An Incremental Class Reorganization Approach

Eduardo Casais

Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14, D-7500 Karlsruhe 1, Germany

Abstract. Software components developed with an object-oriented language require frequent reorganizations before they become stable, reusable classes. We propose a new algorithm that analyses the redefinitions carried out on inherited properties when a class is added to a hierarchy, and restructures the hierarchy to discover missing abstractions and to enforce programming style guidelines. We illustrate our automatic restructuring approach with simple examples, describe formally the algorithm and the object model it is based on, and discuss its suitability for object-oriented software engineering. The results of applying the algorithm to the Eiffel library are examined.

1 Building Reusable Classes

Object-oriented languages are currently considered as one of the most promising approaches for coping with the problems plaguing software development. This favour finds its explanation in the tight integration of a comprehensive set of abstraction facilities — namely, classification, encapsulation, inheritance and delayed binding — within a single programming framework. Together with interactive tools like browsers and debuggers, a high-level graphical interface, and a set of standard reusable classes, object-oriented environments endow programmers with a rich toolkit for exploratory prototyping and present considerable advantages over traditional methodologies for developing applications incrementally.

However, software developers working with an object-oriented system are frequently led to modify extensively or even to reprogram existing classes so that they fully suit their needs. Thus, the library provided with the Eiffel environment incurred major redesigns, mainly to standardize class interfaces and to explicitly factor out common properties appearing in several components [8]. This problem occurred in spite of accumulated and documented experience in building comparable libraries with other programming languages — an unequivocal sign that class design is an intrinsically complex task. Similar problems have been reported about the design of class libraries for various application domains [1].

There are a number of reasons that explain why such difficulties arise with the object-oriented approach:

- User needs are rarely stable: additional functionality has to be constantly integrated into existing applications, resulting in considerable program restructuring.
- Because of the variety of mechanisms provided by object-oriented languages, the best choice for representing a real-world entity in terms of classes is not always readily apparent [4]. Moreover, the object-oriented approach relies heavily on a mechanism, inheritance, that can serve many purposes — such as denoting specialization relationships, enforcing typing constraints, and sharing implementations.

This variety in the permissible usages of inheritance leads to a semantic overloading of links between classes and complicates the task of hierarchy designers and clients.

- Experience shows that stable, reusable classes are not designed from scratch, but are "discovered" through an iterative process of testing and improvement [5][8].

In fact, an important assumption must be satisfied for applying such powerful techniques as inheritance, genericity or delayed binding to application development. Real-world concepts have to be properly encapsulated as classes, so that they can be specialized or combined in a large number of programs. Inadequate inheritance structure, missing abstractions in the hierarchy, overly specialized components or deficient object modelling may seriously impair the reusability of a class collection. The collection must therefore evolve to eliminate such defects and improve its robustness and reusability.

2 Managing Class Evolution

Among the approaches developed in the recent years to control evolution in object-oriented systems, we identify four general categories: tailoring, surgery, versioning and reorganization.

Tailoring consists in adapting slightly class definitions when they do not lead to easy subclassing. The assumption is that the functionality of a hierarchy can be adjusted to derive new classes without actually changing existing definitions.

The common tailoring mechanisms available in object-oriented languages are the renaming of attributes (variables and methods) and the redefinition of properties (variable types, method bodies and class interfaces) [4]. With Objective-C, the programmer can shadow an existing class with its own definition, through the "pose-as" mechanism. With HyperCard, individual cards can extend or modify the definition inherited from their common background with special-purpose fields, buttons, or scripts; this amounts to a per-instance tailoring of classes. Evidently, an undisciplined use of tailoring quickly leads to an incomprehensible subclassing structure, overloaded with special cases, and difficult to manage efficiently with current database technology.

Surgery decomposes all modifications that can be brought to classes into specific, primitive update operations. Because of the multiple connections between class descriptions, care has to be taken so that the consistency of the hierarchy is guaranteed after applying these primitives. This problem has been extensively investigated in the area of object-oriented databases [2][9][12], where the proposed approaches are typically broken down into the following steps:

1. The first step consists of determining a set of integrity constraints that a class collection must satisfy. For example, all instance variables should bear distinct names, no loops are allowed in the hierarchy, and so on.
2. A taxonomy of all possible updates is then established. These changes concern the structure of classes, like "add a method", or "rename a variable"; they may also refer to the hierarchy as whole, as with "delete a class" or "add a superclass to a class".
3. For each of these update categories, a precise characterization of its effects on the class hierarchy is given, and the conditions for its application are analysed. In general, additional reconfiguration procedures have to be applied in order to preserve

class invariants. It is, for example, illegal to delete an attribute from a class C if this attribute is really inherited from an ancestor of C. If the attribute can be deleted, it must also be recursively dropped from all subclasses of C.

4. Finally, the effects of schema changes are reflected on the persistent store; instances belonging to modified classes are converted to conform to their new description.

The major shortcoming of the class surgery approach lies in the fact that it gives no guidance as to why or when specific modifications should be performed.

Versioning consists in recording the major steps in class design and revision, as a way to deal with simultaneous updates to a hierarchy and to capture the intrinsic variations that exist in the modelling of an application domain, without loss of information. It enables teams of programmers to control the creation and dissemination of software components, and to try different paths in a coordinated way when modelling complex application domains.

With versioning, several issues have to be addressed [3]: structuring different kinds of versions (mutable/immutable, private/public); determining which version is used when an object of a particular class is instantiated; deciding what are the operations that justify the creation of new versions. In spite of the complexity and of the overhead incurred by versioning, this mechanism is considered indispensable for managing large object-oriented projects. However, it provides no indications regarding the specific modifications to apply to a library.

Reorganization of a class library is needed after significant, non-trivial changes are brought to it, like the introduction or the suppression of classes. Reorganization procedures use information on class structures to discover and correct imperfections in a library of classes.

Typically, reorganization procedures attempt to eliminate code redundancy from a hierarchy [10] or to reduce unwanted coupling between classes by suppressing certain kinds of inter-attribute dependencies — as in the Demeter approach [7]. Such techniques, originally based on very simple object models, have been improved by explicitly taking into account more features, notably the invariants to be preserved across transformations [6]. Although most of the principles behind proposed reorganization techniques can be stated as informal programming style guidelines, reorganization is the domain of choice for algorithms that automatically transform large libraries of inter-connected classes into improved hierarchies.

Current reorganization methods deal only with a small subset of the features found in object-oriented languages. They work globally, recasting an entire class collection at a time. However, an incremental approach would be more appropriate in the context of object-oriented programming, where libraries are built by successive enhancements to an existing system. Moreover, complete reorganizations are probably too extensive to be easily grasped by software designers. Finally, optimality of code reuse is not always the most desirable property in a hierarchy; analysing the design of classes and suggesting alternative structures are important as well for the software developer. This paper proposes a novel reorganization approach that addresses expressly these shortcomings. We first illustrate this approach with simple examples. We then describe the object model it is based on and explain in some detail the functioning of a new incremental reorganization algorithm. Finally, we examine the properties of the algorithm and evaluate its relevance for software engineering on the basis of its application to the Eiffel library.

3 Principles of Class Reorganization

Our hypothesis is that design flaws can be uncovered at the time a hierarchy is extended with an additional object description. The new class may refine or override the properties inherited from its ancestors. These redefinitions may correspond to inadequate application of object-oriented mechanisms for deriving the new subclass. They may also be caused by defective structures in the library of reusable components, which hinder the incorporation of properties inherited from superclasses into new applications. These redefinitions are analysed and the inheritance hierarchy is subsequently reorganized to improve subclassing patterns, to render existing classes more abstract, and to pinpoint the places warranting further redesign.

We distinguish two kinds of reorganizations: decomposition and restructuring.

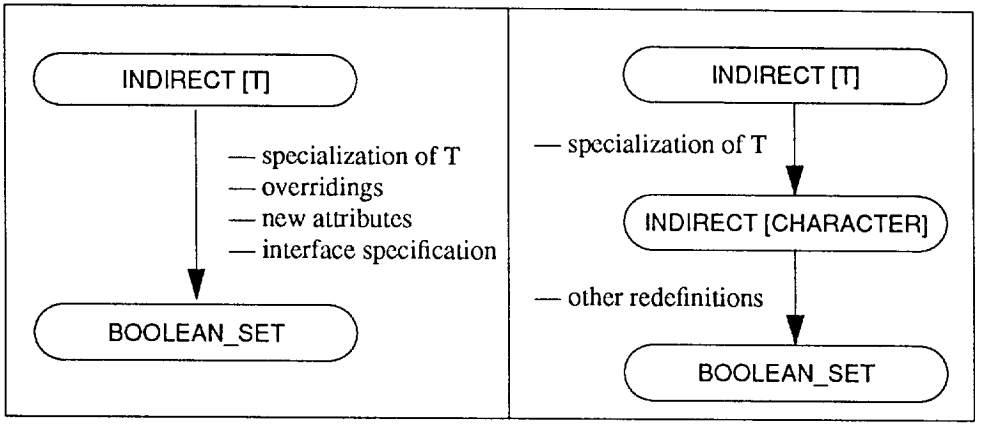

Fig. 1. Decomposing inheritance relationships.

A *decomposition* consists in breaking down various abstraction steps normally merged in a single inheritance link. The goals of this operation are to reduce the semantic overloading of inheritance links and to detect alternative modelling possibilities. Figure 1 depicts a fragment of the Eiffel hierarchy where class INDIRECT, which serves to store elements in main memory, gives rise to BOOLEAN_SET. The link between these classes, when decomposed as illustrated, highlights two usages of inheritance. In a first step, INDIRECT is specialized to store CHARACTER elements; in a second step, an attribute from INDIRECT is overridden, additional functionality is provided, and an interface for BOOLEAN_SET is specified. The inheritance link between BOOLEAN_SET and the auxiliary class added by the decomposition corresponds to an implementation dependency and could be substituted with a *part-of* relationship (requiring the introduction of a variable in BOOLEAN_SET, and the severing of the inheritance link); the latter representation is actually the one used in the Objective-C library for similar classes.

A *restructuring* operation extracts the properties shared by several classes and isolates them in a new, common ancestor. In figure 2, class CIRCLE inherits from ELLIPSE; this subclassing operation is accompanied with a partial replacement of ELLIPSE's behaviour. Simultaneously, CIRCLE changes its ancestor's interface in a way that corresponds neither to a

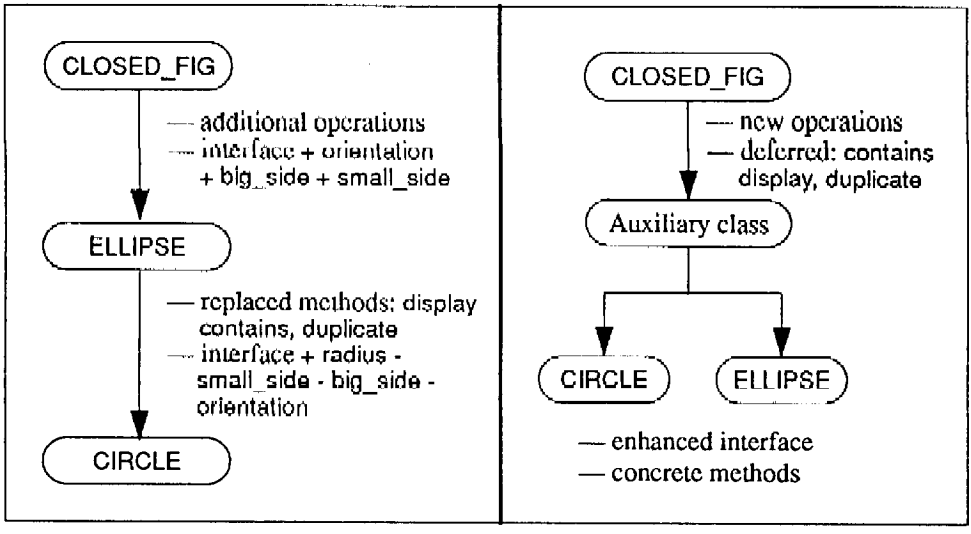

Fig. 2. Restructuring inheritance relationships.

restriction (which would be expected in a specialization relationship) nor to an extension (characteristic of subtyping relationships). A restructuring of the hierarchy eliminates this unnatural subclassing pattern by inserting an intermediate definition containing the properties common to both CIRCLE and ELLIPSE, and by making these two classes descendents from the new auxiliary node. Thus, the goals of restructuring are the factoring of commonalities and the enforcement of specific constraints regarding inheritance patterns. In the following sections, we concentrate on the restructuring algorithm.

4 An Incremental Restructuring Algorithm

Before we discuss the formal specification of our restructuring algorithm, it is useful to illustrate its working with a slightly more detailed example (figure 3). In our notation, capital letters denote attributes (variables or methods); underlined letters correspond to rejected (unwanted) attributes, and those printed in bold to attributes introduced by a class. Initially, the environment contains only three classes (***AB***, AB***C***, and ABC***D***) arranged in a trivial hierarchy. This hierarchy is extended with a new class AD***E*** which cannot be integrated directly without explicitly rejecting inherited properties. The library must be therefore reorganized to accommodate this new class. Our algorithm proceeds as follows:

1. We start from the initial configuration described above; AD***E*** inherits from ABC***D***.
2. Classes A<u>BC</u>D***E*** and ABC***D*** have attributes A and D in common. The algorithm creates a new node and transfers A and D to it. A is in fact inherited from class AB***C***, and cannot be separated from B and C at this stage. The algorithm is invoked recursively to reject B and C from the new intermediate class A<u>BC</u>***D***.
3. Classes A<u>BC</u>***D*** and AB***C*** have attribute A in common. An additional intermediate node is created. Its definition should be limited to attribute A, but because A and B

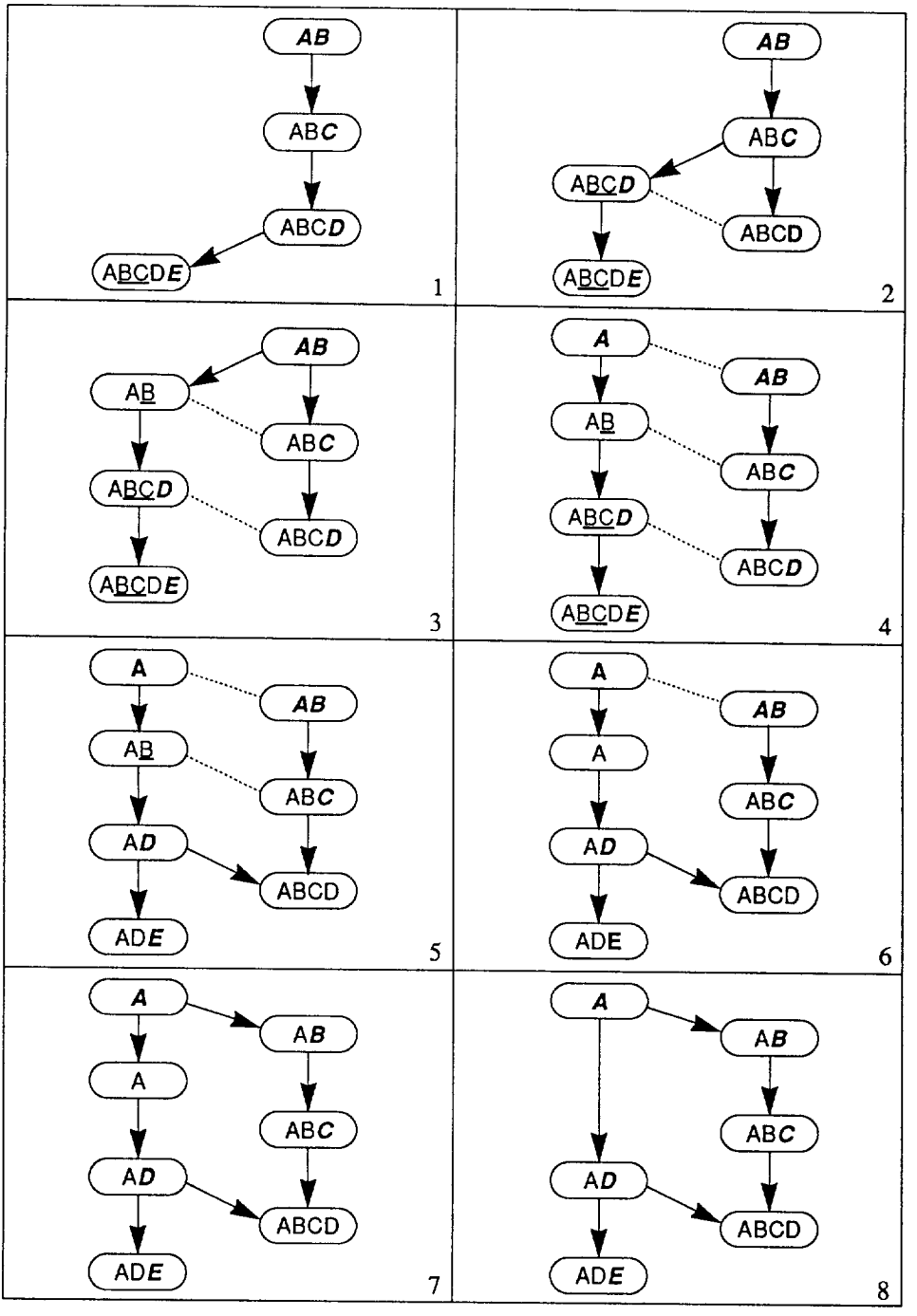

Fig. 3. The details of a complex restructuring example.

are both inherited from another superclass (***AB***), the latter attribute cannot yet be eliminated: the algorithm must therefore be applied to A<u>B</u>.

4. The expansion of the graph ends after getting rid of B (which is inherited from the top-most class), resulting in the creation of node ***A***. It is clear now that in its first pass, the algorithm proceeds by tracing unwanted attributes up the inheritance hierarchy until reaching the nodes where they are introduced. Each one of these classes is split in two parts, the first containing only accepted attributes (or more generally the properties common to the class whose introduction triggered the reorganization and to its ancestors), and the second the remaining ones.

   At this stage, the algorithm checks that the definition of every intermediate node created during the first upward pass is valid. This consists in verifying that there are no unresolved references in these classes — in other words, all attributes that introduced and inherited methods depend on must be present in the definition of the supplementary classes. If the validation succeeds, a traversal of the graph is performed in order to redirect inheritance links appropriately.

5. Restarting from the bottom level, inheritance links are redirected as follows: A***D*** contains attributes common to ABC***D*** and AD***E***, and becomes an ancestor for both classes.

6. Control passes back to the next level, where the appropriate modifications to inheritance structures are performed. In this case, class AB***C*** does not inherit from the new intermediate node A, since this attribute is already acquired, through another path, from A***B***. On the other hand, the class previously named A<u>BC</u>***D*** becomes a descendent of A; B and C are no longer present in its definition.

7. At the top level, class A***B*** now inherits attribute A from root class ***A***. The intermediate node created during the expansion phase (called A<u>B</u> in step 3) also inherits its definition from the root of the hierarchy — without the unwanted attribute B — yielding the graph represented in the seventh picture.

A last simplification phase suppresses all redundant nodes from the graph. In the example, class A is merged with its unique successor A***D***. The final illustration shows the new class AD***E*** with the additional nodes ***A*** and A***D*** needed for its integration in the hierarchy.

5 The Object Model

We base our approach on a formal model that includes classes, methods, and references between methods; this model is suitable for most languages providing multiple inheritance. A more complete object model dealing with variables, static inter-attribute dependencies and the structure of method signatures is possible, but its presentation would too long for this article. In our notation, $\mathcal{P}(s)$ denotes the powerset of s, while $x\ [c_1, \ldots, c_n]$ denotes the projection of x with respect to its components c_1 to c_n. Elements between $\langle \ldots \rangle$ denote tuples, and $\{\ldots\}$ represent sets.

We first define the set of classes: Class = $\{c_i\}$, the set of method signatures: Signature = Name × ArgList, with: Name = $\{m_i\}$, and ArgList = $\{a_i\}$. We use Body = $\{b_i\}$ to denote the set of procedure bodies that constitute possible implementations of a method. We distinguish a particular member of this set, which we call **deferred**,

to represent empty, "abstract" procedure bodies. A method is completely determined by a signature and an implementation: Method = Signature × Body . Methods are the only kind of attributes we consider in our model.

IntroMeth : Class $\rightarrow \mathcal{P}$ (Method) denotes the set of methods introduced by a class. Any attribute introduced by a class is distinguished by its name. We associate to each class the set of its direct ancestors: Ancestors : Class $\rightarrow \mathcal{P}$ (Class) . The structure deduced from inheritance links between classes is restricted to form a directed graph without circuits so that no class can directly or indirectly inherit from itself. It is convenient to define the immediate successors of a class: Descendents : Class $\rightarrow \mathcal{P}$ (Class) .

The methods inherited by a class from an immediate ancestor are given by: AncMeth : Class × Class $\rightarrow \mathcal{P}$ (Method) . We assume that attributes inherited from a superclass are uniquely identified by their name, and that attribute names allow one to distinguish between methods inherited from different ancestors. Moreover, the only possible redefinitions carried out on inherited methods deal with method bodies. Relaxing these constraints requires the introduction of schemes for solving naming conflicts and a more complete formal machinery [11].

The attributes effectively inherited by a subclass are:

$$\text{SuperMeth} : \text{Class} \rightarrow \mathcal{P}(\text{Method})$$

$$\text{where SuperMeth } (c) = \bigcup_{\forall (a \in \text{Ancestors } (c))} \text{AncMeth } (c, a)$$

The set of overridden attributes is: OverMeth : Class × Class $\rightarrow \mathcal{P}$ (Method) , where

$$\text{OverMeth } (c, a) = \{ m \in \text{AncMeth } (c, a) \mid m\,[\text{Name}] \in \text{IntroMeth } [\text{Name}] \}$$

Their corresponding redefinitions are: SubMeth : Class $\rightarrow \mathcal{P}$ (Method) , where

$$\text{SubMeth } (c) = \{ m \in \text{IntroMeth } (c) \mid m\,[\text{Name}] \in \text{AncMeth } [\text{Name}] \}$$

We choose to make all inherited attributes accessible, and we derive RefMeth : Class $\rightarrow \mathcal{P}$ (Method) , the set of attributes that may be referred to (through a method invocation) in a class definition, such that the following relation holds:

$$\text{RefMeth } (c) = \text{IntroMeth } (c) \cup (\cup_{\forall (a \in \text{Ancestors } (c))} \text{AncMeth } (c, a))$$

In our model, we consider that "deferring" or reimplementing an inherited concrete method are inappropriate redefinition patterns that should normally not occur in a hierarchy.

The attributes denoted by VisiMeth : Class × Class $\rightarrow \mathcal{P}$ (Method) can be passed to and become visible in a descendent further down in the hierarchy:

$$\text{VisiMeth } (c, a) = \text{AncMeth } (c, a) - \text{OverMeth } (c, a)$$

Inheritance extends a class with characteristics appearing in its ancestors, so its complete attribute set is:

$$\text{DefMeth } (c) = \text{IntroMeth } (c) \cup (\cup_{\forall (a \in \text{Ancestors } (c))} \text{VisiMeth } (c, a))$$

Naturally, AncMeth (c, a) = DefMeth (a).

SelfMethoDep: Class $\rightarrow \mathcal{P}$(Name) is the set of method names, to which a self (dynamically bound) reference is made in a method introduced by the class itself or inherited from one of its ancestors:

$$\text{SelfMethoDep}(c) = (\cup_{\forall (a \in \text{Ancestors}(c))} \text{SelfMethoDep}(a)) \cup \cup_{\forall (m \in \text{IntroMeth}(c))} \text{MethoDep}(m[\text{Body}])$$

with MethoDep: Body $\rightarrow \mathcal{P}$(Name) representing the set of method names a method body refers to. An obvious constraint is that he set of attribute names belonging to a class definition must contain the set of names that the class, or its ancestors, depend on. Thus, SelfMethoDep $(c) \subseteq$ DefMeth (c) [Name]. The **deferred** body exhibits the following natural property: MethoDep (**deferred**) = $\varnothing$.

The interface of a class lists all methods that are publicly accessible from outside an instance of the class (interfaces are not inherited):

$$\text{Interface}: \text{Class} \rightarrow \mathcal{P}(\text{Name})$$

$$\text{where naturally, Interface}(c) \subseteq \text{DefMeth}(c)\ [\text{Name}]$$

Conversely, Private (c) = DefMeth (c) [Name] − Interface (c).

The object model constitutes the framework for our incremental restructuring algorithm. On the basis of this model, we can define several constraints that the algorithm must satisfy:

- *Consistency.* No dangling references are allowed in a class definition.
- *Integrity.* The behaviour of all classes present in the hierarchy before the reorganization is launched must be preserved. In particular, a message sent to an object of a pre-existing class must still invoke the same method after a reorganization.
- *Redundancy.* An attribute should be introduced only at one point in the hierarchy.
- *Minimality.* The algorithm inserts the number of classes strictly required for rearranging the hierarchy. Limiting the number of new classes makes results easier to understand and improves the performance of the algorithm during subsequent runs.

6 Specifying the Restructuring Algorithm

The reorganization starts with class o, which is just being added to an existing hierarchy. O is a class specification with properties Public (o), IntroMeth (o), and Ancestors (o). O is a leaf of the hierarchy and has therefore no subclasses: Descendents $(o) = \varnothing$.

1. For each ancestor a_i of o, determine the set ProbMeth (o, a_i) of the names of the methods that are inherited by o and then overridden by the latter class in a non-legitimate way (rejections of inherited attributes are dealt with in a similar way):

   $$\text{ProbMeth}: \text{Class} \times \text{Class} \rightarrow \mathcal{P}(\text{Name})$$

   where, $\forall (a_i \in \text{Ancestors}(o))$

$$\text{ProbMeth}\ (o, a_i) = \{ n \in \text{Name} \mid (n = m_2\,[\text{Name}]) \wedge (m_2\,[\text{Name}] = m_1\,[\text{Name}]) \wedge (m_1 \in \text{AncMeth}\ (o, a_i)) \wedge (m_2 \in \text{IntroMeth}\ (o)) \wedge (m_2\,[\text{Body}] \neq m_1\,[\text{Body}]) \wedge \neg(m_2\,[\text{Body}] \neq \textbf{deferred} \wedge m_1\,[\text{Body}] = \textbf{deferred}) \}$$

It is evident from this definition that only methods superseded by o are considered for restructuring, and, among them, only those that do not follow the normal redefinition pattern prescribing that a deferred method may be changed to a non-deferred method.

In addition, we initially define the relation $\text{ProbAnc}\ (o, a_i) = \text{ProbMeth}\ (o, a_i)$.

2. We restructure the hierarchy along the inheritance paths leading to the ancestors of o affected by step 1. $\forall (a_i \in \text{Ancestors}\ (o) \mid (\text{ProbAnc}\ (o, a_i) \neq \emptyset))$, do steps 3 to 7 of the algorithm.

3. Let a_i be the current ancestor of o of interest. We create a new class s_i such that:

   — Methods introduced by a_i and not affected by the restructuring belong to s_i's definition. Thus, $\forall (m \in \text{IntroMeth}\ (a_i))$

   $$(m\,[\text{Name}] \notin \text{ProbAnc}\ (o, a_i)) \Rightarrow (m \in \text{IntroMeth}\ (s_i))$$

   — A generalization of the methods introduced by a_i and that must be restructured because of their redefinition in o is left in s_i. $\forall (m \in \text{IntroMeth}\ (a_i))$

   $$(m\,[\text{Name}] \in \text{ProbAnc}\ (o, a_i)) \Rightarrow (\langle m\,[\text{Signature}], \textbf{deferred} \rangle \in \text{IntroMeth}\ (s_i))$$

   In our case, the generalization consists simply in introducing a deferred representation of the method being restructured into s_i. Let us call GenMeth the set of names of all such methods, i.e.,

   $$\text{GenMeth}\ (s_i) = \{ n \in \text{Name} \mid n \in \text{ProbAnc}\ (o, a_i) \wedge n \in \text{IntroMeth}\ (a_i)\ [\text{Name}] \}$$

   — The ancestors of s_i are the same as for a_i: $\text{Ancestors}\ (s_i) = \text{Ancestors}\ (a_i)$

4. After this is done, we might still need to reorganize the ancestors of s_i themselves. We define $\forall (p_j \in \text{Ancestors}\ (s_i))$

   $$\text{ProbAnc}\ (s_i, p_j) = \text{ProbMeth}\ (s_i, p_j) \cup (\text{AncMeth}\ (s_i, p_j)\ [\text{Name}] \cap (\text{ProbAnc}\ (o, a_i) - \text{GenMeth}\ (s_i)))$$

   In other words, we consider:

   — All methods introduced by s_i that do not refine attributes inherited from p_j in legitimate ways.

   — Those methods that must be reorganized because of o's definition, but that are introduced by p_j and not a_i itself and so could not be reorganized at this level.

5. The algorithm is recursively applied to s_i starting from step 2, viewing s_i as o.

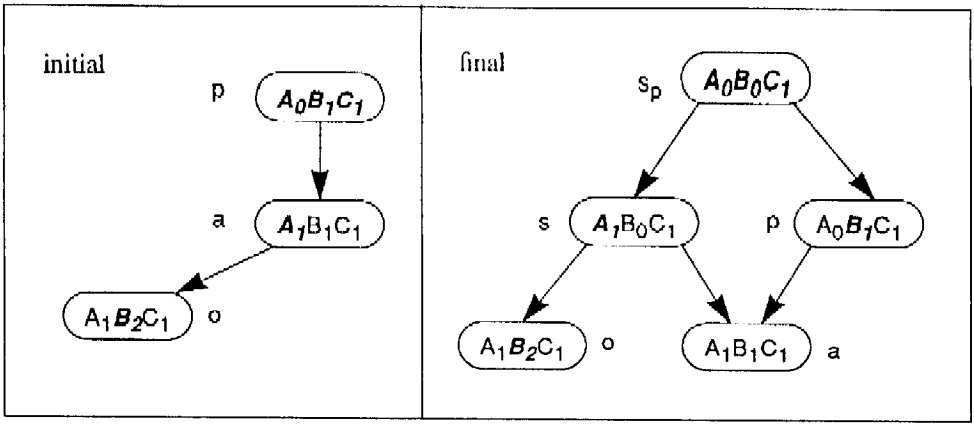

Fig. 4. The restructuring algorithm. We identify classes (o, a, p, s) in the same way as in the formal description of the algorithm. All capital letters denote methods; positively indexed letters represent different implementations of the same methods, 0 is reserved for deferred methods.

6. When the procedure returns, all ancestors of s_i have been reorganized, and only acceptable subclassing patterns connect s_i to its ancestors. We can therefore:

   — Eliminate duplicate definitions from s_i, i.e. those methods $m \in$ IntroMeth (s_i) such that $m \in$ SuperMeth (s_i). Note that, in general,

   $$\text{IntroMeth } (o) \text{ [Name]} \cap \text{SuperMeth } (o) \text{ [Name]} \neq \emptyset.$$

   — Determine the interface of s_i as being the largest possible with respect to o and a_i:

   $$\text{Public } (s_i) = (\text{DefMeth } (s_i) \text{ [Name]} \cap \text{Public } (o)) \cup (\text{DefMeth } (s_i) \text{ [Name]} \cap \text{Public } (a_i))$$

   The idea is that all methods introduced by s_i and appearing in o's or a_i's interfaces should already be present in s_i's list of public attributes. An alternative is to take the greatest interface common to both o and a_i:

   $$\text{Public } (s_i) = \text{DefMeth } (s_i) \text{ [Name]} \cap \text{Public } (o) \cap \text{Public } (a_i)$$

   — Redirect the inheritance link connecting o and a_i to point from o to s_i:

   $$(a_i \notin \text{Ancestors } (o)) \wedge (s_i \in \text{Ancestors } (o))$$

7. We now determine the relationship between a_i and s_i; a_i should become a descendent of s_i if and only if both classes share common properties, that is:

   — s_i effectively introduces some attributes: IntroMeth $(s_i) \neq \emptyset$.

— or s_i and a_i share identical ancestors: Ancestors (a_i) = Ancestors (s_i) .

If either of these conditions is satisfied, then a_i is made a subclass of s_i. This entails that a_i's definition is modified as follows:

— The methods previously inherited by o from a_i without redefinition no longer need to be introduced by a_i; they are deleted from the latter class's definition: $\{m \in \text{IntroMeth}\ (a_i) \mid (m\,[\text{Name}] \notin \text{ProbAnc}\ (o, a_i))\}$.

— However, the redefinitions carried out by a_i on s_i's methods remain in a_i: $\{m \in \text{IntroMeth}\ (a_i) \mid (m\,[\text{Name}] \in \text{ProbAnc}\ (o, a_i))\}$.

As a result, we have IntroMeth $(a_i) \cap$ IntroMeth $(s_i) = \varnothing$.

— The ancestors of a_i that have not been restructured in s_i are discarded from a_i's definition, so that Ancestors $(a_i) \cap$ Ancestors $(s_i) = \varnothing$.

— s_i is made a superclass of a_i: $s_i \in$ Ancestors (a_i) .

8. Finally, we eliminate duplicate attributes from o's definition as explained in step 6.

The restructuring process might create some superfluous classes. The unnecessary nodes of the inheritance graph can be eliminated with additional procedures. A transitive reduction step makes sure that classes inherit only from the most specific ancestors (i.e. the classes with the largest number of most specialized attributes). The second simplification merges each new class with its descendent, if it is unique. Every remaining class is then merged with its ancestor, if it is unique, and if the class being examined does not introduce any property. The final and trivial step consists in discarding all auxiliary classes without descendents.

7 Properties

Consistency. For a particular s_i, it is evident that the deferred methods it introduces do not depend on anything. The concrete methods may refer to a method introduced by the corresponding a_i — in which case either a concrete or a deferred representation of the method referred to also belongs to s_i. If a concrete method depends on an inherited attribute, then the dependencies are satisfied too, since s_i initially inherits from the same ancestors as its corresponding a_i.

Minimality. The algorithm creates one additional class for each ancestor examined during the traversal of an inheritance path. The simplification phase ensures that redundant inheritance links and intermediate classes are suppressed from the hierarchy. The total number of links traversed or nodes created, and hence the complexity of the algorithm, depend directly on the configuration of the inheritance graph. For example, the complexity of restructuring an inheritance tree is linear in the depth of the tree. For other kinds of graphs, the complexity of the algorithm may grow exponentially. During its expanding phase, the algorithm actually creates many more classes than are necessary for a proper restructuring. The simplification procedure is therefore essential to avoid a proliferation of densely interconnected classes, which quickly leads to unacceptable performance and an incomprehensible inheritance structure. In any case, the algorithm is guaranteed to terminate in finite time for every

inheritance path traversed; because no circuits are allowed in the inheritance graph, the algorithm cannot loop forever while trying to restructure a class that inherits from itself.

Redundancy. Although our algorithm does not introduce redundancy into a hierarchy, it is insufficient for suppressing all attribute duplications present in a graph prior to the reorganization. The restructuring procedure operates on local information for achieving the reorganization; it cannot determine whether a particular attribute being examined is introduced somewhere else in the hierarchy. This becomes impossible without a complete inspection of the whole graph when identical attributes appear in two disjoint subclassing paths. It can be proved however that a variant of the incremental algorithm produces the same result as a global reorganization when applied to certain forms (typically trees) of initially non-redundant hierarchies.

Integrity cannot be guaranteed without some assumptions on how class definitions are constructed in the presence of naming conflicts caused by multiple inheritance [11]. It is possible to maintain the integrity of classes by resorting to statically bound methods, to inheritance prioritization schemes as the one used in the CLOS system, and to auxiliary forwarding methods.

8 Application: Reorganizing the Eiffel Library

It is important to determine whether the results produced by the reorganization algorithm are really meaningful or if they are just a consequence of our automated approach. A first

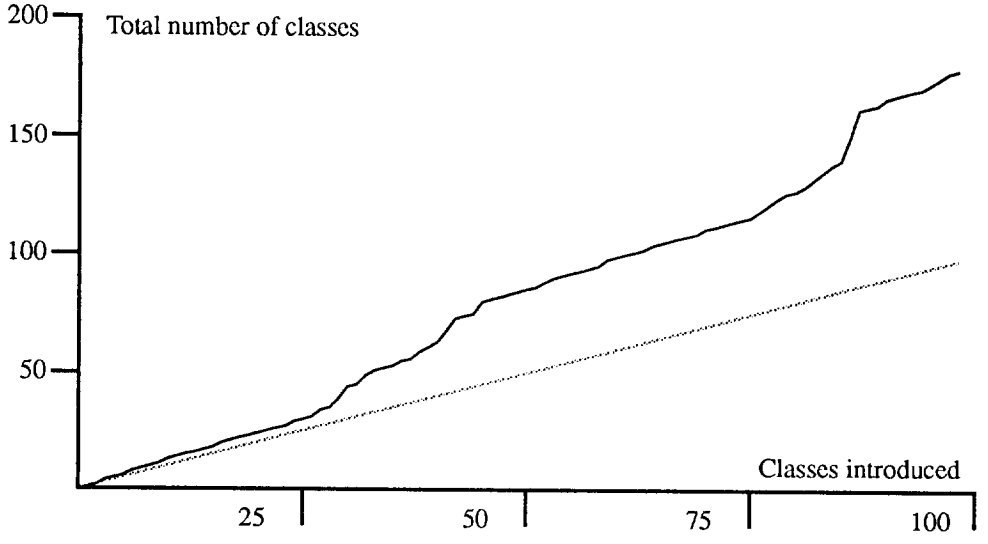

Fig. 5. Restructuring the Eiffel library with respect to class interfaces.

insight into this issue is given by the graphs appearing in figures 5 to 8. These graphs were obtained by applying the reorganization algorithms to version 2.1 of the Eiffel library. Eiffel classes were first ordered according to a topological sort; they were then introduced one after the other into an initially empty hierarchy, which was reorganized after the insertion of

each new class. This procedure allows us to simulate the construction of the library, and to observe the behaviour of long series of incremental reorganizations. The straight line in the graphs serves to compare the evolution of the hierarchy without reorganization with the actual increase in the number of classes.

Figure 5 illustrates the result of incrementally restructuring the Eiffel library when considering only the interfaces of the classes (the peculiarities of interface restrictions afforded by the Eiffel language are always properly handled). The constraint to enforce stipulates that the interface of a subclass must be as extensive as the interfaces of its superclasses; in other words, the subclass must provide as many services as its ancestors. As can be easily inferred from the diagram, Eiffel does not behave gracefully when restructured according to this criterion — which, as a matter of fact, did not guide the implementation of the library. It is interesting to note that analyses of fragments of the Smalltalk library have uncovered similar discrepancies between the actual hierarchy and an inheritance graph based on interface inclusion relationships.

Figure 6 also illustrates the result of restructuring the hierarchy, but this time considering

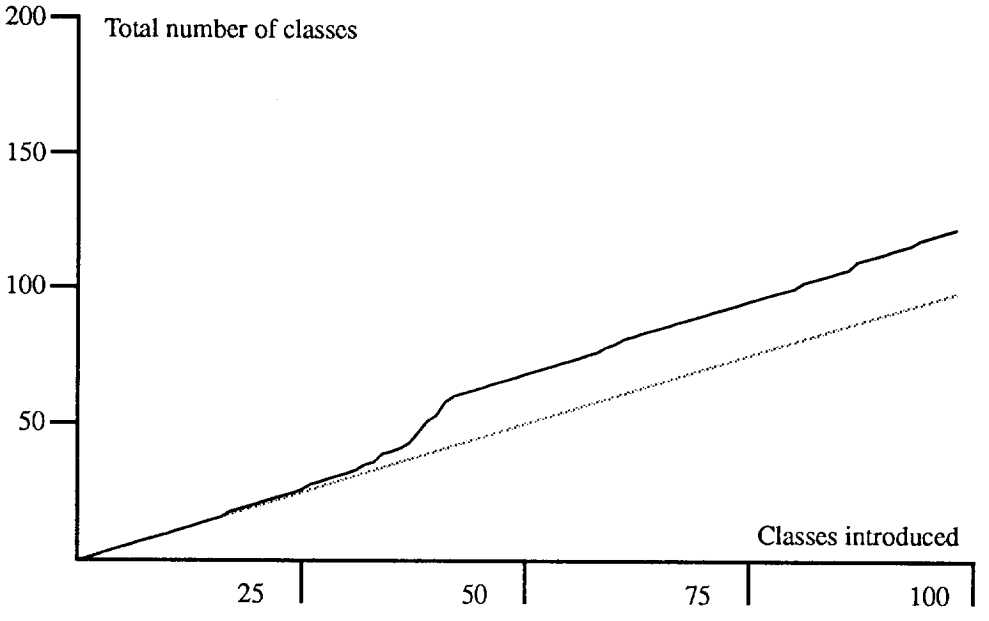

Fig. 6. Restructuring for maximum code sharing.

code sharing as the criterion to drive the reorganization. Although code factorisation in the Eiffel library seems well done in general, there is a small zone in the graph were the introduction of classes causes some important disturbances. The alterations carried out on the inheritance graph pinpoint inadequate abstractions. In particular, table 1 uncovers a problem with classes deriving from lists. Notice for example that TWO_WAY_TREE replaces properties inherited from LIST, an ancestor located three levels higher up in the hierarchy. A closer examination of the most suspicious classes is revealing: a comment in the source code of SORTED_LIST explicitly qualifies the inheritance link between this class and LINKED_LIST

Rank	Introduced class	Auxiliary classes	Ancestors	Distance
17	BOOLEAN_SET	1	INDIRECT	1
26	FIXED_LIST	1	LIST	1
32	INTBINTREE	1	BINSRCH_TREE	1
34	LINKED_LIST	2	LIST	1
37	INPUT	1	GEN_INPUT	1
38	SORTED_LIST	3	LINKED_LIST	1
			LIST	2
39	TWO_WAY_LIST	3	LINKED_LIST	1
			LIST	2
40	BI_LINKABLE	1	LINKABLE	1
41	TWO_WAY_TREE	4	LIST	3
42	GEN_WINDOW	1	RECT_SHAPE	1
59	LINKED_QUEUE	1	QUEUE	1
61	FIXED_STACK	1	STACK	1
81	CIRCLE	1	ELLIPSE	1
87	GTEXT	2	GEN_FIGURE	2
94	TRIANGLE	1	POLYGON	1

Table 1. Restructuring operations for code sharing. For each class causing a reorganization, we indicate when it is introduced, which of its ancestors are redefined in such a way as to warrant a reorganization, the number of inheritance links to traverse before reaching these ancestors, and how many auxiliary classes are added to the hierarchy.

as an improper relationship; TWO_WAY_TREE inherits from TWO_WAY_LIST for code sharing purposes; LINKED_LIST redefines attributes of LIST to take advantage of a new representation; and TWO_WAY_LIST overrides methods acquired from LINKED_LIST for efficiency reasons. In the first two cases, inheritance links should be replaced with other kinds of relationships, while the last two situations seem to indicate that LIST and LINKED_LIST are insufficiently abstract. On the other hand, several reorganizations, and their associated auxiliary classes, can be considered as "noise" — as is the case for the restructurings affecting INTBINTREE or GTEXT.

Figure 7 shows the result of decomposing the inheritance links with respect to class interfaces. We do this by comparing the interface of each class with those of its direct ancestors. We first aggregate the interfaces of superclasses for the comparison (solid curve); then, we study every individual inheritance link separately (dotted curve). We choose to group the various interface redefinitions in four categories: equality (no change in inherited interfaces), restriction (corresponding to a specialization), extension (the converse relationship), and replacement. Intuitively, the latter relationship denotes either a mixture of specialization

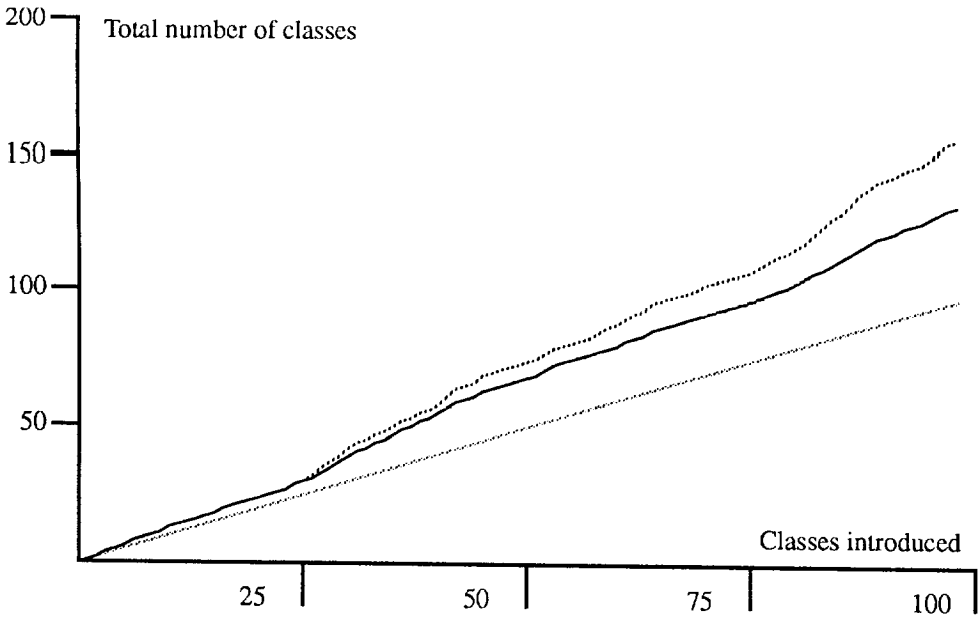

Fig. 7. Applying the decomposition procedure with respect to class interfaces.

and subtyping in the inheritance links, or a code sharing relationship. Some decompositions may also correspond to a lack of consistency in the terminology used for defining classes and their services [8], which a detailed analysis of renaming patterns would uncover. It is of course possible to combine decomposition and restructuring, as depicted in figure 8, and thus to obtain a dynamic measure of the overall quality of the Eiffel library.

9 Evaluation and Future Issues

The experience gained by reorganizing inheritance hierarchies manually also gives interesting insights into the applicability of our class restructuring algorithm. Anderson and Gossain, reporting in [1] on the development of a class library for encapsulating VLSI routing algorithms, made observations that match closely the reorganization patterns of our algorithms:

> «One pervasive pattern is that of splitting, where a class is split into a new class and subclass. There are two complementary processes here, of specialisation and generalisation. ... In generalisation, a class becomes more abstract, shifting upwards in the hierarchy. ... In specialisation, more responsibility and functionality is added to a class.»

We expect reorganization algorithms to fulfil a significant role as an interactive tool for object-oriented design and maintenance, in association with the other class evolution approaches discussed in section 2. A typical scenario for class development begins when a programmer extends a class library with additional components — perhaps by relying on tailoring to adapt existing functionality to define his new classes. He then reorganizes his

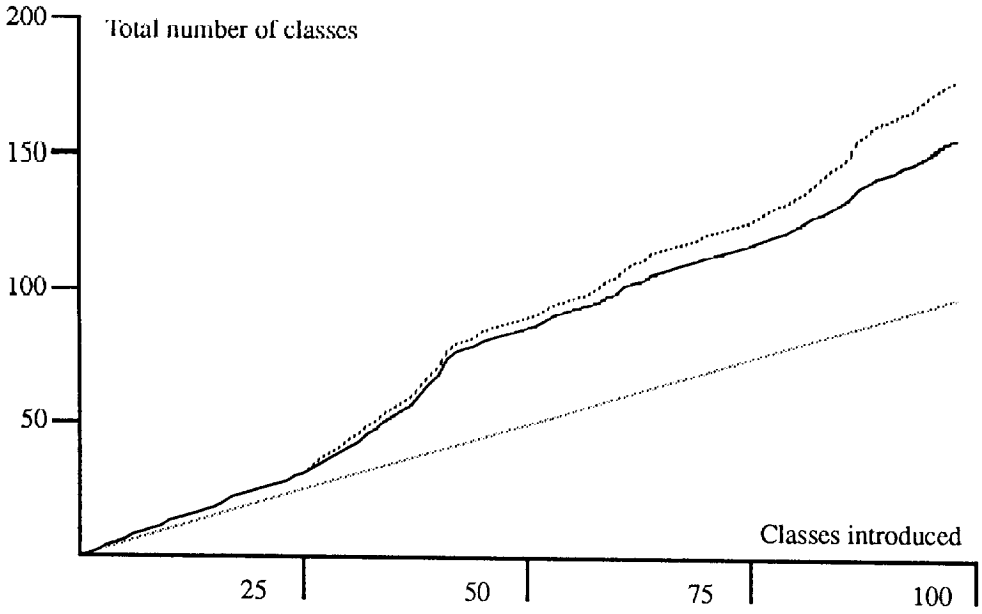

Fig. 8. Combining the restructuring and the decomposition of inheritance links.

hierarchy incrementally, on the basis of various criteria, in order to detect the places in the hierarchy most likely to require supplementary revisions. Decompositions help him uncover improper subclassing patterns and detect alternative ways to model software components — for example by using genericity or delegation instead of inheritance; the results of restructurings serve as rough estimates for the abstractions or for the "mixins" — classes whose purpose is not to describe real-world entities, but rather to support the implementation of other classes — that are missing from the hierarchy.

Because reorganization algorithms perform strictly structural transformations on object descriptions, their results require user intervention to compensate for the lack of knowledge concerning the application domain and the concepts embodied in the class collection. It is up to the software developer to inspect the outcome of the reorganizations, to adjust them with class surgery primitives, and perhaps to embark on more comprehensive restructuring activities. Thus, automatic reorganization provides for a coarse view of how the entire inheritance graph should be updated before fine-grained modifications are applied to selected aspects of class definitions.

The results of different reorganizations and their subsequent adjustments can be kept as versions of the hierarchy, which can be further modified or cancelled by the programmer. When a satisfactory design for the new component and its related classes is achieved, it can be frozen and publicly released as the new version of the class library, while the other working versions of the hierarchy are discarded.

Inheritance is not the only abstraction mechanism available in object-oriented languages, and it would be very productive to provide a tool that is able to select among several mod-

elling mechanisms — such as changing a subclassing dependency to a delegation relationship automatically. There are also important issues related to the preservation of class behaviour across reorganizations which have not yet been adequately addressed. A formal framework for reorganizing hierarchies, similar to the normal forms available in the relational data model, is still lacking; our algorithms are a step towards achieving this goal.

Acknowledgements

This research was carried out at the Centre Universitaire d'Informatique in Geneva. The financial support from the UBS Informatics Laboratory (UBILAB) is gratefully acknowledged.

References

1. B. Anderson, S. Gossain: Hierarchy Evolution and the Software Lifecycle. In: J. Bézivin, B. Meyer, J.-M. Nerson (eds.): Proc. 2nd TOOLS Conference. Paris, 1990, pp. 41–50

2. J. Banerjee, W. Kim, H.-J. Kim, H. F. Korth: Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In: SIGMOD Record (special issue on SIGMOD '87) 16(3), 311–322 (December 1987)

3. A. Björnerstedt, C. Hultén: Version Control in an Object-Oriented Architecture. In: W. Kim, F. H. Lochovsky (eds.): Object-Oriented Concepts, Databases, and Applications. Frontier Series. Addison-Wesley/ACM Press, 1989, pp. 451–485

4. D. C. Halbert, P. D. O'Brien: Using Types and Inheritance in Object-Oriented Programming. IEEE Software, 71–79 (September 1987)

5. R. E. Johnson, B. Foote: Designing Reusable Classes. Journal of Object-Oriented Programming, 22–35 (June-July 1988)

6. K. J. Lieberherr, P. Bergstein, I. Silva-Lepe: Abstraction of Object-Oriented Data Models. In: H. Kangassalo (ed.): Proc. 9th Entity-Relationship Conference. Lausanne: 8–10 October 1990, pp. 81–94

7. K. Lieberherr, I. Holland, A. Riel: Object-Oriented Programming: an Objective Sense of Style. SIGPLAN Notices (special issue on OOPSLA '88) 23(11), 323–334 (November 1988)

8. B. Meyer: Tools for the New Culture: Lessons from the Design of the Eiffel Libraries. CACM 33(9), 68–88 (September 1990)

9. D. J. Penney, J. Stein: Class Modification in the GemStone Object-Oriented DBMS. SIGPLAN Notices (special issue on OOPSLA '87) 22(12), 111–117 (December 1987)

10. W. W. Pun: A Design Method for Object-Oriented Programming. PhD thesis. Department of Computer Science, University College London. London: 1990

11. E. Waller: Schema Updates and Consistency. In: C. Delobel, M. Kifer, Y. Yasunaga (eds.): DOOD '91 Proceedings. Lecture Notes in Computer Science 566, Springer December 1991, pp. 167–188

12. R. Zicari: Schema Updates in the O2 Object-Oriented Database System. Technical report 89–057. Politecnico di Milano, Dipartimento di Elettronica. Milano: 31 October 1989