III. The Automatic Reorganization of Object Oriented Hierarchies

A Case Study

Eduardo Casais

Abstract

Software components developed with an object-oriented language require frequent revisions before they become stable, reusable classes. Class evolution is a complex task, and appropriate support in the form of tools and methodologies is required to help software engineers redesign object-oriented libraries. Recently, several approaches have been proposed to restructure inheritance hierarchies, to discover missing abstractions and to enforce programming style guidelines. We describe such an approach and examine in detail the results of its application to the Eiffel library. This analysis gives interesting insights into the suitability of automatic class reorganization for object-oriented software engineering.

3.1 Object-Oriented Design and Redesign

3.1.1 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 [9]. It is generally assumed that these powerful mechanisms, together with a comprehensive set of interactive programming tools provide the basic functionality required for the large-scale production of highly reusable software components.

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 Eiffel library incurred major redesigns [15], in spite of accumulated and documented experience in building comparable libraries with other programming languages — an unequivocal sign that relying on the object-oriented approach does not suffice for achieving reusable designs. Similar problems have been reported about the development of class collections for various application domains ranging from VLSI design algorithms [1] to operating systems [16].

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 [10]. Moreover, the object orientation 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 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 [11][15].

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.

3.1.2 Managing Class Evolution

Several approaches have been proposed in the recent years to control evolution in object-oriented systems [6].

The simplest one consists in adjusting class definitions when they do not lead to easy subclassing. The assumption is that the functionality of a hierarchy can be tailored to derive new classes without actually changing existing definitions. Some 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) [10][14]. Tailoring is however of limited value when it comes to the redesign of a class collection, and its undisciplined use quickly leads to incomprehensible subclassing structures.

The problem of modifying class definitions in a consistent way has been extensively investigated in the area of object-oriented databases [2][3][17][19]. The proposed approaches decompose all modifications that can be brought to class structures and relationships into primitive update operations — like "add a method", "rename a variable" or "add a superclass to a class". For each primitive, a precise characterization of its effects on the class hierarchy is given; thus, deleting an attribute from a class *C* results in the suppression of this same attribute in all subclasses of *C* and in invalidating all methods accessing this attribute. The objective is to enforce a pre-defined set of integrity constraints on the class library — such as making sure that all instance variables bear distinct names, or that no loops occur in the inheritance graph. In a final step, the effects of schema changes are reflected on the persistent store: instances belonging to modified classes are converted to conform to their new description.

This approach has proved to be extremely valuable for defining consistency-preserving class modifications; its major shortcoming lies in the fact that it gives no guidance as to why or when specific updates 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. With versioning, several issues have to be addressed [4]: 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 indispensable for managing large object-oriented projects [8]. However, it provides no indications regarding the specific modifications to apply to a library.

Extensive transformations of a class library are needed after non-trivial changes are brought to it, like the introduction or the suppression of classes. Information on class structures has to be extracted and analysed to discover and correct mod-

elling imperfections in the library. So far, reorganizations have been generally carried out manually, but these manual restructuring procedures, stated as informal programming style guidelines, are often amenable to automation.

- The Law of Demeter addresses the issue of eliminating certain kinds of unwanted dependencies among classes, and between methods and the objects (variables and parameters) they manipulate. It can be implemented as a utility that checks programs for violations of the law and converts the offending sources to a legal form [13].
- Algorithms have been developed to produce optimal inheritance graphs — without attribute redundancy and with the minimum number of classes and inheritance links — from pre-existing hierarchies. These techniques are based on formal models of inheritance; they work globally, recasting an entire class collection at a time. Although they deal only with a small subset of the features found in object-oriented languages, their characteristics regarding theoretical performance and optimality of results can be stated precisely [12].
- The aforementioned modification primitives from the database field focus on structural integrity constraints; refactoring enhances them with behaviour-preserving capabilities. In addition, refactoring provides higher-level modification operations such as distributing the functionality of a class over multiple subclasses (notably by splitting methods along conditional statements), or transforming an inheritance relationship into a component relationship [16].
- Incremental reorganization algorithms analyse the inheritance relationships between a class newly introduced in a hierarchy and its ancestors; the hierarchy is then automatically transformed to eliminate unwanted subclassing patterns, to pinpoint places requiring a redesign and to discover missing abstractions. These algorithms are based on a comprehensive object model; they attempt to optimize the inheritance graph within reason while keeping the disturbances to the original library to a minimum [5].

Given that most of these methods have not yet found their way into commercial object-oriented CASE products, there are only few, if any, case studies on the automatic reorganization of class hierarchies. This paper presents the results of applying incremental restructuring algorithms to the Eiffel library.

Incremental reorganization algorithms seem particularly appropriate in the context of object-oriented programming, where the mechanism of inheritance

plays such a central role and where libraries are built by successive enhancements to an existing system. Adding a subclass is a major turning point in the development process, warranting an evaluation and, possibly, an improvement of the existing hierarchy. As a matter of fact, extracting abstractions common to several concrete application classes and subsequently rearranging the inheritance hierarchy is viewed as an essential step during object-oriented development [1][11][15]. Analysing the behaviour of incremental restructuring methods should therefore provide interesting information on the general suitability of such techniques as object-oriented design tools.

3.2 Incremental Class Reorganization

3.2.1 Principles and Examples

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 adapted to improve subclassing patterns.

We distinguish two kinds of reorganizations: decomposition and factorisation.

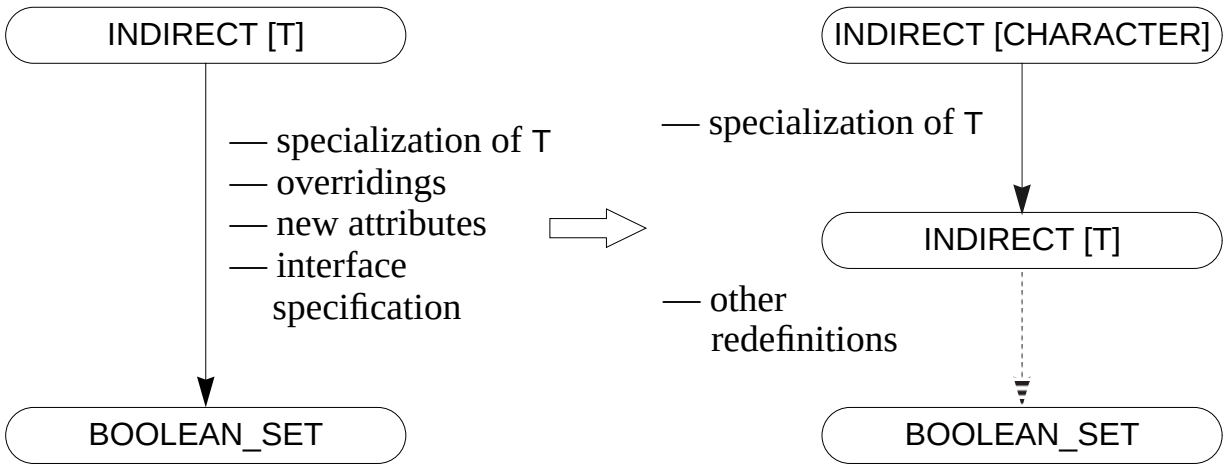


Figure 1 Decomposing inheritance relationships.

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 2.1 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).

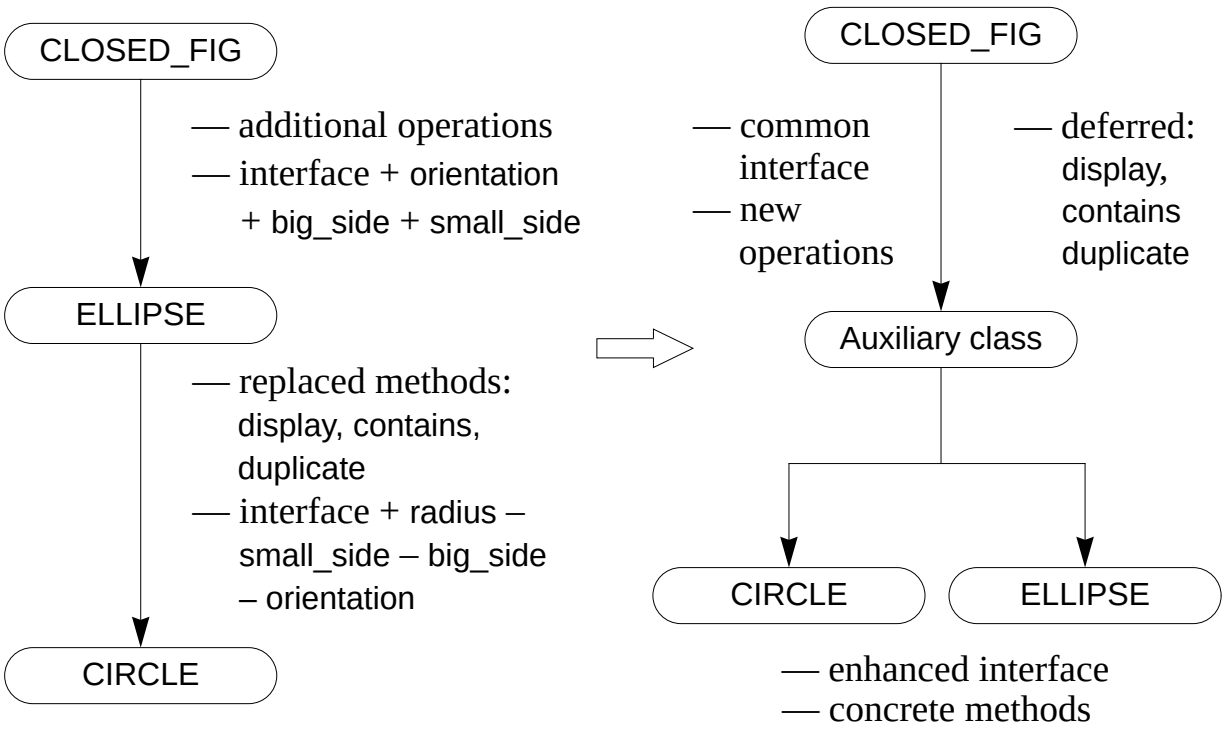


Figure 2 Factorising inheritance relationships.

A *factorisation* 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 restriction (which would be expected in a specialization

relationship) nor to an extension (characteristic of subtyping relationships). A transformation 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.

3.2.2 The Factorisation Algorithm

Since factorisation forms the core of our incremental restructuring approach, it is useful to illustrate its working with a slightly more detailed example (figure 3); the detailed algorithm, its properties and the object model it is based on are discussed elsewhere [5]. In our notation, capital letters denote attributes (variables or methods); underlined letters correspond to rejected (fully overridden) attributes, and those printed in bold to attributes introduced by a class.

Initially, the environment contains the classes denoted by ***B***, ***A***B, B***C***, and ABC***D***. This hierarchy is extended with a new definition AC***E*** which cannot be integrated directly without explicitly rejecting inherited attributes from some other class. The library must be therefore adjusted to accommodate this new class. Our algorithm proceeds as follows:

1. Examine the redefinitions carried out by the newly inserted subclass with each of its direct ancestors and determine unwanted redefinition patterns. In our case, these patterns correspond to fully overridden (rejected) attributes.
2. Factor the properties common to each ancestor and the subclass — that is, the set of attributes belonging to the ancestor and not rejected by the subclass — into an auxiliary definition. There is thus one such auxiliary definition for each ancestor.
3. Link the new auxiliary definitions to the ancestor's superclasses, and then propagate the reorganization upwards — by a recursive invocation of the algorithm on the auxiliary node — to get rid of remaining illegal subclassing patterns.
4. Redirect inheritance links from each ancestor to point to its companion auxiliary node. Because a class could reject all properties it acquires from an ancestor, this redirection occurs only when the auxiliary node actually represents a definition common to the subclass triggering the reorganization and its factorised ancestor.

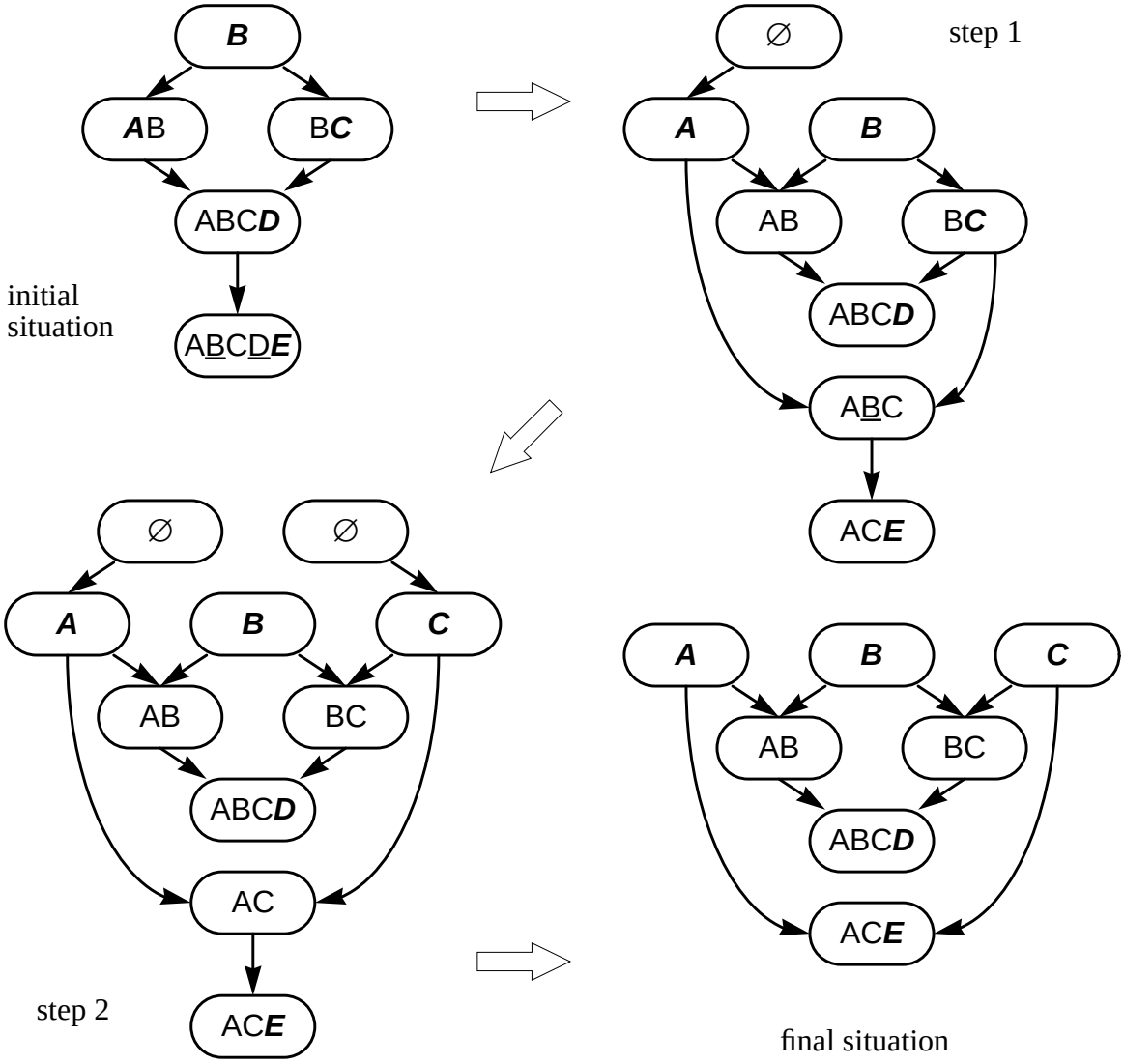


Figure 3 The details of a complex restructuring example.

In step 1 of the example, the algorithm creates class ABC, which represents the definition common to ABC***D*** and AC***E***. At this stage, attribute D is no longer acquired by AC***E*** from its ABC***D*** superclass. No inheritance link is established between ABC and ABC***D***, since the latter class already acquires attributes A and C through other inheritance links. The factorisation is then propagated upwards along the left branch of the inheritance chain leading to ABC. A new class ***A*** is introduced and becomes an ancestor to both ABC and ***A***B. Finally getting rid of attribute B results in the introduction of an empty definition at the top of the hierarchy — corresponding to the common part between ***A*** and ***B***. The right branch of the inheritance path leading to ABC is then traversed, also to eliminate attribute B from AC***E***'s definition, giving the graph depicted in step 2.

A last simplification phase suppresses redundant nodes and links from the graph. In the example, class AC is merged with its unique successor AC***E***, and both empty root classes are eliminated. Other cases to consider include merging an empty auxiliary class with its ancestor, if it is unique, reducing the inheritance relationships of intermediate classes so that they inherit only from their most specific ancestors, and discarding auxiliary definitions without descendents. The final illustration shows the new class AC***E*** with the additional nodes ***A*** and ***C*** needed for its integration in the hierarchy.

It should be noted that the more general factorisation procedure extracts more complex information from subclassing patterns to build the auxiliary nodes [6]. The redefinition of method signatures, of variable types and of class interfaces can be analysed to build intermediate representations comprising the common class interface subset, variables with the same name but a more general type, or methods with identical signatures but a deferred implementation.

3.3 Restructuring the Eiffel Library

3.3.1 Method and Objectives

In order to assess the potential of our reorganization approach, our algorithms were implemented in LISP and applied to the Eiffel library. Eiffel is an interesting case to study: the language was designed around high-level constructs and with an elegant syntax, in order to promote good programming practices; the library has undergone major redesigns, which have been partly documented [15]; and its designers' attitude towards object-oriented design often departs from mainstream approaches.

First, various utility programs were used to parse the source code of all classes in the library and to transform it into a representation suitable for manipulation by the LISP program. The features of the Eiffel language dealt with include classes, their inheritance relationships, their interfaces — including the scope restrictions afforded by the Eiffel language —, and their attributes — variables and methods. The structure of attributes — method parameter lists, variable types, and inter-attribute references — were not handled; we only determined were attributes are introduced and overridden in the hierarchy. In a second step, Eiffel classes were ordered according to a topological sort based on inheritance relationships and on interface scope restrictions. The classes were then introduced one after the other into an initially empty hierarchy, which was incrementally restructured after the in-

sertion of each new definition. This procedure allows us to simulate the construction of the library, and to observe the behaviour of long series of incremental reorganizations. The results of these reorganizations were manually compared to the original source code and evaluated. In addition, several statistics on the reorganization process and on the structure of the hierarchy were produced during various runs of the program.

This investigation should enable us to answer a number of questions:

1. Do automated techniques produce really meaningful results?
2. Do they provide a useful measure of the quality of an object-oriented library, especially as it evolves through successive stages of improvements and redesigns?
3. To what extent does our incremental restructuring approach cover the interesting situations warranting a redesign?
4. What does the overall process teach us in the domain of object-oriented design?

The remainder of this paper examines each one of these problems in turn. The conclusion assesses the relevance of automatic reorganizations for object-oriented software engineering.

3.3.2 Decomposing Class Interfaces

Encapsulation is one of the essential characteristics of object-oriented programming: objects can be manipulated only through the operations appearing in their interface, as defined by the class to which they belong [7][10]. All other attributes remain hidden inside objects. Interfaces, method signatures and pre and post-conditions are instrumental in enforcing a discipline of “programming by contract” and in achieving modularization [14]. As a consequence, interfaces, together with inheritance relationships, should suffice to get a good understanding of how functionality is organized in an object-oriented library. However, this assumption holds only if the relationships between the interface of a subclass and those of its ancestors follow some regular patterns. We distinguish four such patterns:

1. Equality:

$$\text{Interface (subclass)} = \bigcup_{s \in \text{ Superclasses (subclass)}} \text{Interface ('s)} .$$

   This pattern often corresponds to the implementation of abstract functionality in a concrete subclass.

2. Extension:

$$\text{Interface}\,(\text{subclass}) \supset \bigcup_{s \in \text{ Superclasses}\,(\text{subclass})} \text{Interface}\,(s)\,.$$

   This pattern corresponds most closely to subtyping relationships.

3. Restriction:

$$\text{Interface}\,(\text{subclass}) \subset \bigcup_{s \in \text{ Superclasses}\,(\text{subclass})} \text{Interface}\,(s)\,.$$

   Restriction corresponds to specialization relationships.

4. Code sharing; this relationship occurs when the services advertised by the subclass have nothing in common with those of its ancestors:

$$\text{Interface}\,(\text{subclass}) \cap \bigcup_{s \in \text{ Superclasses}\,(\text{subclass})} \text{Interface}\,(s) = \varnothing.$$

There is actually a fifth pattern, covering mixed relationships; this last case denotes either code sharing or a mixture of various other subclassing patterns. Contrarily to pure code sharing however,

$$\text{Interface}\,(\text{subclass}) \cap \bigcup_{s \in \text{ Superclasses}\,(\text{subclass})} \text{Interface}\,(s) \neq \varnothing\,.$$

We admit only simple interface redefinition patterns: all inheritance links corresponding to the last (mixed) kind of relationship must therefore be decomposed into the simpler ones. On the other hand, we allow a chain of subclasses to contain different patterns: our goal is not to enforce homogeneity in class relationships, but rather to render these relationships straightforward.

The outcome of the decomposition process is depicted in figure 4 for version 2.1 of Eiffel. The curve represents the accumulated number of auxiliary classes introduced as the library grows and is reorganized. As can be easily inferred from the diagram, Eiffel does not behave gracefully when restructured according to this criterion: the whole library of 98 classes requires 35 additional classes to be reorganized. Actually, the designers of Eiffel acknowledged that version 2.1 of the system lacked consistency in the terminology used for naming classes and their services [15] — a problem that shows up for example when related data structures export analogous operations under different names, ultimately resulting in decompositions.

It is all the more disturbing to discover that the same decomposition procedure applied to version 2.3 of Eiffel produces results which, taking into account scale factors, look remarkably similar to those for Eiffel 2.1, and this in spite of the considerable efforts made to standardize class interfaces in this more recent release of

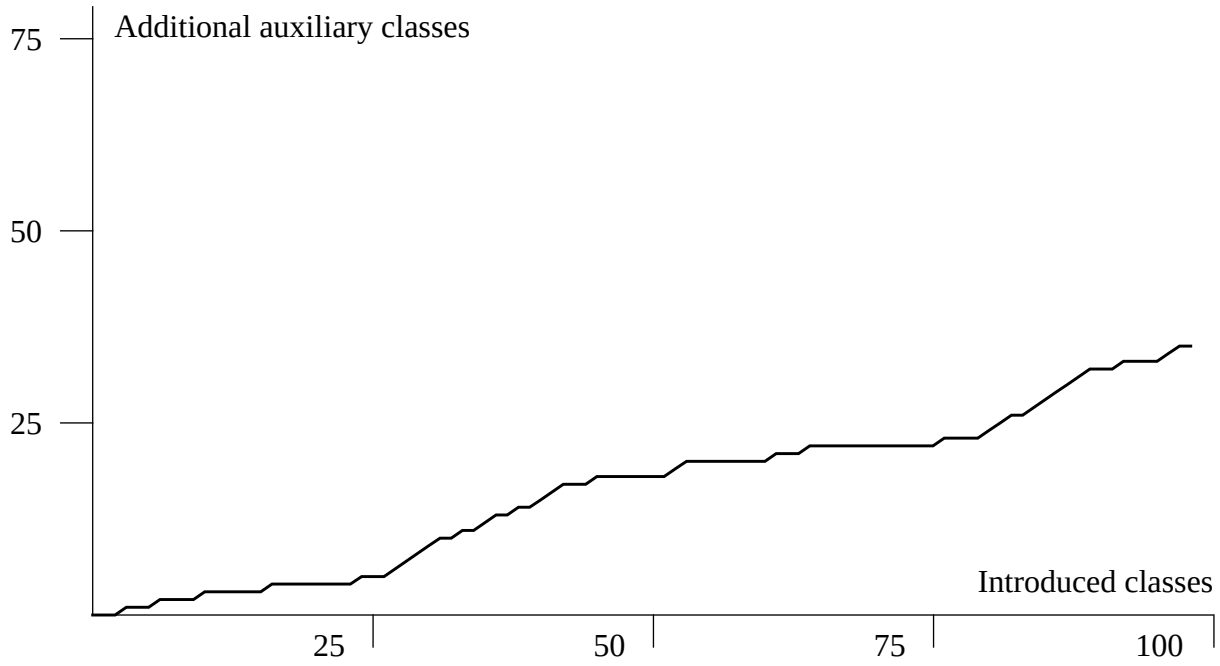


Figure 4 Decomposing class interfaces in the Eiffel 2.1 library. In this and all other graphs the original classes of the Eiffel library are introduced according to the topological sort described in 3.3.1.

the library. In the graph of figure 5, we distinguish the effects of the reorganization on the Eiffel library itself (lib), on the environment utilities (comprising eb — the Eiffel browser — and good — the graphical object-oriented debugger), and on user-contributed software (comprising motif — an encapsulation of Motif widgets, win — additional user interface classes, oracle — classes for database access, and net — classes for accessing network management functions).

Table 1 provides for a closer look at the structure of interfaces in the Eiffel library. The proportions of the various interface redefinition patterns are illustrated for each version of the standard Eiffel environment and for the additional libraries. The average number of direct ancestors per class, and the average number of exported methods per class are also given[1]. We observe that:

- The proportion of mixed relationships increases with the size of the interfaces. For classes with large interfaces, there is a greater probability of finding some identical methods in the list of exported features, as well as

1. For an alternative view on the structural characteristics of very large object-oriented libraries, the reader is referred to [8], which includes a comprehensive compilation of statistics describing a system written in an Eiffel-like language.

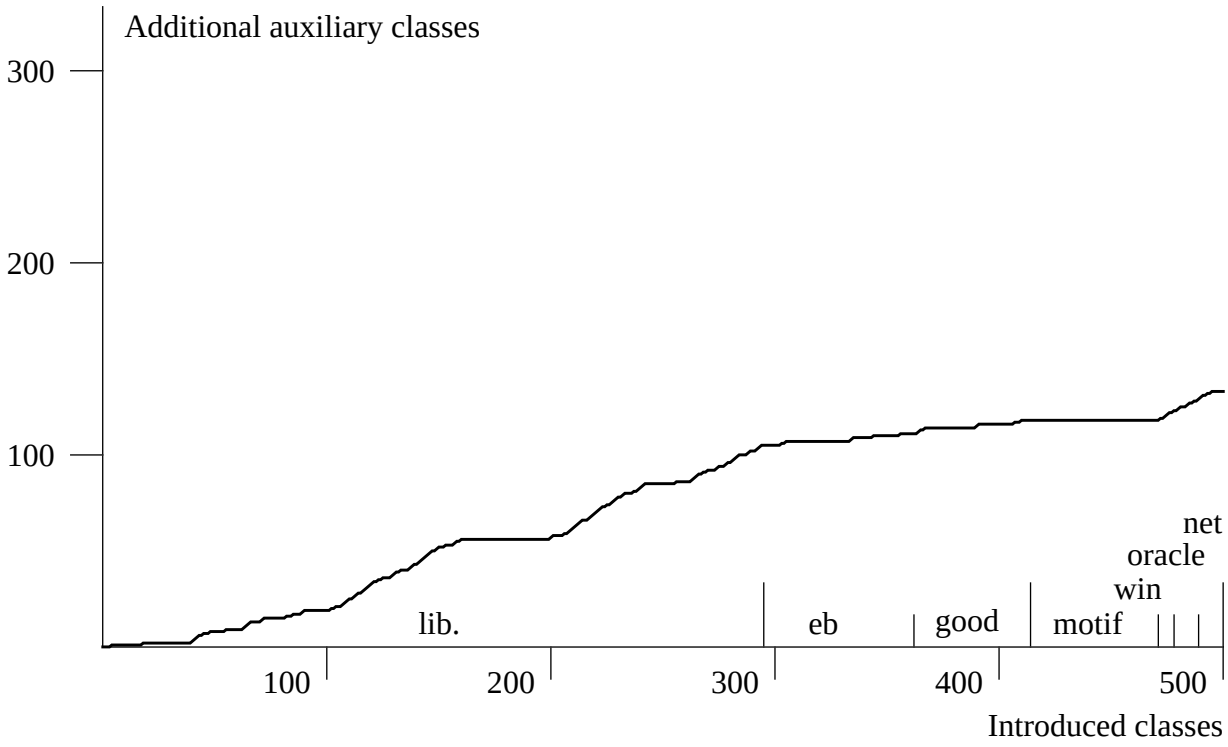


Figure 5 Interface decomposition for the Eiffel 2.3 library.

a greater probability of finding differences in the interfaces; hence the large proportion of mixed relationships. Conversely, classes with small interfaces are either very similar or completely different.

- There is a marked tendency for code sharing relationships to increase as the average number of ancestors per class increases. Overall, and with the exception of the motif sub-library, all parts of the hierarchy are structured with a fairly high proportion of subclassing patterns of the mixed and code-sharing kinds.

In conclusion, the advantages of standardizing interfaces seem to be in great part lost because of the complexity incurred by using inheritance as an implementation shortcut.

3.3.3 Factorising Object Definitions

In a second stage, we consider code sharing as the criterion to drive the reorganization. Of the six ways inherited methods can be redefined in Eiffel, three amount to their complete replacement, that is, inherited methods are overridden and can no longer be referred to in the subclass that redefines them. We choose to factorise

library	interface redefinition patterns scale subdivisions : 10% extension equality specialization mixed code sharing	average size of interface	average number of ancestors
win		31.29	1.0
lib 2.1		21.23	1.08
lib 2.3		18.25	1.33
good		13.27	1.33
net		12.0	1.45
oracle		5.55	1.64
eb		3.1	1.33
motif		12.09	1.0

Table 1 Proportions of the various interface redefinition patterns in the Eiffel libraries.

Eiffel classes exhibiting such redefinition patterns — except for deferred methods, which are supposed to be overridden anyway.

The process is carried out as described in section 3.2.2; as a complementary example to the factorisation of CIRCLE described in section 3.2.1, let us consider what happens to FIXED_STACK during the reorganization of the Eiffel 2.1 library. Class FIXED_STACK is a direct descendent from STACK and ARRAY; it redefines change_top, a method inherited from STACK that serves to modify the element placed at the top of the data structure. Because change_top is redefined without being renamed, this means that the original implementation of the method provided

by STACK has been effectively rejected by FIXED_STACK and is now unavailable in this class. This situation makes STACK and FIXED_STACK eligible for factorisation.

The algorithm creates an auxiliary class containing a deferred version of change_top as well as all other — unchanged — methods and variables defined by STACK. The inheritance link between FIXED_STACK and its ancestor STACK is redirected to connect FIXED_STACK to the auxiliary node. Moreover, an inheritance relation is established between STACK and the auxiliary class; except for the original implementation of change_top, STACK now inherits all its attributes from this node.

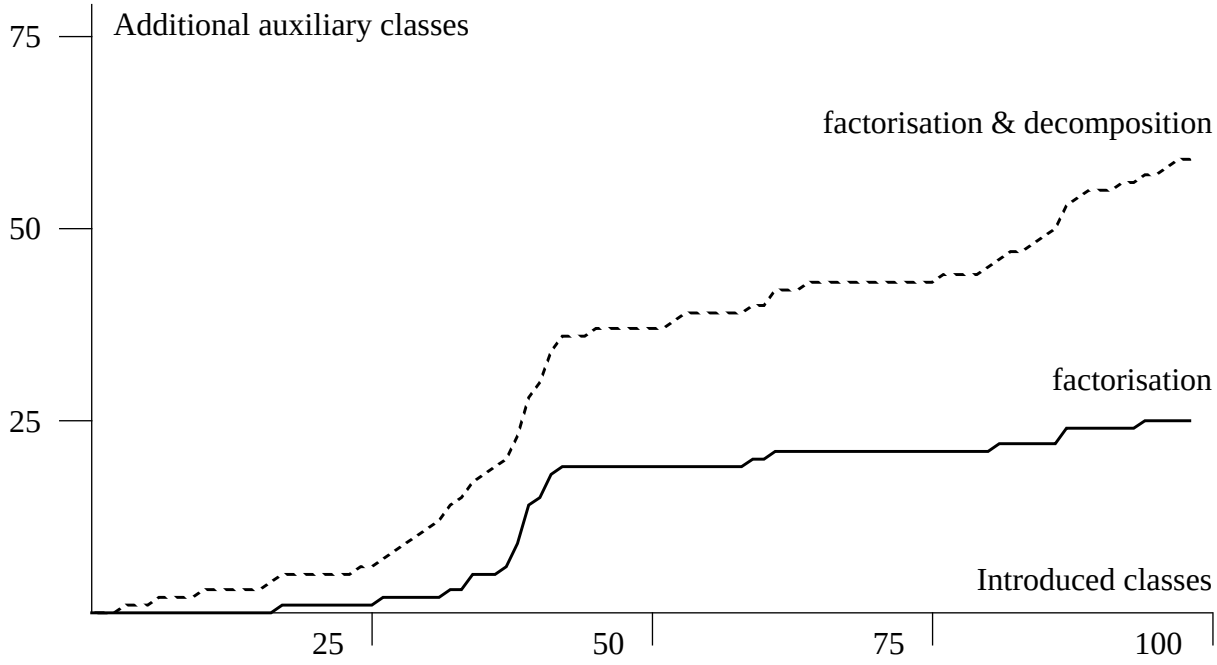


Figure 6 The reorganization of the Eiffel 2.1 library.

Figure 6 illustrates the factorisation of Eiffel 2.1. Although code factorisation seems well done in general, there is a zone in the graph were the introduction of classes causes important disturbances. The alterations carried out on the inheritance hierarchy pinpoint inadequate abstractions and, in particular, uncover a problem with classes deriving from lists (see table 2). TWO_WAY_LIST, for example, replaces properties inherited from LIST, an ancestor located two 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 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

Introduced class			Factorised ancestors	
rank	name	auxiliary	name	distance
17	boolean_set	1	indirect	1
18	string	—	indirect	1
19	integer_set	—	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	5	linked_list	1
			list	2
40	bi_linkable	1	linkable	1
41	two_way_tree	3	linked_list	2
42	gen_window	1	rect_shape	1
59	linked_queue	1	queue	1
61	fixed_stack	1	stack	1
62	linked_stack	—	stack	1
81	circle	1	ellipse	1
87	gtext	2	gen_figure	2
94	triangle	1	polygon	1

Table 2 *Factorisation in Eiffel 2.1: for each class causing a reorganization, we indicate when it is introduced, which of its ancestors are factorised, the number of inheritance links to traverse before reaching these ancestors, and how many auxiliary classes are added to the hierarchy.*

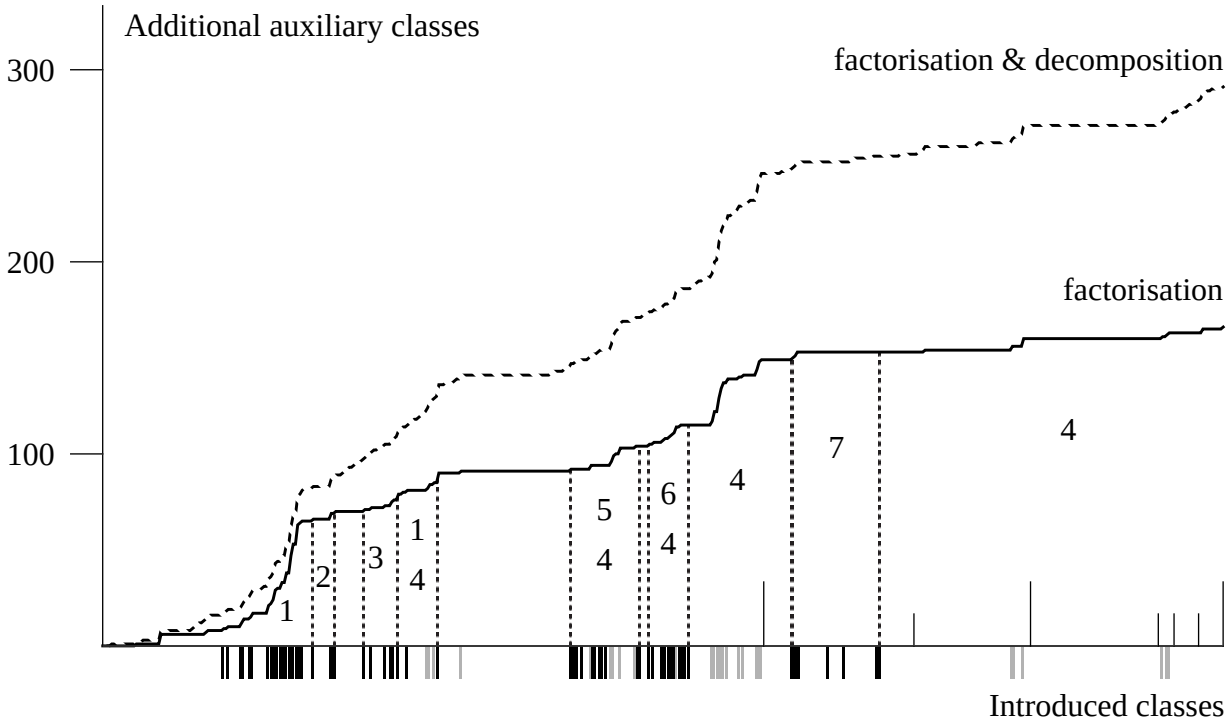
other kinds of relationships, while the last two situations seem to indicate that LIST and LINKED_LIST are insufficiently abstract. Conversely, several reorganizations, and their associated auxiliary classes, can be considered as "noise" — as is the case for the factorisations affecting INTBINTREE or GTEXT.

All these inheritance patterns explain why the curve of figure 6 exhibits such a steep step in its middle part: because several classes derive from LIST and LINKED_LIST in improper ways, there is a corresponding cascade of factorisations as descendents of these two classes are introduced in the hierarchy. The impact of such clustered factorisations actually tends to increase after each introduction of a class, since new descendents are usually inserted at progressively deeper locations in the inheritance graph, and because successive reorganizations render the connections in the hierarchy denser; the consequence is a local augmentation in the number of links and nodes to be traversed and restructured during factorisation.

A useful procedure for judging the overall quality of a library consists in applying a decomposition after a factorisation. For example, FIXED_STACK extends the interface of STACK but suppresses the export list defined by ARRAY, a situation corresponding to the fifth pattern of interface redefinition discussed in section 3.3.2. A decomposition serves to indicate that the relation between FIXED_STACK and its ancestors mixes subtyping with code sharing; it results in intercalating a node between FIXED_STACK on the one hand, and ARRAY and the auxiliary definition created by the factorisation of STACK on the other hand. After this operation, FIXED_STACK no longer inherits directly from its previous superclasses, but rather from a supplementary intermediate definition whose interface is similar to the one of STACK. As figure 6 shows, the main jump in the graph is quite recognizable when following this combined reorganization procedure.

The redesign of Eiffel 2.3 involved extracting commonalities from previously independent object descriptions and adding these intermediate abstractions to the hierarchy in the form of deferred classes. The presence of insufficiently abstract classes in Eiffel 2.1 caused problems with derived definitions — for example with QUEUE and its descendent LINKED_QUEUE, or with STACK and its subclasses FIXED_STACK and LINKED_STACK; the corresponding factorisations are no longer necessary in Eiffel 2.3. However, the redesign seems to have little overall effect on the library behaviour when it is reorganized: the steep jumps of the solid curve in figure 7 demonstrate that the way classes relate to each other in the hierarchy is still not satisfactory.

Figure 8 is a synthetic illustration of the consequences of factorisation in Eiffel 2.3. Each disc represents one class; the position of the disc in the diagram corre-



1. data structures
2. iterators
3. lexical & parsing
4. user interface, figures & windows (gray marks)
5. Gen_event
6. cursors
7. browser

Figure 7 Restructuring the Eiffel 2.3 library. The clusters of classes responsible for reorganizations correspond to those of figure 8.

sponds to the level at which the class appears in the inheritance graph — the root of the hierarchy is located at the top of the figure. The arcs in the diagram represent the important inheritance links among classes — direct relationships are denoted by full lines, indirect relationships (via classes not depicted in the diagram) by broken lines. The area of a disc is proportional to the number of times the class was reorganized because some methods in its description were overridden by a subclass. The black part of the disc represents those factorisations that led to the introduction of auxiliary classes in the hierarchy; the white part represents factorisations that did *not* lead to the addition of such intermediate definitions, and thus reveals recurring defects in the hierarchy that are eliminated in the same way. This graph confirms the clustered nature of reorganizations, which is already evident in figure 7, and shows that they are often caused by small groups of closely related classes.

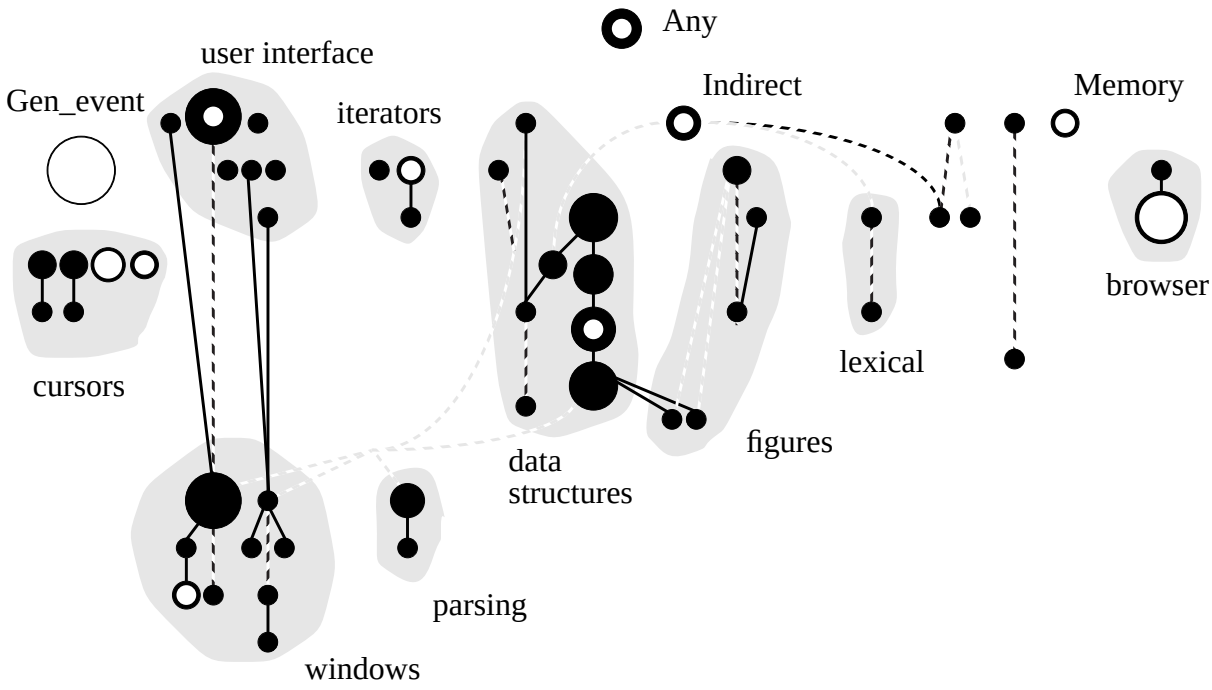


Figure 8 *The impact of factorisation on the Eiffel 2.3 libraries. Each disc represents one class, the links the major inheritance relationships between classes. The larger a disc, the more frequently the corresponding class has been factorised because of some dubious relationship with its descendents.*

- The group *user interface* includes basic classes for window systems, such as menu items and drawing surfaces. The larger disc represents ACTION_IMP, a class providing default actions for various user interface components. These defaults consist simply in printing out a message acknowledging receipt of an event; they are frequently overridden in the classes inheriting from ACTION_IMP for their implementation.
- GEN_EVENT is an interesting case where the reorganization algorithm did uncover a problem with the way this class and its descendents are defined, although factorisation is actually not the best solution for it. GEN_EVENT defines some variables and a method feature_count, which returns the number of variables that must be assigned a value for the interface Eiffel/X-Windows to function properly. GEN_EVENT defines four variables and therefore its method feature_count returns the value 4. Class BUTTON_EVENT, derived from GEN_EVENT, introduces seven new variables; however, instead of basing the definition of feature_count on the method acquired from its superclass, BUTTON_EVENT simply redefines

it to return the value 11. An identical subclassing pattern occurs in a number of other event classes; should a change occur in GEN_EVENT, then one would need to update the definition of feature_count in all its descendents. The factorisation suggests placing feature_count as a deferred method in a new ancestor; a better solution consists in changing the descendents of GEN_EVENT, so that, for example, BUTTON_EVENT would compute its feature count by relying on the general method in its parent with a call such as: return 7 + gen_event_feature_count.

- The classes modelling *cursors* in window systems exhibit what is sometimes called "inheritance for variation". Thus, CURS_DOWN inherits from CURS_UP, CURS_RIGHT from CURS_LEFT, and SHIFT_DOWN, SHIFT_RIGHT and SHIFT_LEFT from SHIFT_UP; each time, inherited methods are overridden. The algorithm finds the obvious missing abstractions — that could be named CURS_Y, CURS_X and SHIFT — and endows them with a deferred version of the overridden methods. Some supplementary adaptations are needed to take into account the redefinition of the behaviour of CURS_RIGHT, CURS_LEFT, CURS_UP and CURS_DOWN that occurs because of their descendents SCROLL_RIGHT, SCROLL_LEFT, SCROLL_UP and SCROLL_DOWN. The final class structure reflects precisely what we would expect from a more sensible design of the Eiffel library.
- The WINDOW class in Eiffel serves to define various kinds of windows (such as graphic windows or scrollable windows); it is also a superclass for other kinds of components such as command buttons, selection boxes and popup menus. The big black disc corresponding to WINDOW in the diagram shows that this class does not adequately capture the commonalities between different user interface components; some additional abstractions should therefore be added to the hierarchy.
- The central cluster deals with *data structures*, especially with lists. As with Eiffel 2.1, lists are used rather liberally, through inheritance, for defining and implementing other components. They serve as ancestors for classes as varied as SORTED_SET, FIXED_TREE and LINKED_STACK. The massive use of inheritance for implementation purposes is often decried in the object-oriented literature; we find here a quantitative justification for this criticism. Because the subclassing relationships between lists and their descendents are not semantically meaningful, the latter classes are forced to override a number of inherited properties. The extent of the factorisation shows that such inheritance patterns leads to overly

complex hierarchies and do not actually improve code sharing. Moreover, the group of classes defining lists do not even seem to be complete or abstract enough: thus, FIXED_LIST and LINKED_LIST override many operations acquired from the top abstraction LIST, while TWO_WAY_LIST replaces a significant part of the behaviour it inherits from LINKED_LIST.

- The situation involving circles and ellipses described in section 3.2.1 appears, in a slightly more complex form, in the *figures* group. The *browser* group includes the deferred class COMMAND, which defines two methods (redo and save) whose default implementations do not carry out any useful work. These methods are frequently overridden in the concrete descendents of COMMAND; the factorisation algorithm suggests replacing them with deferred methods in an auxiliary definition.

Other characteristic situations occur while restructuring Eiffel 2.3. For example, the defaults provided by classes responsible for low-level memory management functions are often overridden in the same way by their subclasses. This is the case for INDIRECT and its subclasses STRING, BOOL_STRING and INT_STRING, which replace the definition of generics — a private feature of INDIRECT that is in fact not to be used nor exported by its descendents; or for the garbage-collection class MEMORY and its method dispose — which is overridden in CURSOR and PIXEL. Another peculiarity is the use of repeated inheritance in the iterator classes C_TREE_ITER — that inherit three times from LINEAR_ITER — and TWC_ITER — twice a descendent from CHAIN_ITER. This pattern is accompanied by replacement of inherited behaviour, and it provokes some amount of class splitting and link reshuffling. The classes implementing lexical analysers and parsers also cause some restructuring. As for class ANY, the root of the Eiffel hierarchy, its method for testing the equality of objects is replaced with different implementations at several places in the library.

Generally, the class whose introduction triggers a reorganization and the ancestor from which it acquires and rejects properties are located one level apart in the hierarchy, but they can be as much as six levels distant from each other. We restrict reorganizations to situations involving only effective behaviour replacements; the median proportion of attributes defined by an ancestor (excluding those it inherits from other superclasses) and rejected by a subclass is thus 11% (the interquartile range goes from 5.6% to 20%). Some nodes in the inheritance graph introduce just a few methods and have as much as 75% of their definition overridden in subclasses. The entire factorised library is 26 levels deep and 126 classes wide (at its largest breadth), compared to a depth of 15 and a width of 110 for the original, unreorganized Eiffel library.

3.4 Evaluation

3.4.1 Measuring the Quality of the Eiffel Libraries

Figure 9 shows, for each kind of reorganization, how many auxiliary definitions are added to each library making up the Eiffel environment. The area of each disc is proportional to the size of each unreorganized library. Surprisingly enough, the redesign of Eiffel has not improved the quality of the standard library — according to our metrics, it has actually worsen. However, this simple analysis does not take into account structural factors affecting the results of the reorganizations: contrarily to Eiffel 2.3, the inheritance hierarchy of Eiffel 2.1 does not have a single root — it does not even form a connected graph. Since roots can never cause a reorganization, it is interesting to compute the increase of the restructured library in proportion to the number of original non-root classes. The adjustment demonstrates that the quality of the standard Eiffel library has indeed significantly increased, although it still lags behind those of other class collections.

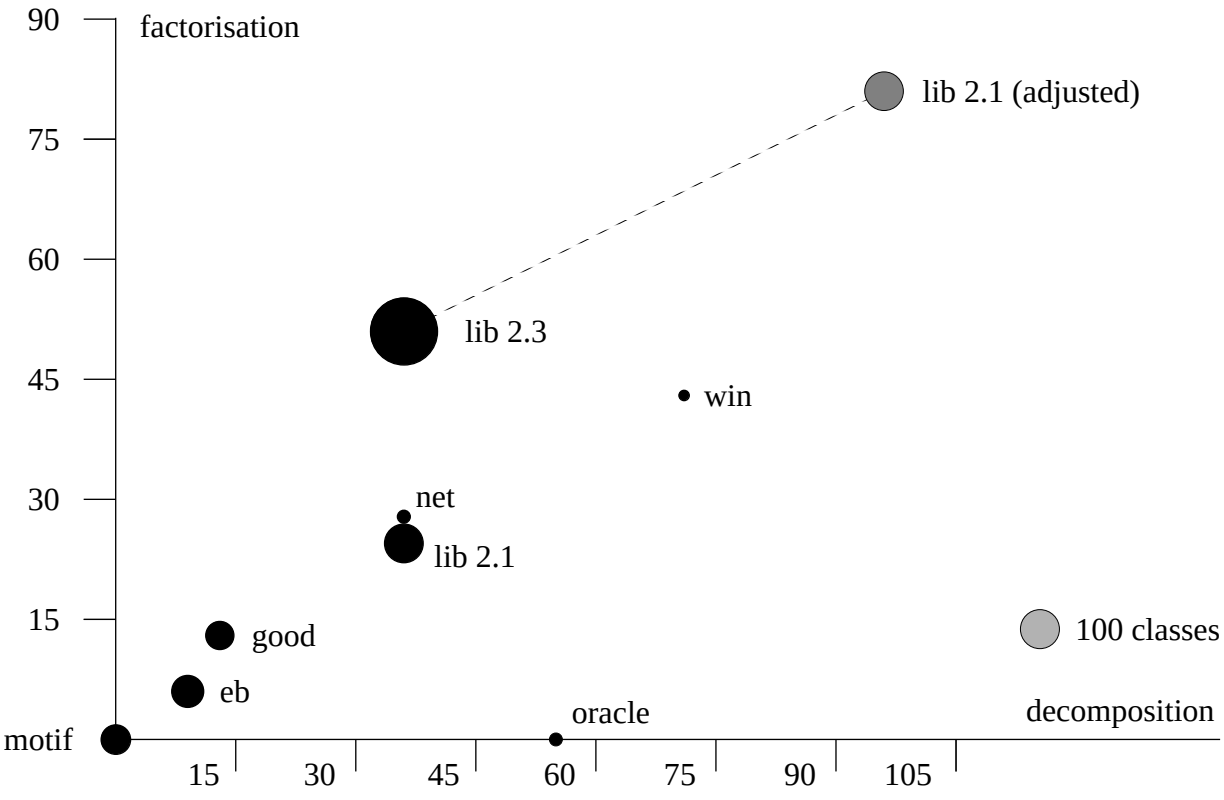


Figure 9 Increase (in %) in the number of classes of the Eiffel libraries because of reorganizations.

As in the graphs of the preceding sections, the motif library appears to be clearly an outlier: it is never altered by the reorganization algorithms. In fact, this library just serves to encapsulate Motif functionality; its inheritance graph is practically flat and most of the classes define only one operation for creating and initializing widgets. As such, it cannot be considered as a typical example of object-oriented design. An analogous remark applies to the oracle library, which encapsulates access functions to a relational database management system.

3.4.2 Relevance

Reorganization algorithms perform strictly structural transformations on object descriptions; their results provide for a coarse view of how the entire inheritance graph should be updated before fine-grained modifications are applied to selected aspects of class descriptions. We must determine whether factorisations and decompositions really provide a good starting point for redesigning an object-oriented library, or if the proportion of spurious transformations renders restructuring algorithms useless as a software engineering tool.

Decompositions serve to pinpoint inadequate subclassing structures and naming conventions. These problems are usually solved by substituting inheritance links with component relationships, and by selecting alternative method identifiers. The auxiliary definitions constructed by the decomposition algorithm are thus of little value in themselves; on the other hand, the characterization of inheritance relationships it affords is valuable for analysing a library.

The main goal of factorisation is to find missing abstractions in a hierarchy; this kind of transformation is therefore of great practical importance, especially since it matches closely the reorganization patterns observed by software developers when manually restructuring inheritance hierarchies [1][11]. Table 3 assesses the relevance of factorisation for the various Eiffel 2.3 libraries. It indicates whether reorganizations were caused by derivations from insufficiently abstract ancestors (containing non-deferred methods), to the lack of some abstractions in the inheritance graph, to resorting to inheritance for implementation purposes, or to other kinds of unusual subclassing relationships. The reorganizations are labelled as significant when they correspond to a good redesign sketch for improper class structures (as with cursor classes), as interesting when they uncover severe problems in the hierarchy that are better solved with mechanisms other than inheritance (as in the case of GEN_EVENT), and as inconclusive when the redefinitions causing the reorganization are insignificant (as it occurs when one unique method of a large class is overridden by a single descendent) or when they could not be properly

evaluated. The numbers correspond to the percentage of factorisations (in other words, the fraction of the total area made up by all discs in figure 8) that can be assigned to each category.

Cause	inconclusive	interesting	significant	Total
concrete ancestors	7		2.5	9.5
missing abstractions	1	8	18.5	27.5
Total classes				37
code sharing	17	10		27
other subclassing	21	15		36
Total inheritance				63
Total	46	33	21	100

Table 3 Evaluating the relevance of factorisation for the Eiffel 2.3 libraries.

The most salient feature of these statistics is the high proportion of reorganizations due to inadequate subclassing patterns (63%); this sets an obvious upper bound on the effectiveness of algorithms, such as ours, that aim at improving inheritance structures. Nevertheless, factorisation does detect actual defects in the modelling of the hierarchy in a majority of cases (interesting and significant cases: 54%); a few remaining inconclusive situations can be elucidated by a combination of decomposition and factorisation. If we examine the defects caused by missing abstractions or concrete ancestors, we observe that the algorithm produces useful results in more than 78% of the cases (21% + 8% over 37%). Overall, our incremental restructuring approach is reasonable, if we consider that the algorithms used in the tests are simplified and that they operate automatically, without any additional information to fine-tune the transformations.

The question whether our factorisation algorithm uncovers all situations where it can be applied successfully is easily settled [5]. Because the algorithm works incrementally and manipulates only local information, it cannot extract missing abstractions when the definitions to be compared are located in disjoint subclassing paths. To achieve this goal, one must resort to global restructuring algorithms that operate on the entire inheritance graph[1] [12]. However, it should be noted that glo-

1. The conditions under which an incremental factorisation produces the same (optimal) results as a global reorganization are discussed in [6].

bal revisions may thoroughly transform the hierarchy; the results are therefore much more difficult to grasp and to utilize, particularly with libraries comprising hundreds of classes. Incremental factorisation, on the other hand, limits its scope to the inheritance paths leading to one new class — an approach that also guarantees better performance in an interactive environment. Finally, it is doubtful that a global reorganization can achieve significantly better results without additional processing to extract the structural similarities between class interfaces or method signatures that are hidden because of diverging naming and programming conventions [16]. A case in point are the two window classes present in the Eiffel library — one dealing with character-oriented terminals, the other interfacing with X-Windows — that do not derive from a common window abstraction [15].

3.4.3 Future Issues

Our study of the automatic reorganization of the Eiffel library mostly confirms design guidelines advocated by developers of object-oriented libraries [11][15]: classes with small interfaces are easier to understand and to use — more complex functionality is better distributed among several components; method signatures must be standardized according to consistent naming conventions; subclassing must represent meaningful concepts and not be used for code sharing — an utilization of inheritance that paradoxically undermines the ultimate goal of reusing implementations as well as abstractions. A less expected observation is the extent to which the mechanism of inheritance is abused in the Eiffel library. Actually, this trait is widespread and appears, to different degrees, in other large object-oriented libraries. Thus, an analysis has shown that 5 additional abstractions are necessary to restructure the Smalltalk collection hierarchy (originally comprising 12 components) so as to enforce behavioural compatibility between classes — a 42% increase in the size of the library [7]. Carrying out the same analysis on the basis of purely syntactical elements produces an almost identical, albeit imperfect hierarchy containing 6 supplementary definitions — i.e. 50% more classes. Both reorganizations of the Smalltalk collection hierarchy amount essentially to factorisations, and they introduce auxiliary definitions in a proportion comparable to the one observed for the Eiffel 2.3 standard library (figure 9). However, because Smalltalk only provides simple inheritance, avoiding code sharing relationships altogether proves to be much more difficult than with languages where classes can be derived from several ancestors.

As another practical consequence, multiple inheritance seems somewhat overrated as a programming construct — being overwhelmingly relied upon for implementation purposes rather than for representing useful concepts. It should

therefore be used sparingly, and every utilization of this mechanism should warrant a serious attempt to find an alternative design. An important conclusion is the fact that, as suggested in one previous research on class evolution [1], further reorganization is unavoidable even after repeatedly recasting a hierarchy.

Factorising abstractions out of concrete, specialized descriptions and discovering intermediate modelling concepts are essential activities during object-oriented programming [15]; we therefore expect reorganization algorithms to fulfil a significant role as an interactive tool for object-oriented design and maintenance. The statistics presented in table 3 show that this approach, focusing on inheritance relationships, must nevertheless be supplemented with methods for analysing and standardising naming and signature patterns among classes, and for transforming inadequate inheritance links into component relationships [16]. The question of performance after reorganizations remains to be settled. Optimizing method invocations is customary when dealing with routines defined along inheritance paths (such as it occurs when sending messages to *self* or to *super*); in this respect, the alterations carried out by our factorisation algorithms should not significantly impact the efficiency of a library. On the other hand, replacing inheritance links with component relationships introduce additional indirections that can be difficult to get rid of — for example with code inlining techniques. Finally, there are important issues related to the preservation of class behaviour across reorganizations which, so far, have been addressed only partially [18].

Building an effective object-oriented redesign environment requires integrating the various approaches to class evolution discussed in the introduction — incremental reorganization, refactoring, class modification primitives and versioning — with browsing and debugging utilities. Programmers will then be able to analyse the components they add to a library, incrementally reorganize class hierarchies on the basis of various criteria, and detect the places most likely to require further revisions. After inspecting the outcome of reorganizations, programmers may adjust them with modification primitives, and perhaps embark on more comprehensive refactoring activities. The results of different reorganizations and their subsequent adjustments can be kept as versions of the hierarchy, which can be further modified, tested, debugged and possibly cancelled by the programmers. When a satisfactory design for a 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 temporary versions of the hierarchy are discarded.

Ultimately, programming environments should be configurable with design and reorganization patterns, so as to be able to take into account some knowledge

concerning the application domain, the concepts embodied in the class collection, and the principles that guide the construction of the library. A formal and comprehensive model for describing and restructuring hierarchies, similar to the normal forms available in the relational data model, is also lacking. New approaches to object-oriented design are needed to enable the systematic design and production of reusable class libraries and frameworks.

Acknowledgments

A version of this paper is submitted for publication.

Part of the work was supported by the German Ministry of Research and Technology (BMFT) in the project STONE under number 01 IS 104 A/7.

References

[1] Bruce Anderson and Sanjiv Gossain, “Hierarchy Evolution and the Software Lifecycle,” in *Proc. 2nd TOOLS Conference*, ed. Jean Bézivin, Bertrand Meyer and Jean-Marc Nerson, pp. 41–50, Paris, 1990.

[2] Jay Banerjee, Won Kim, Hyoung-Joo Kim and Henry F. Korth, “Semantics and Implementation of Schema Evolution in Object-Oriented Databases,” SIGMOD Record (special issue on SIGMOD ’87), vol. 16, no. 3, pp. 311–322, ACM, December 1987.

[3] Gilles Barbedette, “Schema Modification in the $LISPO_2$ Persistent Object-Oriented Language,” in *Proc. 5th ECOOP Conference*, ed. Pierre America, Lecture Notes in Computer Science no. 512, pp. 77-96, Springer Verlag, Geneva, 15-19 July 1991.

[4] Anders Björnerstedt and Christer Hultén, “Version Control in an Object-Oriented Architecture,” in *Object-Oriented Concepts, Databases, and Applications*, ed. Won Kim and Frederic H. Lochovsky, Frontier Series, pp. 451–485, Addison-Wesley/ACM Press, 1989.

[5] Eduardo Casais, “An Incremental Class Reorganization Approach,” in *Proc. 6th ECOOP Conference*, ed. Ole Lehrmann Madsen, Lecture Notes in Computer Science no. 615, pp. 114-132, Springer Verlag, Utrecht 29 June-3 July 1992.

[6] Eduardo Casais, *Managing Evolution in Object-Oriented Environments: An Algorithmic Approach*, PhD Thesis, Université de Genève, Geneva, 1991.

[7] William R. Cook, “Interfaces and Specifications for the Smalltalk-80 Collection Classes,” SIGPLAN Notices (special issue on OOPSLA ‘92), vol. 27, nr. 10, pp. 1-15, ACM, October 1992.

[8] Bernard Coulange and Alain Roan, “Object-Oriented Techniques at Work: Facts and Statistics,” in *Proceedings 10th TOOLS Conference, Versailles*, eds. Boris Magnusson, Bertrand Meyer and Jean-François Perrot, Prentice-Hall, 1993, pp. 89-94.

[9] Adele Goldberg and Daniel Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

[10] Daniel C. Halbert and Patrick D. O’Brien, “Using Types and Inheritance in Object-Oriented Programming,” IEEE Software, pp. 71–79, IEEE, September 1987.

[11] Ralph E. Johnson and Brian Foote, “Designing Reusable Classes,” Journal of Object-Oriented Programming, pp. 22–35, June-July 1988.

[12] Karl J. Lieberherr, Paul Bergstein and Ignacio Silva-Lepe, “From Objects to Classes: Algorithms for Optimal Object-Oriented Design,” BCS/IEE Software Engineering Journal, pp. 205-228, July 1991.

[13] K. Lieberherr, I. Holland and A. Riel, “Object-Oriented Programming: an Objective Sense of Style,” SIGPLAN Notices (special issue on OOPSLA ’88), vol. 23, no. 11, pp. 323–334, ACM, November 1988.

[14] Bertrand Meyer, *Object-Oriented Software Construction*, Series in Computer Science, Prentice-Hall International, 1988.

[15] Bertrand Meyer, “Tools for the New Culture: Lessons from the Design of the Eiffel Libraries,” Communications of the ACM, vol. 33, no. 9, pp. 68–88, September 1990.

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

[17] D. Jason Penney and Jacob Stein, “Class Modification in the GemStone Object-Oriented DBMS,” SIGPLAN Notices (special issue on OOPSLA ’87), vol. 22, no. 12, pp. 111–117, ACM, December 1987.

[18] Emmanuel Waller, “Schema Updates and Consistency,” in *DOOD’91 Proceedings*, eds. Claude Delobel, Michael Kifer and Yoshifumi Yasunaga, Springer Verlag, Lecture Notes in Computer Science no. 566, December 1991, pp. 167–188.

[19] Roberto Zicari, “A Framework for Schema Updates in an Object-Oriented Database System,” in *Building an Object-Oriented Database System — The Story of O_2*, eds. François Bancilhon, Claude Delobel and Paris Kanellakis, Morgan Kaufmann Publishers, 1992, pp. 146–182.