o4'. The edges from ClassVertices to ObjectVertices are enough to ensure that each object inherits the features it needs (as Figure 9 shows) but gives only a two level inheritance hierarchy. The complete set of edges gives the hierarchy shown in Figure 10, and satisfies the third criterion by construction, but clearly does not satisfy the fourth criterion.

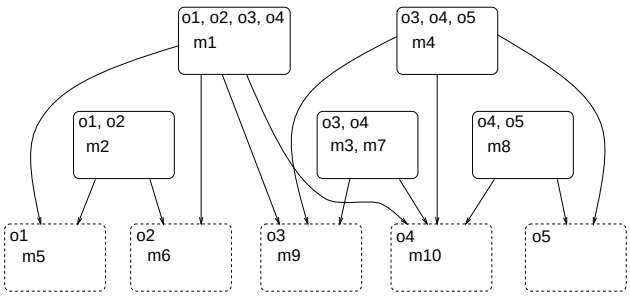

Figure 9: Inheritance graph with class-object links only

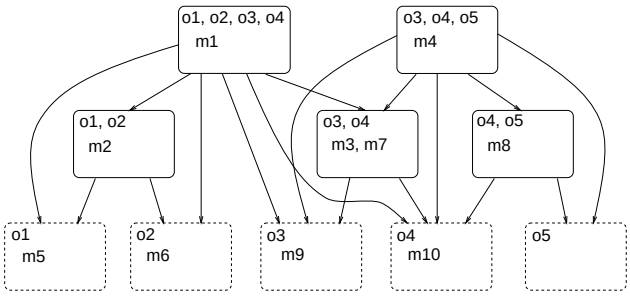

Figure 10: Inheritance graph with all edges from third step

In order to remove InheritanceEdges which are unnecessary due to transitivity; for each ClassVertex PV and all ClassVertices CV with an InheritanceEdge (that exists before any edges are removed) from PV, remove all InheritanceEdges from PV to all vertices (both ClassVertices and ObjectVertices) which have an InheritanceEdge from CV. For example, the ClassVertex labelled (o3, o4, o5) has InheritanceEdges to (o3, o4), (o4,o5), (o5), (o4) and (o3). ClassVertex (o3, o4) has InheritanceEdges to (o4) and (o3), therefore, InheritanceEdges from (o3, o4, o5) to (o4) and (o3) are removed. Similarly, the InheritanceEdge from (o3, o4, o5) to (o5) is removed because (o4, o5) has an InheritanceEdge to (o5). The resulting graph will now represent the inferred inheritance hierarchy as objects and their immediate children, as shown in Figure 11.

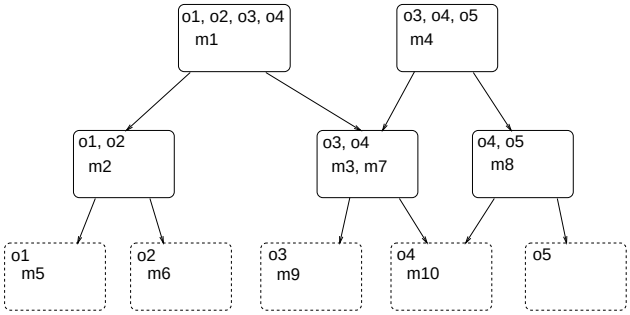

Figure 11: Inferred inheritance hierarchy

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

4 Comparison with previous work

There have been other investigations of automatic inheritance hierarchy construction from object descriptions, producing hierarchies which satisfy different criteria.

Those constructed by Pun and Winder [Pun 89] satisfy our first criterion, but are not guaranteed to satisfy the other criteria used in this paper. Furthermore, they favour multiple inheritance over single inheritance. The stronger criteria used here, which have been justified on general grounds, seem to lead to better hierarchies. An example taken from [Pun 90] 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 12. Their algorithm produces the hierarchy in Figure 13, while the IHI algorithm produces the hierarchy shown in Figure 14.

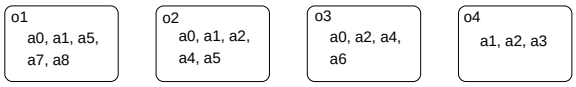

Figure 12: An example from [Pun 90]

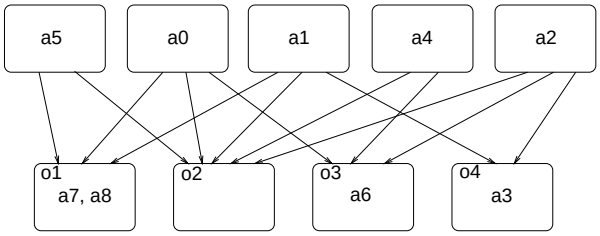

Figure 13: Hierarchy produced by [Pun 90]

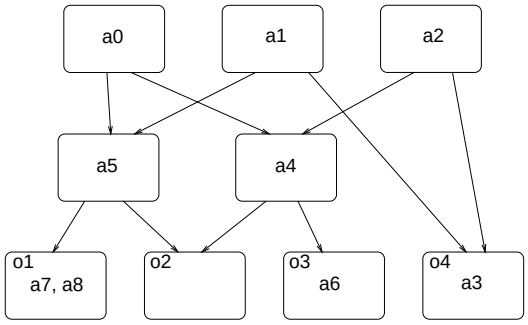

Figure 14: Hierarchy produced by the IHI algorithm

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

Lieberherr et al [Lieberherr 91] adopt the first two optimality criteria used here, but replace the other criteria with a requirement that the number of inheritance links should be minimised. This clearly subsumes the fourth criterion, but gives different results from our third one. For example, while the third criterion here will give the hierarchy shown in Figure 15 the minimality requirement forces one of the top edges to be removed, leaving a hierarchy like that of Figure 16. (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.

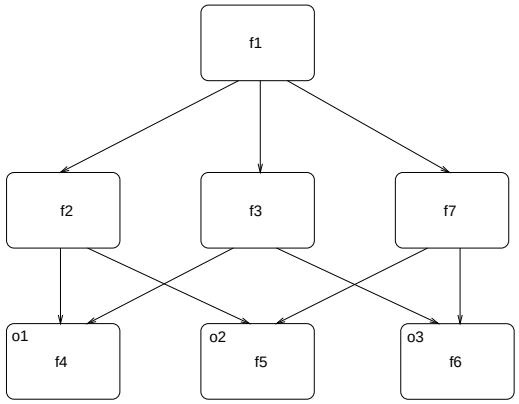

Figure 15: Hierarchy produced by the IHI algorithm

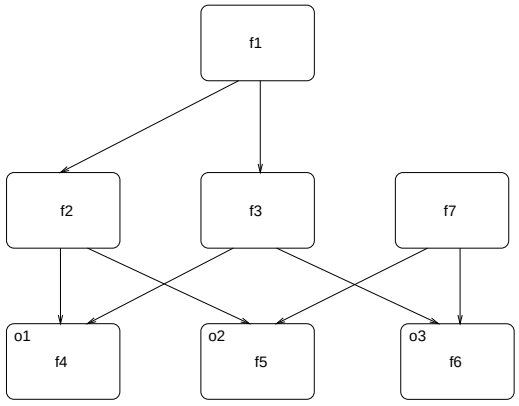

Figure 16: Hierarchy satisfying minimal inheritance criterion

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 outgoing InheritanceEdges of each of the ClassVertices, 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 their criteria or ours: a reason for this will be discussed in Appendix A.

If minimising the number of edges is held to be important, the IHI algorithm can be extended to meet the criterion by adding a new final step. The label of a ClassVertex is given by the set of objects which inherit from the class. 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 ClassVertex 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 15 and 16 show. The InheritanceEdges from CV to 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.

Casais[Casais 92] describes an incremental approach to restructuring (rather than creating) 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.

Problems similar to inferring inheritance hierarchies appear in several areas of research, in particular conceptual clustering [Fisher 87] and data mining [Holsheimer 94]. Note that the inheritance hierarchy structure is only one aspect of the design of an object-oriented system, and other work ([Casais 90, Hoeck 93, Opdyke 92] as well as parts of [Lieberherr 91]) has investigated (semi) automatic restructuring of object-oriented systems with regard to other aspects of design.

5 Application of the IHI algorithm

The IHI algorithm has been used in the implementation of a re-engineering tool for Self[Ungar 87], called Guru[Moore 95]. This tool can automatically restructure a Self inheritance hierarchy whilst preserving the behaviour of its objects. An evaluation of the application of Guru has been presented in [Moore 95]. It was applied to part of the collection hierarchy in the standard Self image, shown in figure 17, which was restructured with a small amount of user intervention into the hierarchy shown in figure 18. In the restructured hierarchy, 34 fewer methods redefined inherited methods. Furthermore, 8 redundant methods were removed, but 8 disambiguating methods were added due to the multiple inheritance introduced into the hierarchy. Inheritance from the Self equivalent of concrete classes was eliminated.

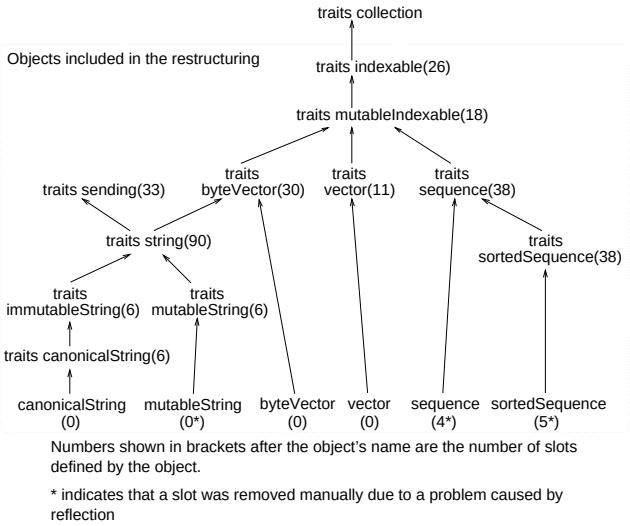

Figure 17: Collection objects inheritance hierarchy

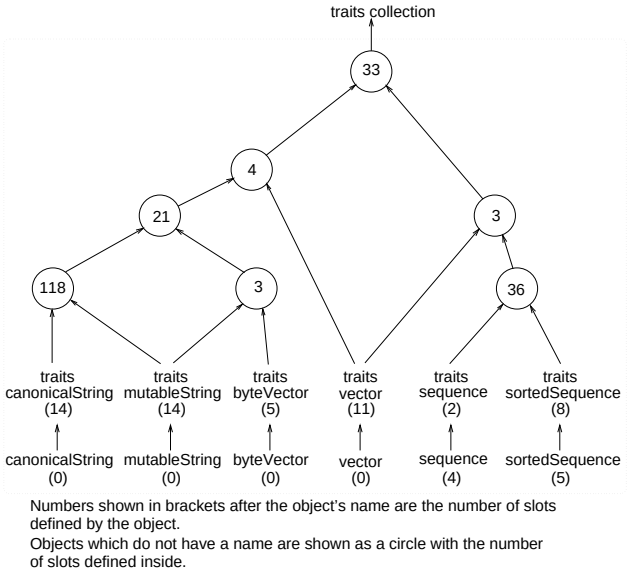

Figure 18: Result of restructuring collection objects

6 Summary

The algorithm described in this paper is useful for a variety of applications, in particular

for inheritance hierarchy creation and restructuring [Lieberherr 91, Pun 89, Moore 95]. Creating inheritance hierarchies manually is difficult. Compared to other algorithms for achieving similar results, the IHI algorithm is simpler and easier to understand and implement and produces results which satisfy well justified criteria. An implementation of the algorithm in Self [Ungar 87] is efficient enough for use on problems of around 500 objects, which is more than adequate for most applications. A more formal discussion of the complexity of the algorithm is given in Appendix A.

The hierarchies produced by the algorithm may in some circumstances be unnatural because it makes no provision for overriding in the inheritance. Used to excess, overriding is a sign of a badly conceived hierarchy, but in moderation it 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 classes in the hierarchy. Some work has been done on extensions to the algorithm which can consider hierarchies with overriding.

The IHI algorithm aims to infer hierarchies which reflect the structure of the objects in the system. 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, even programmers cannot predict the future accurately, so hierarchies often require restructuring despite attempts to make them easy to evolve and reuse.

Related to the inference and restructuring of inheritance hierarchies is the problem of extending an existing hierarchy to include new classes. It is sometimes difficult to find which classes a new class should inherit from, and it may be that there are no classes to inherit from without either inheriting features which are not wanted, or duplicating features. One way of approaching this problem is to create the new class with no inheritance, duplicate the features which are required, and then restructure the hierarchy using the IHI algorithm.

Acknowledgements

The first author would like to thank the EPSRC for funding this work. We are both indebted to Richard Banach and Carole Goble for informative discussions, and to Jon Taylor and George Paliouras for their comments on drafts of this paper.

References

[Casais 90] Eduardo Casais. Managing Class Evolution in Object-Oriented Systems. In *Object Management*. Centre Universitaire d'Informatique, Geneve, 1990.

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

[Fisher 87] Douglas H. Fisher. Knowledge Acquisition Via Incremental Conceptual Clustering. *Machine Learning*, 2:139–172, 1987.

[Garey 79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

[Hoeck 93] Bernd H. Hoeck. *A Framework for Semi-Automatic Reorganisation of Object-Oriented Design and Code*. MSc thesis, University of Manchester, 1993.

[Holsheimer 94] Marcel Holsheimer and Arno Siebes. Data mining: The search for knowledge in databases. Technical Report CS-R9406, Centrum voor Wiskunde en Informatica, 1994.

[Johnson 88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, 1988.

[Jones 90] C. B. Jones. *Systematic software development using VDM* Prentice-Hall, 1990.

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

[Meyer 88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall, 1988.

[Moore 95] Ivan R. Moore. Guru - A Tool for Automatic Restructuring of Self Inheritance Hierarchies. In *TOOLS USA 1995.* (TOOLS 17) Prentice-Hall, 1995.

[Opdyke 92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Pun 90] Winnie W. Y. Pun. *A Design Method for Object-Oriented Programming.* PhD thesis, University College London, 1990.

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

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

A Complexity of the 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 ClassVertices. For each FeatureVertex, the set of ObjectVertices associated with that vertex is compared against those in all previously created ClassVertices: if one is found to be equal then the FeatureVertex is linked to that ClassVertex, while if the new set is equal to none of them a new ClassVertex is created and labelled with the new set, and the FeatureVertex linked to that. (Where only one object has a feature, no ClassVertex 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 ClassVertices is straightforward. There are at most d ClassVertices (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 ClassVertex links in each ClassVertex 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 for pruning the unwanted InheritanceEdges, all (i.e. $O(d)$) ClassVertices must be considered as grandparents. Each may have InheritanceEdges to $O(d)$ ClassVertices and $O(o)$ ObjectVertices. The ClassVertices may in turn have $O(d)$ ClassVertices and $O(o)$ ObjectVertices as children. These sets must be subtracted from the sets of children of the grandparent. The ordering of ObjectVertices and ClassVertices in each ClassVertex allows the common objects to be marked as deleted in time $O(o)$ and the common classes 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 ClassVertex 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 ClassVertex 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 4 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 [Garey 79], this is probably a lower bound on the problem. [Lieberherr 91] 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 4.

B A formal description of the criteria

To supplement the informal description of the criteria satisfied by the inheritance hierarchies produced by the IHI algorithm given in Section 2, we provide here a formal definition. It will be presented using the syntax of VDM [Jones 90], 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.

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

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.

$$Closure = Graph$$

where

$$\mathbf{inv}\text{-}Closure(c) \triangleq \forall n_1, n_2, n_3 : Node \cdot (n_1, n_2) \in c \wedge (n_2, n_3) \in c \Rightarrow (n_1, n_3) \in c$$

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

$(^+)(g\colon Graph)\ c\colon Closure$

post $g \subseteq c \wedge \forall c'\colon Closure \cdot g \subseteq c' \ \Rightarrow\ c \subseteq c'$

The reflexive transitive closure adds edges from each node to itself

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

An inheritance hierarchy is a graph where the nodes are classes or 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).

$$Node = C \mid O$$

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

where

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

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.

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

   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).

   $$objects : Hierarchy \rightarrow ObjectDef$$

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

1. Features appear at a single node

   $$unique\_features(i,f) \ \triangleq\ \forall fs_1, fs_2 \in \mathbf{rng}\, f \cdot fs_1 \cap fs_2 \neq \{\} \ \Rightarrow\ fs_1 = fs_2$$

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

   $$minimal(i,f) \ \triangleq\ \forall (i', f')\colon Hierarchy \cdot objects(i,f) = objects(i',f') \ \Rightarrow\ \mathbf{card}\, f' \leq \mathbf{card}\, f$$

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

   $$all\_inheritance(i,f) \ \triangleq\ \forall n_1, n_2\colon Node \cdot (\forall o\colon O \cdot (n_2, o) \in i^+ \ \Rightarrow\ (n_1, o) \in i^+) \ \Rightarrow\ (n_1, n_2) \in i^+$$

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

   $$no\_transitivity(i,f) \ \triangleq\ \forall (n_1, n_2) \in i \cdot \nexists n_3\colon Node \cdot (n_1, n_3) \in i \wedge (n_3, n_2) \in i$$

5. Objects are leaves

   $$objects\_are\_leaves(i,f) \ \triangleq\ \nexists o\colon O, n\colon Node \cdot (o,n) \in i$$

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.

C A formal definition of the 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.

$$GroupingGraph = (F \times O)\text{-}\mathbf{set}$$

The transformation associates each object with the features it contains.

$$step_1 : ObjectDef \to GroupingGraph$$

$$step_1(od) \triangleq \{(f, o) \mid f\!: F, o\!: O \cdot o \in \mathbf{dom}\ od \wedge f \in od(o)\}$$

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

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

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

$$step_2 : GroupingGraph \to MappingGraph$$

$$step_2(gg) \triangleq \{f \mapsto \{o \mid o\!: O \cdot (f, o) \in gg\} \mid f\!: F\}$$

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

$$InheritanceGraph = (O\text{-}\mathbf{set} \times O\text{-}\mathbf{set})\text{-}\mathbf{set}$$

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.

$$step_3 : MappingGraph \to InheritanceGraph$$

$$step_3(mg) \triangleq \mathbf{let}\ nodes = \mathbf{rng}\ mg \cup \{\{o\} \mid o\!: O\}\ \mathbf{in}\ \{(v_1, v_2) \mid v_1, v_2 \in nodes \wedge v_2 \subset v_1\}$$

(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.

$$step_4 : InheritanceGraph \to InheritanceGraph$$

$$step_4(tg) \triangleq \{(v_1, v_2) \mid (v_1, v_2) \in tg \wedge \neg\exists v_3\!: O\text{-}\mathbf{set} \cdot (v_1, v_3) \in tg \wedge (v_3, v_2) \in tg\}$$

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 classes. This can easily be recovered by inverting the mapping graph.

$$invert : MappingGraph \to (O\text{-}\mathbf{set} \xrightarrow{m} F\text{-}\mathbf{set})$$

$$invert(mg) \triangleq \{os \mapsto \{f \mid f \in \mathbf{dom}\ mg \wedge mg(f) = os\} \mid os \in \mathbf{rng}\ mg \cup \{\{o\} \mid o\!: O\}\}$$