Managing Class Evolution in Object-Oriented Systems

Casais, Eduardo

Managing Class Evolution in Object-Oriented Systems

Eduardo Casais

Abstract

Software components developed with an object-oriented language undergo considerable reprogramming before they become reusable in a wide range of applications or domains. Tools and methodologies are therefore needed to cope with the complexity of designing, updating and reorganizing vast collections of classes. This paper describes several techniques for controlling change in object-oriented systems, illustrates their functionality with selected examples and discusses their advantages and their limitations. As a complement to traditional approaches like version management, we propose new algorithms for automatically restructuring a hierarchy when classes are added to it. These algorithms not only help in handling modifications to libraries of software components, but they also provide useful guidance for detecting and correcting improper class modelling.

1. Introduction

1.1 The problem

Object-oriented languages are currently considered a promising approach for coping with the problems plaguing software development. Mechanisms like classification and the uniformity of the object model support the large-scale reuse and recombination of pre-defined software components—thus boosting programmer productivity. Subclassing allows additional definitions to be easily accommodated in a class hierarchy, through marginal extensions to inherited properties. With genericity and delayed binding, some characteristics of a class can be left unspecified, to be determined at a later time. 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 [12][16][21].

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. This is typically achieved by redefining variables, reimplementing methods, rearranging inheritance links, etc. Such modifications indicate that the collection of software components is not entirely satisfactory, since it cannot be reused in its current form. As an example, the library provided with Eiffel had to be partly redesigned when the second major version of the language was released, mainly to standardize class interfaces [38]. 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.

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

CONTENTS

- User needs are rarely stable: additional constraints and functionality have to be constantly integrated into existing applications, resulting in considerable program restructuring.
- Software components are difficult to classify in pre-defined taxonomies. Specializing a category whenever a particular entity does not quite fit into it leads to the famous "one instance class" problem [34].

- Experience shows that stable, reusable classes are not designed from scratch, but are "discovered" through an iterative process of testing and improvement [25]. In particular, 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 [23].
- In principle, a hierarchy constitutes a standard kernel of functionality that can only be extended with additional subclasses. This may hamper the design and development of innovative software components if the modelling of objects in the hierarchy does not follow the advances in its corresponding application domain.
- Reusing software raises complex integration issues when teams of programmers share classes that do not originate from a common, compatible hierarchy. Classes may require significant adaptations—like reassigning inheritance dependencies or renaming properties—to be exchanged between different environments.

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.

1.2 The solutions

Among the approaches that have been proposed to control evolution in object-oriented systems, we identify the following general categories:

- *Tailoring* consists in adapting slightly class definitions when they do not lead to easy subclassing. Most object-oriented languages provide built-in constructs for making limited adjustments to class hierarchies.
- *Surgery.* Every possible change to an object definition can be decomposed into specific, primitive update operations. Maintaining the consistency of a class hierarchy requires that the consequences of applying these primitives be precisely determined.
- *Versioning* enables teams of programmers to record the history of class modifications during the design process, to control the creation and dissemination of software components, and to try different paths in a coordinated way when modelling complex application domains.
- *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 and on the operations performed by programmers on these structures to discover imperfections in the hierarchy and to suggest alternative designs.

A second problem, related to class evolution, is that instances must be updated after their representation is modified. We consider in detail three techniques to tackle this crucial issue:

- *Change avoidance* consists in preventing any impact from class modifications on existing instances, for example by restricting the kind of changes that are brought to object definitions.
- *Conversion* physically transforms objects affected by a class change so that they conform to the new definition.
- *Filtering* hides the differences between objects belonging to several variants of the same class. This result is achieved by encapsulating instances with an additional software layer that extends their normal properties.

This article is divided in three parts. The first part explains the principles behind class tailoring, class surgery and class versioning, referring when appropriate to the research prototypes or industrial products that implement these techniques. For each approach, we illustrate its functionality with simple examples, and we give an appreciation of its advantages and of its shortcomings. The second part is devoted to class reorganization. It discusses informal guidelines for restructuring inheritance hierarchies and explains how these guidelines can be automated. Some recent results of the research performed by the author for developing class reorganization algorithms are highlighted; this paper should therefore not be considered strictly as a survey. Our automatic reorganization algorithms are described through several examples, and our contribution is evaluated in the context of the general class design problem. The third part considers the problem of propagating class changes to instances, and analyses how change avoidance, conversion and filtering relate to class modification methods. The conclusion compares the potential of the various methodologies and argues for an integrated approach to class evolution.

MODIFYING CLASS DEFINITIONS

The strategies adopted for modifying class hierarchies generally do not allow the programmer to perform free, uncontrolled changes to a collection of classes. The available methods make quite different assumptions on the scope and on the aims of the update operations that can be carried out on object descriptions. They are used at various points during application development, according to their perspective of class evolution.

- With tailoring, existing class definitions are not directly modified; adaptations are applied to inherited properties when deriving new subclasses. Keeping a stable hierarchy is a fundamental goal with this approach.
- Surgery considers all the implications of direct modifications to a class, and studies what additional adjustments are required to leave the whole hierarchy in a valid state. No matter how the classes evolve, a number of essential integrity constraints have to be satisfied.
- Versioning records 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.

2. Class Tailoring

2.1 Issues

Object-oriented programming draws a large part of its power and flexibility from the concept of inheritance. A class groups a number of attributes that are used either to store information (variables), or to implement the operations that an object supports (methods). New class structures are easily derived from previous ones through subclassing: one just needs to specify the parents from which the new definition acquires its essential properties, and then to augment this kernel of functionality with some additional, more specialized behaviour. In principle, a subclass has access to all attributes inherited from its ancestors, and so can build on the basic services they provide [42][48]. Inheritance has revealed itself to be an extremely powerful mechanism for incremental programming; its use has typically resulted in the development of medium size to large hierarchies of software components comprising, among others, fundamental data structures (trees, stacks, queues, etc.), user interface tools (windows, menus), and graphical elements (lines, polygons) [12][22][49].

Quite often, programming does not follow the ideal scenario where superclasses, extended with additional attributes, naturally give rise to new object descriptions. Inherited variables and methods do not necessarily satisfy all the constraints to be enforced in specialized subclasses. Typically, one prefers an optimized implementation of a method to the general, and perhaps inefficient algorithm defined in a superclass. Similarly, a variable with a restricted range may be more appropriate than one admitting any value. Moreover, subclassing serves several purposes simultaneously, like code sharing, type validation, or modelling generalization/specialization relationships; these various goals are difficult to reconcile in a unique structure [23]. Tailoring mechanisms alleviate these problems by allowing the programmer to replace unwanted characteristics from standard classes with properties better suited to new applications.

2.2 Language mechanisms

Object-oriented languages have always provided simple constructs for tailoring classes. These mechanisms have proved useful to overcome the difficulties caused by strict inheritance, which forces the programmer either to reuse all attributes from a superclass or to split a hierarchy in numerous partial class definitions—a solution feasible only if multiple inheritance is available. The Eiffel language constitutes a good example of the possibilities afforded by such mechanisms [37]; we present here an overview of its tailoring capabilities—which for the most part exist, perhaps in a different form, in many other programming languages.

- *Renaming* is the simplest way to effectively modify a class definition. Renamed variables and methods can no longer be referred to by their previous identifier, but they keep all their remaining properties, like their type or their argument list.
- *Redefinition* enables the programmer to actually alter the implementation of attributes. The body of a method may be replaced with a different implementation, which is then executed in the place of the code inherited from the superclass when the corresponding operation is invoked on an instance of the subclass. Eiffel also allows the type of inherited variables, parameters and function results to be redeclared, provided the new type is

compatible with the old one. Finally, the pre and post-conditions of a method may be redefined, as long as the new pre-condition (respectively the new post-condition) is weaker (respectively stronger) than the original one.

- *Interfaces* are not statically defined in Eiffel. A subclass may choose to change its list of visible attributes entirely, even when these are inherited. An attribute declared as private in an ancestor may be made accessible; conversely, a previously visible attribute may be excluded from the subclass interface.

The following excerpt from the Eiffel library illustrates the use of these various tailoring mechanisms. Notice the changes in class interfaces, the redefinition of the variable parent, and the renaming and overriding of the operations for creating tree objects.

```
--
--    Trees where each node has a fixed number of children (The number of children is
--    arbitrary but cannot be changed once the node has been created).
--
class FIXED_TREE [T]
    export
        start, finish, is_leaf, arity, child, value, change_value, node_value,
        change_node_value, first_child, last_child, position, parent, first, last,
        right_sibling, left_sibling, duplicate, is_root, islast, isfirst, go,
        delete_child, change_child, attach_to_parent, change_right, change_left,
        go_to_child, wipe_out
    inherit
        ...
    feature
        parent    :    FIXED_TREE [T];
        Create (n : INTEGER; v : T) is
            --    Create node with node_value v and n void children.
            ...
        end; -- Create
        ...
    end -- class FIXED_TREE

--
--    Binary trees.
--
class BINARY_TREE [T]
    export
        start, finish, is_leaf, arity, child, value, change_value, node_value,
        change_node_value, left, right, has_left, has_right, has_both, has_none,
        change_left_child, change_right_child
    inherit
        FIXED_TREE [T]
        rename
            Create as fixed_Create, first_child as left, last_child as right
        redefine
            parent
    feature
        parent    :    like Current;
        Create (v : T) is
            --    Create tree with single node of node value v
        do
            fixed_Create (2, v)
        ensure
```

```
            node_value = v;
            right.Void and left.Void
        end; -- Create
        --
        --  Methods has_left, has_right, has_both, has_none, change_left_child,
        --  and change_right_child are defined here.
        --
    end -- class BINARY_TREE [T]
```

Other techniques integrated with a programming language have been proposed for configuring objects. In HyperCard, individual cards can extend or modify the definition inherited from their common background; this amounts to a per-instance tailoring of objects. With the object-oriented variants of LISP, the programmer chooses how to combine inherited methods in a new class, and thus controls the way the behaviour of subclasses is determined. In particular, it is possible to extend (with :before or :after methods), shadow (with primary methods), or refer selectively to (with :around methods) the operations of a superclass [27][39].

These techniques assume that all adaptations to existing classes are carried out through subclassing, by overriding some inherited properties. Sometimes however, adaptations cannot be limited to local class adjustments: global changes to the hierarchy are required. Objective-C provides a mechanism where a user-defined class can "pose" as any other class in the hierarchy. When the "posing" class is installed in the system, it shadows the original definition. Objects depending on the "posed" class, whether by inheritance or by instantiation, do not have to be changed: the method dispatching scheme guarantees that a message sent to an object of the posed class actually result in invoking a procedure in a posing object. The posing class may override any method of the posed class, and define additional operations; it has access to all original, now shadowed, properties. The posing mechanism serves to change a class definition when its source code is unavailable.

2.3 Excuses

An approach analogous to the attribute redefinition techniques is described in [8]. The author proposes a formal mechanism for "excusing" abnormal cases that arise when modelling an application domain and that do not fit within the existing hierarchy. For example, a system keeping information on students may have to cope with the case of people who did part of their studies in foreign countries with different grading schemes and academic titles. Another problem occurs when attributes with contradicting properties are inherited through different paths. The traditional example, originally studied as a test for default reasoning in artificial intelligence, is based on the situation where a person is at the same time a Quaker (and therefore a pacifist) and a Republican (and therefore opposes pacifism). IS-A relations often exhibit such inconsistencies that can rarely be avoided when a hierarchy is initially constructed.

Possible solutions to overcome these problems include the disciplined use of strict inheritance (which leads either to the creation of intermediate classes for purely technical reasons, or to a decrease in the degree of sharing among classes, if they have to specify individually their version of the conflicting attributes), the dissociation of classes and types (which defeats polymorphism), or resorting to default reasoning techniques (whose properties are not always well-determined). The excuse mechanism is suggested as an alternative, flexible way for managing non-strict inheritance.

With the excusing approach, contradictions between the definition of a class and its ancestor must be explicitly acknowledged. Thus, while

```
class STUDENT
    variables
        Name     :   STRING;
        Address  :   STRING;
        Degree   :   (Bachelor, Masters, Doctor-of-Philosophy);
    end;
```

denotes the standard class of students,

```
class FOREIGN-STUDENT
    inherits STUDENT;
    variables
        Degree   :   (Licence, Diplômo, Doctorat)
                         excuses Degree on STUDENT;
    end;
```

represents a subclass where the Degree attribute of STUDENT is redeclared. Since the new type of Degree is not a refinement of the previous one, an explicit excuse has to be provided to inform the system that this situation represents an exception and not a programming error. This excuse is inherited by all subclasses of FOREIGN-STUDENT; in the case where the Degree attribute is once again superseded, it may be necessary to excuse its new type with respect to the one of FOREIGN-STUDENT—and with the one of STUDENT as well—if they are incompatible.

In his paper, the author sketches a system for integrating types with excuses. The development of such a scheme would facilitate the type checking of expressions, the detection of unsafe operations on objects (that refer to values or attributes that have been redeclared in incompatible ways), and the correct handling of database queries (without overlooking exceptional entities). Formally, consider

```
class X
    variables
        a   :   T;
    end;

class Y
    inherits X;
    variables
        a   :   S excuses a on X;
    end;
```

where Y is a subclass of X, and T and S denote the type of attribute a. Then, for every instance $x \in X$, the following condition holds: $x.a \in T \vee (x \in Y \wedge x.a \in S)$. This basic property of excuses is used for reasoning about expressions involving entities from exceptional classes, as happens when trying to retrieve all students with a Masters degree; the system should avoid processing instances from FOREIGN-STUDENT, because their Degree attribute does not conform to the information normally associated with students and could even have a different physical representation.

2.4 Evaluation

Tailoring techniques are useful for performing small adjustments to a class collection. The overriding of inherited attributes enables the programmer to escape from a rigid inheritance structure that is not always well-suited to application modelling. It facilitates the handling of exceptions locally, and does not require the factoring of common properties into numerous intermediate classes. Tailoring mechanisms correspond to constructs of object-oriented languages; consequently, they can be considered as a standard programming technique and they can be implemented efficiently within compilers.

On the other hand, over-reliance on tailoring and excuses may quickly lead to an incomprehensible specialization structure, overloaded with special cases, and, as far as persistent object-oriented systems are concerned, difficult to manage efficiently with current database technology. Introducing exceptions in a hierarchy destroys its specialization structure and obscures the dependencies between classes, since a property cannot be assumed to hold in every object derived from a particular definition. Renaming and interface redeclaration may completely break down the standard type relations between classes and subclasses. When signature compatibility is not respected, polymorphism becomes impossible: an instance of a class may no longer be used where an instance of a superclass is allowed. Nothing prevents programmers to radically alter the semantics of a method by changing its implementation, say by replacing the code for computing the area of a polygon with one that returns its perimeter. Changing attribute representations also cancels many of the benefits of code sharing provided by inheritance.

If tailoring is allowed, one must be wary of developing a collection of disorganized classes. Exceptions should not only be accommodated, but also integrated into the type hierarchy when they become too numerous to be considered as special cases. Unfortunately, the techniques we have described in this section do not really help in detecting design flaws in object descriptions.

3. Class Surgery

3.1 Issues

Whenever changes are brought to the modelling of an application domain, corresponding modifications must be applied to the classes representing real-world concepts. This kind of operation disturbs an class hierarchy much more profoundly than the tailoring techniques we have described in the previous pages: instead of overriding some inherited properties when new subclasses are defined, it is the structure of existing classes themselves which is to be revised. Because of the multiple connections between class descriptions, care has to be taken so that the consistency of the hierarchy is guaranteed.

This problem also arises in the area of object-oriented databases, where it has been extensively investigated [4][29][43][50]. There, the available methods determine the consequences of class changes—on other definitions and on existing instances as well—so that possible integrity violations be avoided. These methods can be broken down into a number of 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 to the system 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*, or possibly replaced by another attribute with the same identifier inherited through another subclassing path.
4. Finally, the effects of schema changes are reflected on the persistent store. If necessary, instances belonging to modified classes are converted to conform to their new description. For example, additional storage space must be allocated for objects whose definition has been augmented with additional attributes.

The next sections examine in detail these various aspects of schema evolution management. We base our discussion mainly on the research performed around the following object-oriented database systems:

- GemStone, a commercial system developed at Servio Logic, that extends the basic Smalltalk object model with persistency and database functionality. The OPAL language is used for defining classes and manipulating objects.
- ORION, a research prototype developed at MCC. ORION is intended to serve as a general-purpose persistent object-oriented environment integrated with Common LISP.
- O_2, a prototype object-oriented database system developed by the GIP Altaïr for supporting application development in a wide range of domains and programming languages (Basic, C, LISP).
- OTGen, a research system developed at Carnegie Mellon University for investigating issues dealing with object-oriented database management.

Since instance conversion techniques are an important issue common to other approaches, most notably versioning, we defer their description to the part devoted to change propagation on page 182.

3.2 Class invariants

Every class collection contains a number of integrity constraints that must be maintained across schema changes. These constraints, generally called class invariants in the literature, impose a certain structure on class definitions and on the inheritance graph; depending on the object model, they may allow more or less restricted types of class hierarchies.

	GemStone	ORION	O_2	OTGen
Representation	✓			
Inheritance graph	✓	✓	✓	✓
Distinct name		✓	✓	✓
Full inheritance	✓	✓	✓	✓
Distinct origin		✓	✓	
Type compatibility	✓	✓	✓	✓
Type variable				✓
Reference consistency	✓			✓

Table 1 Comparison of four object-oriented database systems with respect to their class invariants. Some constraints (like the representation and type variable invariants) are in fact implicit in most models. The invariants associated to O_2 are implied by more general integrity constraints.

- *Representation invariant.* This constraint is explicit only in the GemStone system. It states that the properties of an object (attributes, storage format, etc.) must reflect those defined by its class.
- *Inheritance graph invariant.* The structure deriving from inheritance dependencies is restricted to form a connected, directed graph without circuits (so that classes may not recursively inherit from themselves), and rooted at a special predefined class called OBJECT. The GemStone system does not implement multiple inheritance and so requires the graph to be a tree.
- *Distinct name invariant.* All classes, methods and variables must be distinguished by a unique name. In addition, ORION requires inheritance links between classes to be unambiguously labelled; two links may therefore bear identical names, except if they are directed to the same class.
- *Full inheritance invariant.* A class inherits all attributes from its ancestors, except those that it explicitly redefines. Naming conflicts occurring because of multiple inheritance are resolved by applying some precedence scheme and selecting one attribute as the one being inherited.
- *Distinct origin invariant.* No repeated inheritance is admissible in ORION and O_2: an attribute inherited several times via different paths appears only once in a class representation.
- *Type compatibility invariant.* The type of a variable redefined in a subclass must be consistent with its domain as specified in the superclass. In all systems this constraint means that the new type must be a subclass of the original one. O_2 also requires a strict compatibility between the method signatures of a class and those of its descendents.

- *Type variable invariant.* The type of each instance variable must correspond to a class in the hierarchy.
- *Reference consistency invariants.* GemStone guarantees that there are no dangling references to objects in the database. When a user holds a reference to an object, then both the object and the reference to it are retained. Instances can only be deleted when they are no longer referred to. OTGen requires that two references to the same object before modification also point to the same entity after modification.

The designers of ORION list additional rules to select the most meaningful way to maintain these invariants in the presence of ambiguous class descriptions or when a class schema is modified. These rules serve in particular to solve name clashes caused by multiple inheritance, to define how properties are propagated from a class to its subclasses, and to manipulate the inheritance graph.

3.3 Primitives for class evolution

Updates to a schema are inferred from an analysis of the static structure of classes [40]. These changes are subsequently assigned to a relevant category in a pre-determined taxonomy that covers all possibilities for schema evolution. Every definition affected by these modifications must then be adjusted. If the invariant properties of the inheritance hierarchy cannot be preserved, the transformation of the class structure is rejected.

- *Adding an attribute*, whether it is a variable or a method, is an operation that must be propagated to all descendents of the class where it is initially applied, in order to preserve the full inheritance invariant. When a naming or a type compatibility conflict occurs, or when the signature of the new method does not match the signature of other methods with the same name related to it via inheritance, one either disallows the operation (as in O_2 and GemStone), or resorts to conflict resolution rules. Thus, in ORION, if an attribute with the same name exists in the original class schema, it is replaced with the new one. On the other hand, an attribute from a subclass bearing an identical name with the new, inherited one takes precedence over it. In all systems, instances of all modified schemas are assigned an initial value for their additional variables which is either specified by the user or the special nil value.
- *Deleting an attribute* is allowed only if the variable or method is not inherited from an ancestor. Because of the full inheritance and representation invariants, the attribute must also be dropped from all subclasses of the definition where it is originally suppressed[1]. If a subclass, or the class itself, inherits a variable or a method with an identical name through another inheritance path, this new attribute replaces the deleted one. Of course, all instances lose their values for deleted attributes. O_2 forbids the suppression of attributes if the operation results in naming conflicts or in type mismatches with other attributes.

1. It is interesting to note that this propagation is not performed automatically in GemStone.

Scope of changes	GemStone	ORION	O_2
Instance variables			
add a variable	✓	✓	✓
remove a variable	✓	✓	✓
rename a variable	✓	✓	✓
redefine the type of a variable	✓	✓	✓
change the origin		✓	
change the default value		✓	
Methods			
add a method		✓	✓
remove a method		✓	✓
rename a method		✓	✓
redefine the signature			✓
change the code		✓	✓
change the origin		✓	
Other attributes			
add a shared value		✓	
change a shared value		✓	
remove a shared value		✓	
remove a composite link		✓	
Classes			
add a class	✓	✓	✓
remove a class	✓	✓	✓
rename a class		✓	✓
Other class properties			
make a class indexable	✓		
make a class non-indexable	✓		
Inheritance links			
add a superclass to a class		✓	✓
remove a superclass		✓	✓
change superclass precedence		✓	

Table 2 A comparison of schema evolution taxonomies.

- *Attribute renaming* is forbidden if the operation gives rise to ambiguities in the class itself or in its subclasses, or, in GemStone, if the attribute is inherited from an ancestor. If this is not the case, the change is propagated to all schemas affected by the renaming.
- The *type* of a variable (or of a method argument) can rarely be arbitrarily modified because of the subtyping relations imposed by the compatibility invariant. In ORION and GemStone, the domain of a variable can be generalized—but not beyond its original type, if the attribute is inherited.

  GemStone also allows a variable to be specialized, except if the new domain causes a compatibility violation with a redefinition in a subclass. Operations that are neither specializations nor generalizations are not directly supported; moreover, type changes are not propagated to subclasses. Instances violating new typing constraints have their variables reinitialized to nil.

- Other attribute properties, like the default value of a variable, the realization of a method—or, in Smalltalk, the grouping of messages into categories—can be modified. These operations usually do not require any elaborate consistency checks. Changing the origin of an attribute is an operation supported only in ORION. It serves to override default inheritance precedence rules and is logically handled as a suppression followed by the insertion of an attribute. Subclasses are adapted to conform to the new inheritance pattern.
- *Shared values* correspond to the class variables of Smalltalk and to the shared slots of CLOS. In ORION, instance variables can be converted to shared variables and their global value can be modified; the operation is propagated to the subclasses affected by these changes. A variable can also revert to being a normal instance variable, in which case its value is reinitialized to nil.

  Some environments extend the standard object model with other kinds of attributes. ORION defines composite links as special variables used to implement aggregation hierarchies. Suppressing a composite link breaks an aggregation relation by disconnecting an entity from its dependent object.
- *Adding a class* to an existing hierarchy is a fundamental operation for object-oriented programming, and as such it appears in all systems examined here. Connecting a new class to the leaves of a hierarchy is trivial—possible conflicts caused by multiple inheritance are solved with standard precedence rules. GemStone allows inserting a class in the middle of an inheritance graph, provided the new class does not initially define any property: this basic template may be subsequently augmented by applying the attribute manipulation primitives described in the preceding pages. With O_2, a new class may be connected to only one superclass and one subclass initially. The definition must specify which inherited attributes are superseded—the redeclaration must comply with subtyping compatibility rules—as well as the newly introduced attributes.
- *Removing a class* causes inheritance links to be reassigned from its ancestors to its subclasses. All instance variables that have the class as their type are assigned the suppressed class's superclass as their new domain. GemStone assumes that a class to be discarded no longer defines any property and that no associated instances exist in the database. O_2 forbids class deletion when this would result in dangling references in other definitions, when instances belonging to the class still exist, or when the deletion leaves the inheritance graph disconnected.
- *Renaming a class* is allowed only if the new identifier is unique among all class names in the inheritance hierarchy.
- As with attributes, each object model may define supplementary class properties and their corresponding manipulation primitives. Indexable classes in GemStone store information that is accessed by an integer offset instead of by name, like normal variables, and for which space is allocated dynamically. Classes may have their indexable part re-

moved, except when this part is inherited; the modification is not propagated to subclasses. Classes may be made indexable, provided this operation does not violate a typing constraint in an already indexable subclass.

- *Adding a superclass* to a schema is illegal if the inheritance graph invariant cannot be preserved. In particular, no circuits must be introduced in the hierarchy. The consequences of this operation are analogous to those of introducing attributes in an object description.
- *The deletion of a class S* from the list of ancestors of a class *C* must not leave the inheritance graph disconnected. O_2 provides a parameterized modification primitive that enables the programmer to choose where to link a class which has become completely disconnected from the inheritance graph (by default, it is connected to OBJECT). One may also specify whether the attributes acquired through the suppressed inheritance link are preserved and copied to the definition of *C*.

  In most other systems, if *S* is the unique superclass of *C*, inheritance links are reassigned to point from the immediate superclasses of *S* to *C*. In the other cases, *C* just loses one of its ancestors; no redirection of inheritance dependencies is performed. Of course, the properties of *S* no longer pertain to the representation of *C*, nor to those of its subclasses. The primitives for suppressing attributes from a class are applied to convert the definition of all classes and instances affected by this change.
- *Reordering inheritance dependencies* results in effects similar to those of changing the precedence of inherited attributes.

3.4 Completeness and correctness

Two important issues have to be addressed to ensure that these primitives capture interesting capabilities. The first one concerns completeness: does the set of proposed operations actually cover all possibilities for schema modifications? The second question deals with correctness: do these operations really generate class structures that satisfy all integrity constraints?

These problems have been studied in the context of the ORION methodology, where it has been demonstrated that a subset of its class transformation primitives exhibits the desired qualities of completeness and, partially, of correctness. In contrast, the GemStone approach does not strive for completeness: only meaningful operations, which can be implemented without undue restrictions or loss of performance are provided. An interesting result is provided by the O_2 approach, where it is shown that although a set of basic update operations may be complete at the schema level (i.e. all changes to a class hierarchy can be derived from a composition of these essential operations), this same set may not be complete at the instance level, when changes are carried out on objects and not on classes. For example, renaming an attribute in a class definition is equivalent to deleting the attribute and then reintroducing it with its new name; if the same sequence of operations is applied to a variable of an object, the information stored in the attribute is lost.

Ensuring correctness of class changes is much more difficult than it appears at first sight: additional adjustments have to be performed in order to guarantee the consistency of all class representations. Thus, because a method implementation may depend on other methods and variables, one cannot consider the deletion of one attribute in isolation. This operation may have far-reaching consequences if an attribute is excluded from a class interface. Similarly, introducing a new method in a class may raise problems because the code of the method may refer to attributes that are not yet present in the class definition and because of implicit scoping changes. If the method supersedes an inherited routine, subclasses referring to the previous method may become invalid. The transformation of method definitions to take into account the consequences of class updates is not easily amenable to automation, and is not implemented in any of the systems mentioned here. The O_2 approach recognizes that class definitions have to be augmented with information on inter-attribute dependencies, scoping constraints (when referring to attributes that can be inherited from multiple ancestors), and interface usage (which methods from an object's interface are used by a method in another schema) in order to be able to guarantee the behavioural consistency of class modifications. This requires a thorough, and perhaps impractical, analysis of all methods' code.

3.5 Evaluation

Decomposing all class modifications into update primitives and determining their consequences brings several advantages. During class design, this approach helps developers detect the implications of their actions on the class collection and maintain the consistency of class specifications. During application development, it guides the propagation of schema changes to individual instances. For example, renaming an instance variable, changing its type or specifying a new default value usually has no impact on an application using the modified class. Introducing or discarding attributes (variables or methods), on the other hand, generally leads to changes in programs and requires the reorganization of the persistent store—although the conversion procedure can be deferred in some situations.

Depending on its modelling capabilities and on the integrity constraints, an object-oriented programming environment may provide different forms of class surgery. It is easy to envision a system where class definitions are first retrieved with a class browser, and then modified with a language-sensitive editor where each editing operation corresponds to a schema manipulation primitive like those of ORION or GemStone. Such an environment would nevertheless fall short of providing a fully adequate support for the design and evolution processes. Class surgery forms a solid and rigorous framework for defining "well-formed" class modifications; in this respect, it improves considerably over uncontrolled manipulations of class hierarchies that are more or less the rule with current object-oriented programming environments. But it limits its scope to local, primitive kinds of class evolution: it gives no guidance as to when the modifications should be performed, and does not deal with the global management of multiple, successive class changes carried out during software development.

4. Class Versioning

4.1 Issues

Ensuring that class modifications are consistent is not enough; they must also be carried out in a disciplined fashion. This is of utmost importance in environments where a number of programmers collectively reuse and adapt classes developed by their peers and made available in a shared repository of software components. The early experiences with the Smalltalk system demonstrated that the lack of a proper methodology for controlling the extensions and alterations brought to the standard class library quickly resulted in a disastrous situation: although every user eventually had, at his workstation, an environment perfectly tuned to his needs, the variations between individual class hierarchies were sufficient to hinder the further exchange of software, or at least to make its porting non-trivial [28].

In the case of single-user environments, the exploratory way of programming advocated by the proponents of the object-oriented approach requires some support so that a software developer may correct his mistakes by reverting to a previous stable class configuration. When experimenting with several variants of the same class—to test the efficiency of different algorithms, for example—care has to be taken to avoid mixing up class definitions and dependencies.

The acute need for structuring class evolution quickly prompted Smalltalk programers to develop ingenious schemes for recording, grouping and disseminating information on system changes [44]. The notion of project was introduced into the Smalltalk system to separate the workspaces relative to several development tasks. An independent log file associated with each such workspace enables programmers to record and manage the history of revisions brought to the classes of a project. However, all information is still shared among projects. Thus, every class added to the hierarchy in a project is immediately available to all other projects; updates to a class or a global variable are visible in the other workspaces. More elaborate mechanisms rely on a central database to store data about class changes, bug reports, bug corrections and other "goodies" [44]. A browser facilitates the retrieval and insertion of items in the database, which is accessible by all users on a local network connecting Smalltalk workstations together. Because the database acts like a common information pool, complicated procedures are required to incorporate all bug corrections into the hierarchy, to detect conflicts among the proposed updates, to clean up class definitions, to get rid of obsolete objects, and finally to generate a new running version of the Smalltalk system valid for all users. These techniques do not seem to scale well for large, distributed programming environments. Current approaches favour a more structured organization of software development and a tighter control of evolution based on class versioning.

Versioning basically consists in checkpointing successive, and in principle consistent states of a class structure. The creation and manipulation of versions raises complex issues that we discuss in the following sections.

- How is version management organized with respect to software development and sharing?
- How does one distinguish between different versions of the same class?
- What are the circumstances that justify the creation of new versions, and how is this operation carried out?
- What can be done to handle the relations between different and perhaps incompatible versions?

It should be noted that versioning can be applied at the level of instances as well. In fact, many approaches, derived from techniques used notably in the field of CAD/CAM, originally considered versioning for objects and not for class definitions. Since we are mainly interested in class design and evolution, we will not deal with this possibility. Furthermore, managing instance versions and class versions are quite similar tasks, so most of the techniques we describe in this article find their use in both domains.

4.2 The organization of version management

An environment for version management is divided in several distinct working spaces, each one providing a specific set of privileges and capabilities for manipulating different kinds of versions [11]. Three such contexts are generally recognized in the literature:

- A private working space supports the design and development activities of one programmer. The programmer acts as the administrator of his private space. The information stored in his private environment—in particular the software components he is currently designing or modifying—is not accessible to other users.
- All classes and data produced during a project are stored in a corresponding context, which is placed under the responsibility of a project administrator. They are made available to all people cooperating in the project, but remain hidden from other users, since they cannot yet be considered as tested and validated.
- A public context contains all released classes from all projects, as well as data on their status. This information is visible to all users of the system.

It is natural to associate one kind of version with each working space:

- Released versions appear in the public context. They are considered immutable and can therefore neither be updated nor deleted, although they may be copied and give rise to new transient versions.
- Working versions exist in project and possibly private contexts. They are considered stable and cannot be modified—but they can be deleted by their owner, that is, the project administrator or the user of a private context. Working versions are promoted to released versions when they are installed in the public repository; they may give rise to new transient versions.
- A transient version is derived from a released, a working or another transient version. It belongs to the user who created it, and it is stored in his private environment. Transient versions can be updated, deleted, and promoted to working versions.

A typical scenario begins when a project is set up to build a new application. The programmers engaged in the development copy from the public repository class definitions they want to reuse or modify for the project: these definitions are added to their private environments as transient versions. Each programmer individually updates these classes and perhaps creates other definitions (via usual subclassing techniques) in his context as additional transient versions. In order to try different designs for the same class, or to save the result of his programming activity, he may derive new transient versions from those he is currently working on, while simultaneous-

	Transient	Working	Released
Location			
public context			✓
project contexts		✓	
private contexts	✓	✓	
Admissible operations			
update	✓		
delete	✓	✓	
Origin			
from a transient version by	derivation	promotion	
from a working version by	derivation		promotion
from a released version by	derivation		

Table 3 Principal characteristics of version types. Some systems consider only two kinds of versions (transient and released) and two levels of contexts (private and public) for managing their visibility.

ly promoting the latter to working versions. When he achieves a satisfactory design for a software component, he installs it as a working version in the project context. Of course, these working versions can subsequently be copied by his colleagues and give rise to new transient versions in their respective environments. Once software components have reached a good stage of maturity in terms of reliability and design stability, they are released by the project administrator and made publicly available in the central repository.

Since all operations of version derivation and freezing are done concurrently, careful algorithms are required to ensure that the system remains consistent. Fortunately, all updates are applied to local, transient objects, and not directly to global, shared definitions. As a consequence, concurrency control does not have to be as elaborate as traditional database transaction mechanisms. Some systems, like IRIS, implement simple schemes enabling programmers to prevent other users from deriving new versions from an existing object by setting a lock on it [18].

4.3 Version identification

An essential problem to deal with concerns the identity of classes. It is no longer enough to refer to a software component by its name, since it might correspond to multiple variants of the same class. An additional version number, and possibly a context name, must be provided to identify unambiguously the component referred to.

When the version number is absent from a reference, a default class is assumed. Typical choices for resolving the dynamic binding of version references include:

- The very first version of the class referred to.
- Its most recent version. The idea behind this decision is that this version can be considered the most up-to-date definition of a class. This is a solution commonly used in object-oriented databases to bind version references in interactive queries.

- Its most recent version at the time the component making the reference was created. This is the preferred option for dealing with dynamic references in class specifications.
- A default class definition specified by the administrator in charge of the context. This definition, called a generic version, can be coerced to be any element in a version derivation history.

The default version is first searched for in the context where the reference is initially discovered to be unresolved; the hierarchy of contexts is then inspected upward until finding the appropriate definition. Thus, to bind an incomplete reference to a class made in a project context (i.e. a reference consisting only in the class name, without additional information), the system first examines the class hierarchy in the current context; if this context does not contain the class definition referred to, the search proceeds in the public repository. No private context is inspected, for stable versions are not allowed to refer to transient versions, which can be in the process of being revised. Similarly, dynamic references to classes in the public context cannot be resolved by looking for unreleased components in a project context. Naturally, dynamic binding can be resolved at the level of a private context for all classes pertaining to it.

If only the most recent version gives rise to new versions, there is in principle no need for a complex structure to keep track of the history of classes: their name and version number suffice to determine their relationship to each other. The situation where versioning is not sequential, i.e., where new versions derive from any previous version, requires that the system record a hierarchy of versions somewhat similar to the traditional class hierarchy. When a version is copied or installed in a context, the programmer decides where to connect it in the derivation hierarchy. AVANCE provides an operation to merge several versions of the same class; with this scheme, the derivation history takes the form of a directed graph without circuits [7].

The information on derivation dependencies is generally associated with the generic version of a class version set. Version management systems like IRIS or AVANCE implement a series of primitives for traversing and manipulating the derivation graph [5][7]. Programmers can thus retrieve the predecessors and the successors of a particular version, obtain the first or the most recent version of a class on a particular derivation path, query their status (transient, released, date of creation, owner), determine which version was valid at a certain point in the past and bind a reference to it, freeze or derive new versions, etc.

The management of versions and related data obviously entails significant storage and processing overhead. This is why in most systems one is required to explicitly indicate that classes are versionable by making them descendents of a special class from which they inherit their properties of versions. This class is often called Version, as in AVANCE and IRIS; the former system defines another class, Checkpointable, that provides reduced, lower-level versioning functionality.

4.4 Versioning and class evolution

The derivation of class versions is partly automatic and partly the result of user decisions. It is evidently impossible to delegate full responsibility to the system for determining when a transient version should be frozen and a new transient one created, or if a component is sufficiently polished to be released. Such actions must be based on design knowledge which is best mastered by the software developers themselves. The automatic generation of new versions triggered, for example, by update operations on object definitions is a scheme that has found limited application in practice.

Another difficulty arises because of the superimposition of versioning on the inheritance graph. For example, when creating a new variant for a class should one derive new versions for the entire tree of subclasses attached to it as well? A careful analysis of the differences between two successive versions of the same class gives some directions for handling this problem [7].

- If the interface of a class is changed, then new versions should be created for all classes depending on it, whether by inheritance (i.e. its subclasses) or by their instance variables (i.e. classes using variables whose type refers to the now modified definition).
- If only non-public parts are changed, like the methods visible only to subclasses (such methods are called "protected methods" in C++), the type of its variables, or its inheritance structure, then versioning can be limited to its existing subclasses.
- If only the realizations of methods are changed, no new versions for other classes are required; this kind of change is purely internal and does not affect other definitions.

For reasons analogous to those exposed above, some approaches prefer to avoid introducing a possibly large number of new versions automatically and rely instead on a manual procedure for reestablishing the consistency of the inheritance hierarchy. The users whose programs reference the class that has been updated are simply notified of the change and warned that the references may be invalid. Two strategies are commonly adopted to do this: either a message is directly sent to the user, or the classes referencing the modified object definition are tagged as invalid. In the latter case, class version timestamps are frequently used to determine the validity of references [11]. Thus, a class should never have a "last modification" date that exceeds the "approved modification" date of the versions it is referred by. When this situation occurs, the references to the class are considered inconsistent, since recent adaptations have been carried out on the component, but have not yet been acknowledged on its dependent classes. It is up to the programmer to determine the effects of the class changes on other definitions and to reset the approved revision timestamp to indicate that the references have become valid again.

Maintaining compatibility between entities belonging to different versions is a major issue, and an object-oriented system should provide support for dealing with this aspect of version management. Application developers may want to view objects instantiated from previous class versions as if they originated from the currently stable version, or they may want to forbid objects from old versions to refer to instances of future variants. These effects are seldom achieved by fully automatic means. For every new version, one must program special functions for mapping between old and new class structures. These functions filter the messages sent to objects, so that proper actions can be taken, like translating between method names, returning a default value when accessing a non-existent variable, or simply aborting an unsuccessful operation. We describe in more detail the functionality of such systems in the part devoted to update propagation.

4.5 Evaluation

Versioning is a particularly appealing approach for managing class development and evolution. Recording the history of class modifications during the design process brings several benefits: it enables the programmer to try different paths when modelling complex application domains, and it helps avoid confusion when groups of people are engaged in the production of a library of common, interdependent classes. Versioning also appears useful when keeping track of various implementations of the same component for different software environments and hardware platforms. Besides, the hierarchical decomposition of the programming environment into workspaces, the attribution of precise responsibilities to their administrators, and the possibilities afforded by this kind of organization (for example, the separation of the long-term improvement of reusable components from the short-term development of new applications) are considered to be particularly valuable for increasing the quality and efficiency of object-oriented programming [47].

The main drawback of versioning techniques resides in the considerable overhead they impose on the development environment. Programmers have to navigate through two interconnected structures, the traditional inheritance hierarchy and the version derivation graph. They have to take into account a greater set of dependencies when designing a class. The system must store all information needed for representing versions and their reciprocal links, and implement notification. Moreover, version management methodologies still lack some support for design tasks: at what point does a version stop to be a variant of an existing class to become a completely different object definition?

In spite of their overhead, class and object versioning techniques have proved invaluable in important application domains like CAD/CAM, VLSI design and Office Information Systems. They have therefore been integrated in several object-oriented environments, including Orwell [47], AVANCE [6], ORION [4] and IRIS [18].

CLASS REORGANIZATION

In an object-oriented environment, programmers are supposed to build applications chiefly in a bottom-up fashion, by reusing existing classes. Classes often require adaptations so that they fully suit the needs of software developers. Such modifications indicate that the current hierarchy is not satisfactory: if software components cannot be reused as they are, then one is well-advised to look for missing abstractions, to try making some classes more general, to increase modularity.

Tools that automatically restructure a class collection and suggest alternative designs can reduce the efforts required to improve a class hierarchy. The available methods range from informal guidelines for detecting and then correcting class deficiencies, to algorithmic approaches that adjust object definitions on the basis of structural criteria. As we show in the following sections, all these techniques face significant difficulties because of the lack of a clear supporting design model and because of the very general applicability of object-oriented mechanisms, notably inheritance.

5. Empirical Guidelines

5.1 Issues

The object-oriented approach aims at dramatically increasing programmer productivity by relying on the massive reuse of numerous pre-packaged software components. The experience gained with Smalltalk demonstrates that this goal can indeed be achieved, provided that the design of objects fulfils some important requirements:

- The application domain must be adequately represented in terms of objects.
- Standardized class interfaces must be adopted to allow polymorphism.
- Classes must encapsulate behaviour general enough to serve in many applications.
- The functionality must be properly decomposed into the various levels of a class hierarchy.

Object definitions exhibiting these desirable properties are easier to combine and refine for building new applications or other software components.

Since software components cannot be assumed to be perfectly reusable the first time they are built, developers of object-oriented libraries must be supplied with tools to assess the quality of an inheritance hierarchy, to spot design imperfections, and to improve class definitions. Some authors propose general, informal principles for carrying out reorganization tasks. These principles are based on the lessons drawn from the building of large collections of reusable classes, like the Smalltalk hierarchy, the Eiffel library, the ET++ class collection [49], etc.

5.2 General redesign rules

According to Johnson and Foote [25], a reorganization methodology should deal with three main characteristics of a class hierarchy:

- *The standardization of class interfaces.* Objects answering the same set of messages are easier to combine. The structure of the inheritance hierarchy is simpler and more understandable—since one may assume that definitions with identical interfaces satisfy similar properties. Common class interfaces allow polymorphism—objects with equivalent interfaces may be substituted with each other in a type safe way. Table 4 illustrates the result of applying this principle to the basic data structures of the Eiffel library.
- *The building of an abstract, coherent class hierarchy.* Abstract classes exhibit a higher degree of reusability—because they define general properties and are not tied to a particular, narrow application. Inheritance relationships consistently based on specialization facilitate the analysis of the hierarchy and the modelling in terms of existing classes—as opposed to the situation where inheritance is used for code sharing or implementation purposes.
- *The modularization of functionality.* Small, cohesive classes are more resilient to change, while large classes grouping loosely related functions are difficult to adapt and utilize in other programs.

Interface	STACK	ARRAY	QUEUE	H-TABLE
Old version				
	push	enter	add	insert
	pop		remove_oldest	delete
	top	entry	oldest	value
New version				
	put	put	put	put
	remove		remove	remove
	item	item	item	item

Table 4 Class interfaces for some basic data structures of the Eiffel library. The conventions adopted in the second version of the language have resulted in greater naming consistency across object definitions.

Several practical rules corresponding to each of these categories have been proposed [25]. As far as interface standardization is concerned, they are as follows:

- Adopt a uniform terminology for classes belonging to a similar group of concepts.
- Standardize the interface of classes that communicate with each other.
- Decrease the number of arguments of a method, either by splitting the method into several simpler procedures, or by creating a class to represent a group of arguments that appear often together. A method with a reduced number of parameters is more likely to bear a signature similar to some other method in a different class; both methods may then be given the same name, thus increasing interface standardization.
- Eliminate code that explicitly checks the type of an object. Rather than introducing large case statements to execute some actions on the basis of an object's class, one should invoke a standard message in the object and let it carry out the appropriate actions, as in the example in Figure 1

The next set of rules is used to find more general classes and introduce higher-level abstractions into the hierarchy.

- Factor out behaviour common to several classes into a shared superclass. Introduce abstract classes (with deferred methods) if convenient to avoid attribute redefinitions.
- Minimize the accesses to variables so as to reduce the dependency of methods on the internal class representation. This can be achieved for example by resorting to special accessors instead of referring directly to variables.
- Ensure that inheritance links match specialization relationships. If necessary, revert class dependencies to make the hierarchy conform to this constraint. For example, ELLIPSE

Handling class differences by explicit type testing:

```
class A
     variables
          V    :    OBJECT;
     methods
          M
               begin
                    ...
                    if V.Class () = B then
                         V.Method-X (); -- do something;
                    elseif V.Class () = C then
                         V.Method-Y (); -- do something completely different;
                    end if
                    ...
               end M;
     end;
```

Delegating class-specific code:

```
class A
     variables
          V    :    OBJECT;
     methods
          M
               begin
                    ...
                    V.TestClass ();
                    ...
               end M
     end;
class B
     methods
          TestClass
               begin
                    self.Method-X ();        -- do something;
               end TestClass;
     end;
class C
     methods
          TestClass
               begin
                    self.Method-Y ();        -- do something completely different;
               end TestClass;
     end;
```

Figure 1 Reorganizing classes to avoid explicit type testing.

should be a superclass of CIRCLE, even if it is possible (and perhaps advantageous for implementation purposes) to install it as a descendent of CIRCLE in the hierarchy [23].

We end with the rules used to modularize the functionality of object definitions.

- Split large classes into smaller classes.

- Factor out implementation differences into subclasses, or via multiple inheritance, into "mixins"; if possible, delegate functionality to other objects.
- Separate groups of methods that do not communicate. Such sets of methods represent either totally independent behaviour or different views of the same object, which are perhaps better represented by distinct classes.
- Decouple methods from global attributes or internal class properties by sending messages to parameters instead of to self or to instance variables.

These design principles are obviously intertwined; for example, the methods derived from the decomposition of a large procedure most probably require less arguments than the original routine. Similarly, suppressing test cases diminishes the size of methods. Factoring out commonalities in superclasses results in smaller subclass descriptions, with fewer redundant declarations.

5.3 Evaluation

All these guidelines are of course very general. They provide no accurate criteria to drive the restructuring process: the software maintainer is left on his own to detect the situations warranting a reorganization of the hierarchy and the best way to accomplish it. An important requirement is that the reorganization of a class should affect its clients as little as possible; this imposes additional constraints on the kind of modifications that may be safely applied to object definitions and to the inheritance graph. Some authors propose using a shortened description of a class (excluding details about method implementation) as a starting point for generalization [38]. Another technique consists in associating pre and post-conditions to each method and then analysing these invariants to discover abstraction opportunities.

Fortunately, some of the rules appearing in this informal reorganization framework are amenable to a rigorous formulation and a subsequent automation. Software developers can thus take advantage of both the informal, but extensive framework—which gives them considerable freedom for choosing class evolution paths, and its detailed formalization—with its facilities for in depth analysis of certain choices.

6. Restructuring Inter-Attribute Dependencies

6.1 Issues

The list of reorganization rules exposed in the previous section clearly shows that avoiding unnecessary coupling between object definitions and reducing inter-attribute dependencies are two important prerequisites for well-designed classes. This is clearly justified from a software engineering point of view, since tightly encapsulated classes are easier to reuse and to maintain. Two major issues have to be addressed:

- What are the inferior or "harmful" dependencies?
- How can unsafe expressions be automatically replaced with adequate constructs?

This problem has been studied by Liebermann et al. [32][33]; the results of their investigations have been condensed in a short design principle known as the Law of Demeter, and in a small set of techniques for transforming object definitions so that they comply with this law.

6.2 The Law of Demeter

The Law of Demeter distinguishes three types of inter-attribute dependencies, and three corresponding categories of relationships between class definitions:

- A class C_1 is an *acquaintance class* of method M in class C_2, if M invokes a method defined in C_1 and C_1 does not correspond to the class of an argument of M, to the class of a variable of C_2, to C_2 itself, or to a superclass of the aforementioned classes.
- A class C_1 is a *preferred-acquaintance class* of method M in C_2, if C_1 corresponds to the class of an object directly created in M or to the class of a global variable used in M.
- A class C_1 is a *preferred-supplier class* of method M in C_2, if M invokes a method defined in C_1, and C_1 corresponds to the class of a variable of C_2, or to the class of an argument of M, to C_2 itself, to a superclass of the aforementioned classes, or to a preferred-acquaintance class of M.

These definitions form what is called the "class form" of the law.

In its weak version, the law simply aims at minimizing the number of acquaintance classes over all methods. A stricter version states that methods may only have preferred-supplier classes. The so-called "object form" of the law prohibits references inside a method to objects that are not created directly by the method, are not variables introduced by the class where the method is defined, and do not correspond to arguments passed to the method, to global variables or to the pseudo-variable self (identifying the object executing the method).

In all versions of the law, the basic goal remains the same: a method should only access attributes which are closely related to it or to the class it belongs to. Typically, a method should not traverse the structure of an object to manipulate its internal variables. It should not access variables pertaining to its class definition when they are actually inherited from a superclass. It should not directly send messages to foreign objects created by another method. The elimination of these dependencies promotes information hiding and restricts the visible properties of classes to narrow interfaces. As a consequence, the degree of coupling between classes, the size of the methods and the number of their arguments diminish—thus facilitating the application of program validation techniques, increasing the chances for reuse, and simplifying software maintenance.

6.3 Application and examples

We illustrate the main reorganization aspects dealt with by the Demeter approach for a group of simple object descriptions. The example is an adaptation of the case discussed in [32].

Let us assume the following definitions:

```
class LIBRARY
    variables
        Catalog   :    CATALOG;
```

```
            Loans       :     LOAN;
            ...
        methods
            Search-book (title : STRING) returns LIST[BOOK]
                begin
                    books-found     :     LIST[BOOK];
                    books-found := Catalog.Microfiches.Search-book (title);
                    books-found.Merge (Catalog.Optical-Disk.Search-book (title));
                    return (books-found);
                end Search-book;
            ...
        end;

    class CATALOG
        variables
            Microfiches     :     MICROFICHE;
            Optical-Disk    :     CD-ROM;
            ...
        methods
            ...
        end;

    class CD-ROM
        variables
            Book-References     :     FILE[BOOK];
            ...
        methods
            Search-book (title : STRING) returns LIST[BOOK]
                begin
                    book            :     BOOK;
                    books-found     :     LIST[BOOK];
                    books-found.New ();
                    Book-References.First ();
                    loop
                        exit when Book-References.End ();
                        book := Book-References.Current ();
                        if title.Equal (book.Title) then
                            books-found.Add (book)
                        end-if;
                        Book-References.Next ();
                    end loop;
                    return (books-found);
                end Search-Book;
            ...
        end;

    class MICROFICHE
        variables
            Book-References     :     FICHES[BOOK];
            ...
        methods
            Search-book (title : STRING) returns LIST[BOOK]
                begin
                    ...
                end Search-book;
            ...
        end;
```

```
class BOOK
    variables
        Title       :   STRING;
        SubTitle    :   STRING;
        Authors     :   LIST[STRING];
        ...
    methods
        ...
    end;
```

The definition of class LIBRARY obviously does not conform to the law: method Search-book accesses internal components of Catalog—the variables Microfiches and Optical-Disk; it sends messages to these variables and receives as result objects that are not subparts of LIBRARY instances and whose type—LIST[BOOK]—is not a preferred-supplier class of LIBRARY. We also note that the algorithm for retrieving all references stored on the optical disk violates elementary encapsulation constraints: it manipulates the internal structure of books to find whether their title matches a specific search criterion.

It is clear that the details of scanning microfiche and CD-ROM files to find a particular reference should be delegated to the CATALOG class. This would make the querying methods of LIBRARY immune to alterations in the internal structure of the catalog—for example the replacement of the microfiches with an additional CD-ROM file. A first transformation (called "pushing" by Lieberherr et al.) allows us to get rid of this problem.

```
class LIBRARY
    variables
        Catalog     :   CATALOG;
        Loans       :   LOAN;
        ...
    methods
        Search-book (title : STRING) returns LIST[BOOK]
            begin
                return (Catalog.Search-book (title));
            end Search-book;
        ...
    end;
--
--  The definition of LIBRARY is now correct.
--

class CATALOG
    variables
        Microfiches     :   MICROFICHE;
        Optical-Disk    :   CD-ROM;
        ...
    methods
        Search-book (title : STRING) returns LIST[BOOK]
            begin
                books-found     :   LIST[BOOK];
                books-found := Microfiches.Search-book (title);
                books-found.Merge (Optical-Disk.Search-book (title));
                return (books-found);
            end Search-book;
        ...
    end;
```

```
--
--   The definitions of CD-ROM, MICROFICHE and BOOK remain unchanged.
--
```

This organization is still not adequate: the CATALOG class manipulates objects, returned by the Search-book methods of Microfiches and Optical-Disk, that do not belong to it. Moreover, the type of these objects does not correspond to a preferred-supplier class of CATALOG. In this case, it is rather difficult to delegate further the processing of searches for references to the components of CATALOG: merging the results of several sub-queries should remain the responsibility of the catalog object. On the other hand, this cannot be done without accessing LIST[BOOK] entities, which is in principle incorrect.

The Law of Demeter proposes to deal with such situations by making use of an additional method:

```
class CATALOG
    variables
        Microfiches     :   MICROFICHE;
        Optical-Disk    :   CD-ROM;
        ...
    methods
        Search-book (title : STRING) returns LIST[BOOK]
            begin
                return (self.Merge-refs (Microfiches.Search-book (title),
                                         Optical-Disk.Search-book (title)));
            end Search-book;
        ...
        Merge-refs (microfiche-refs : LIST[BOOK]; cd-rom-refs : LIST[BOOK])
            returns LIST[BOOK]
            begin
                return (microfiche-refs.Merge (cd-rom-refs));
            end Merge-refs;
    end;
```

The definition of CATALOG now fully complies with the law. Search-book no longer manipulates objects of type LIST[BOOK], but instead passes them to Merge-refs for processing. Because the latter method declares microfiche-refs and cd-rom-refs as arguments, it may invoke methods on them freely; LIST[BOOK] is effectively a preferred-supplier class of Merge-refs. The introduction of such a method may appear artificial, but it makes explicit the dependency between the CATALOG and LIST[BOOK] classes. The programmer is informed about these dependencies at the level of method signatures, while in the previous situation he had to delve into the code of Search-book to detect them.

The last transformation concerns the implementation of Search-book in CD-ROM. Ensuring the proper encapsulation of BOOK objects requires that their variables which are publicly visible be manipulated through special-purpose procedures like the slot accessors of CLOS. It is easy to derive the appropriate adjustments to the definitions of CD-ROM and BOOK that enforce this rule. According to the strictest interpretation of the law, BOOK objects cannot be considered preferred-suppliers of CD-ROM. A technique similar to the one described above for CATALOG suppresses this inconsistency, although it produces a somewhat cumbersome program structure. Conversely, it is perfectly legal to let Search-book manipulate the object books-found, of type LIST[BOOK], since this object is actually created by method itself.

```
class CD-ROM
    variables
        Book-References     :     FILE[BOOK];
        ...
    methods
        Search-book (title : STRING) returns LIST[BOOK]
            begin
                books-found     :     LIST[BOOK];
                books-found.New ();
                Book-References.First ();
                loop
                    exit when Book-References.End ();
                    if title.Equal (self.RefTitle (Book-References.Current ())) then
                        books-found.Add (Book-References.Current ());
                    end-if;
                    Book-References.Next ();
                end loop;
                return (books-found);
            end Search-Book;
        RefTitle (reference : BOOK) returns STRING
            begin
                return (reference.Get-Title);
            end GetRefTitle;
        ...
    end;

class BOOK
    variables
        Title       :     STRING;
        SubTitle    :     STRING;
        Authors     :     LIST[STRING];
        ...
    methods
        Get-Title returns STRING
            begin
                return (Title);
            end Get-Title;
        ...
    end;
```

The Demeter approach provides still another reorganization technique, the "lifting" method, but it can be applied only in rarer circumstances. The reader should refer to [33] for more details about this technique.

6.4 Mechanical reorganization

In [32], the authors sketch an algorithm for mechanically transforming programs that do not conform to the law into an equivalent legal form. A first trivial adaptation consists in replacing all direct references to inherited variables and to subcomponents of a variable by calls to specialized accessors in charge of retrieving or setting their value. All other violations are assumed to occur within nested expressions, when objects returned by a method become the target for further messages, as in the class definition below:

```
class $C_0$
    methods
        $M_0$ returns $C_n$
```

```
            begin
                x2   :    C2;
                x3   :    C3;
                ...
                xn   :    Cn;
                --
                --   X1 of class C1 is an argument of M0 or a variable of C0.
                --
                x2 := X1.M1 ();
                x3 := x2.M2 ();
                ...
                xn := xn-1.Mn-1 ();
                return (xn);
            end M0;
    end;
```

In a first step, the responsibility for carrying out the M_1 method is delegated to class C_1. Thus, C_0 now satisfies the law, since it no longer invokes a method on the object returned by M_1; remember that this object is not a preferred supplier of M_0.

```
class C0
    methods
        M0 returns Cn
            begin
                return (X1.New-M1 ());
            end M0;
    end;

class C1
    methods
        New-M1 returns Cn
            begin
                x2   :    C2;
                x3   :    C3;
                ...
                xn   :    Cn;
                x2 := self.M1 ();
                x3 := x2.M2 ();
                ...
                xn := xn-1.Mn-1;
                return (xn);
            end New-M1;
    end;
```

By an analogous transformation, we eliminate all method invocations on objects x_3 to x_n from C_1's definition:

```
class C1
    methods
        New-M1 returns Cn
            begin
                x2   :    C2;
                x2 := self.M1 ();
                return (x2.New-M2 ());
            end New-M1;
    end;

class C2
```

```
    methods
        New-$M_2$ returns $C_n$
            begin
                $x_3$   :   $C_3$;
                ...
                $x_n$   :   $C_n$;
                $x_3$ := self.$M_2$ ();
                $x_4$ := $x_3$.$M_3$ ();
                ...
                $x_n$ := $x_{n-1}$.$M_{n-1}$;
                return ($x_n$);
            end New-$M_2$;
    end;
```

We repeat this process until classes $C_{i=1..n-1}$ are of the form:

```
class $C_i$
    methods
        New-$M_i$ returns $C_n$
            begin
                $x_{i+1}$   :   $C_{i+1}$;
                $x_{i+1}$ := self.$M_i$ ();
                return ($x_{i+1}$.New-$M_{i+1}$ ());
            end New-$M_i$;
    end;
```

and class C_n is:

```
class $C_n$
    methods
        New-$M_n$ returns $C_n$
            begin
                return (self.$M_n$ ());
            end New-$M_n$;
    end;
```

These classes, except for C_0 and C_n, do not yet conform to the law of Demeter. For each of them, we must determine whether the return value of the M_i method is an instance of a preferred-supplier class of C_i. We must consider two cases:

- M_i returns a variable pertaining to C_i. Since x_{i+1} identifies a component of C_i, the method New-M_i conforms naturally to the law of Demeter.
- M_i does not return a variable belonging to C_i. New-M_i must then be recoded so that X_{i+1} becomes a preferred-supplier object of C_i. This is achieved by introducing an auxiliary method—as was done in the example of the previous section.

```
class $C_i$
    methods
        New-$M_i$ returns $C_n$
            begin
                return (self.Suppl-$M_i$ (self.$M_i$ ()));
            end New-$M_i$;
        Suppl-$M_i$ ($X_{i+1}$ : $C_{i+1}$) returns $C_n$
            begin
                return ($X_{i+1}$.New-$M_{i+1}$ ());
            end Suppl-$M_i$;
    end;
```

This description omits a number of elements that must be taken into account in more realistic situations. In particular, arguments passed with a message must also be checked for conformance to the law, since they might be the result of evaluating nested expressions involving non-preferred-supplier objects.

The other reorganization techniques available in the Demeter framework—"pushing" and "lifting"—produce interesting class structures, that are generally more intuitive than the systematic replacement of dubious dependencies with forwarding calls to auxiliary methods. Unfortunately, these techniques cannot be easily automated.

6.5 Evaluation

In spite of the fact that it can be only partly automated, the Law of Demeter provides a sound and useful conceptual framework for disciplining object-oriented programming and improving class design. Its application has revealed to be beneficial for guiding the design of modular object-oriented libraries. But, as Sakkinen points out, putting the law into practice raises several difficulties [45].

The Demeter rules cannot be completely enforced at compile-time with languages that allow message-passing expressions to be generated dynamically (for example, with LISP macros) or to be given as arguments (i.e. the method to invoke on an object is not known a priori), or that allow the introduction or redefinition of methods at run-time. Languages like LISP-CLOS, Smalltalk and Objective-C fall, to a various extent, into this category. Object aliasing (i.e. referring to the same entity through different names) also complicates the checking of expressions for conformance to the law. In general, the "class form" law of Demeter does not seem to be fully effective for untyped languages: since objects manipulated by the language expressions are not declared to belong to a certain class, it is not possible to determine the conformity of programs to the law by a static examination of their source code. Violations of the law can only be monitored and detected during program execution. This is why in practice the "object form" of the law appears to be more tractable as a framework for checking expressions at compile-time.

As far as typed languages are concerned, applying the Demeter principles is not always straightforward either. First, there are some pathological cases where the spirit of the law is violated, although all dependencies and message-passing patterns formally respect all Demeter rules stated on section 6.2. Fortunately, such anomalies are rare and occur only in very contrived situations. More importantly, the law requires significant enhancements and reformulation to handle language peculiarities correctly. Smalltalk allows classes and metaclasses to be targets for message-passing instructions, so these entities could presumably be considered to be preferred-suppliers of all methods referring to them. Eiffel implements a form of genericity where classes can bind type arguments inherited from their ancestors. Thus, for the generic class LIST[T], LIST[BOOK] is a descendent that binds the generic argument T to the class BOOK. It would be perfectly legitimate to view BOOK as a preferred-acquaintance or even a preferred-supplier of LIST[BOOK] methods. In the case of C++, the law of Demeter has to deal with mechanisms like friend functions or type casts. It must also take into account the fact that the objects and the constructs of the language do not exhibit equivalent properties. Basic data types like integers and characters, or structures and arrays do not behave like user-defined classes and may not be manipulated with the same primitives. For example, arrays are not explicitly created by invoking a constructor operation. Although appropriate adaptations have been proposed for C++, translating the law of Demeter into equivalent terms for a specific object-oriented language is not always a trivial task.

It must be noted that some of the issues considered by the law are already handled by the mechanisms of existing programming languages. In CLOS, generic reader and writer functions for accessing variables can be automatically generated when a new class is declared. With C++, the methods and variables of a class are separated into three categories with different scoping constraints. Public attributes are visible to all clients of a class; protected attributes are only accessible to its subclasses; private attributes are not visible outside a class definition. These constructs greatly help in enforcing the encapsulation rules advocated by the Demeter approach, but they are insufficient for coping with the other dependency problems or disciplining the use of object-oriented mechanisms; this is precisely why programming style guidelines and design methodologies like the law of Demeter are useful.

7. Advanced Class Reorganization Techniques

7.1 Issues

Several authors point out that improper class modelling is a common phenomenon and consider the recasting of class hierarchies as an unavoidable aspect of the design process [17][23][38]. From this point of view, the traditional separation between development and maintenance activities loses much of its relevance. The goal then is to deal with the imperfections of a class hierarchy. These imperfections may stem directly from the coding activities of a software developer who is producing new unstructured programs from a well-designed class library. Alternatively, the defects may be already present in the hierarchy, but are only revealed when attempts at class reuse fail or meet with considerable obstacles in spite of a careful modelling of the new application. A good reorganization methodology should detect the places in a class hierarchy warranting redesign and propose better ways to structure dubious class definitions.

We propose a new approach for automating precisely these tasks. We emphasize inheritance and the structuring of classes and class attributes as the essential abstraction mechanisms that drive the modelling of applications and the constitution of class hierarchies. This is in line with the general idea of object-oriented programming, which assumes that a great part of software development rests on the reuse capabilities afforded by inheritance. In contrast, the Demeter approach focuses on the calling and dependency patterns between methods.

Our fundamental hypothesis is that design flaws can be uncovered at the time an object description is added to a hierarchy. The new class may refine, override or eliminate altogether the properties inherited from its ancestors—which were found in the existing library and supposedly encapsulate some interesting functionality in a reusable form. The redefinitions may correspond to:

- Inadequate application of object-oriented mechanisms for deriving the new subclass.

- Defective structures in the library of reusable components, which hinder the incorporation of properties inherited from superclasses into new applications.

As Johnson and Foote note [25], a frequent problem lies in the fact that programmers often overlook intermediate abstractions needed for establishing clean inheritance dependencies, and develop components too specialised to be effectively reusable. The consequences of early class specialization are often manifest when modules are created, and become acute when other people try to reuse these same modules. By extracting as much information as possible from the inheritance and redefinition patterns of classes, one should be able to improve libraries of software components and gain insight into the modelling of class hierarchies.

The analysis of the tailoring operations proceeds in several stages:

- All redefinitions of inherited properties are decomposed into more primitive operations, which apply only to a limited class characteristic. For example, the redefinition of a method signature can be broken down into the renaming of the arguments and into the redeclaration of their type.
- Typical redefinition patterns are assigned to individual categories. Such categories represent typical ways for deriving new classes from existing ones; as such, they correspond to various semantics of the general inheritance mechanism.
- According to each of these categories, the new class definition is adjusted to make all abstraction steps explicit. The hierarchy is also reorganized so as to eliminate the need for non-legitimate uses of inheritance, i.e. those that do not strictly correspond to one of the aforementioned categories.

The reorganization process may create intermediate nodes in the inheritance graph, shuffle attributes among them and rearrange inheritance paths. It is carried out by an algorithm that we describe informally in the next pages.

7.2 Model and terminology

Our approach is based on a very simple model that encompasses the major characteristics of most object-oriented databases and programming languages. We have deliberately avoided the inclusion of language-specific constructs closely matching those of actual systems in order to avoid compromising the generality of our methodology.

A class *defines* a set of *attributes*, which can be either *variables* or *methods*. Variables are repositories for information hidden inside objects. A variable has a *name* and a *type*, which corresponds to the name of a class. The idea is that only objects from the specified class (or compatible with that class) may be assigned to the variable. We assume that the typing mechanism is integrated with class relationships, although it may not necessarily be grounded on it[1]. Primitive types like INTEGER or BOOLEAN have therefore an associated class.

1. CLOS constitutes a good example of this possibility: any traditional LISP object, like a list or an integer, can be tested for membership to a class, thanks to a common typing scheme for fundamental LISP types and user-defined classes. However, the definition and manipulation of objects from both categories are carried out by totally distinct mechanisms.

Methods form the executable part of an object; they correspond to functions and procedures in traditional programming languages. A method is identified by its *name*; it may define a list of *arguments* that must be supplied to the method when it is invoked. Arguments are identified by their *name* and have an associated *type*, that, as with variables, corresponds to a class in the system. Arguments may be classified into *input parameters*—that only serve to provide some information to the method—*output parameters*—that serve to return results to the invoker of the method—and *input-output parameters*—objects that are exchanged between the calling and the called method, and that can be modified freely by the latter. Since operations on objects are performed by sending messages, it is often not possible to determine whether an argument belongs to the input, output or input-output parameters in the absence of any explicit declaration in method headers; this would require an analysis of the side-effects of all messages that can be sent to an argument. Current object-oriented programming languages frequently make a distinction between the arguments supplied to a method and the result returned by it. In this case, it is natural to presume that arguments are all input-output objects, while the result is just an output value.

The code that a method executes upon invocation is called its *implementation*. A method may call other methods or refer to the variables defined in its class. For each implementation, we associate the list of methods and the list of variables it depends on for its proper execution. By default, the names appearing in these lists refer to the attributes defined in the class the method and its implementation belong to. If it is necessary to override scoping constraints to refer directly to a method or a variable defined in a superclass, we assume that the attribute identifier can be prefixed with the superclass name. The implementation of a method may be *deferred*; in this case, the class only defines the method *signature* (name and arguments) and leaves the code unspecified, to be determined in a subclass. Constructing so-called abstract or deferred classes is a technique widely used in object-oriented programming.

The attributes of a class are either *inherited* from other classes or *introduced* by the class itself. Because of multiple inheritance, naming conflicts may occur between attributes bearing identical identifiers but acquired through different subclassing paths. We suppose that some disambiguating mechanism allows one to distinguish correctly between attributes with similar names. It is reasonable to assume that attributes introduced in a class take precedence over those with the same name defined in the class's ancestors. 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.

A class may *redefine* the attributes inherited from its superclasses. For each ancestor, and for each redefined attribute pertaining to this ancestor, there is a specification of how its characteristics are overridden in the subclass. For example, it is possible to redeclare the type of a method argument and leave the remaining properties of the attribute unchanged. As with new, introduced attributes, a redefinition supersedes the inherited characteristic in the class definition.

Our model also allows inherited attributes to be *rejected* when a new subclass is added to a hierarchy, in which case they should not appear at all in the subclass definition. Rejecting attributes is not really a subclassing operation, but rather a modification of the inheritance schema. No current object-oriented language allows one to reject or to selectively inherit attributes from a superclass; to achieve this result, the hierarchy must be reorganized, by separating rejected and accepted attributes into different classes. Our approach assumes that the need for such modifications becomes apparent when extending the class hierarchy; it is therefore interesting to integrate this evolution pattern with our reorganization algorithm.

The attributes defined by a class are either *public*, i.e. accessible from the outside of an object, or *private*, inaccessible from other objects. Variables are in principle private to an object, but, if needed, they can be manipulated via special accessor methods.The *interface* of a class determines which of its methods are made publicly visible. The interface is not inherited and may be freely overridden in subclasses.

We use the general term of *property* to designate any of the characteristics belonging to a class definition, that is, its attributes, the redefinition performed by the class, its inheritance links and the specification of its interface. A property may exhibit several *facets*; for example, a method may be considered from the point of view of its name, its signature, the type of its arguments, its implementation, etc.

7.3 Principle of the algorithm

In order to avoid lengthy developments, we present only a general outline of our algorithm. Some details of its anatomy can be fairly easily inferred from the examples given in this and the next section; more complete information on its reorganization and simplification components, described below, can be found in [10].

The algorithm is built around three major components that operate on a class definition newly installed as a subclass in an existing hierarchy. This new class may contain any kind of attribute redefinition; it may also reject inherited properties and add its own set of characteristics to those acquired from its ancestors. The algorithm isolates the various abstraction steps that compose the set of redefinitions; for each of these steps, a new class is created, thus making explicit the succession of logical subclassing operations needed to arrive at the new class schema.

When the redefinitions are actually caused by a deficiency in the structure of inherited superclasses, the algorithm reorganizes the hierarchy to suppress these defects and ensure that the inheritance graph is only composed of legitimate subclassing relationships. This step is carried out by a separate procedure that traces unwanted properties (i.e. those that are redeclared because they can neither be inherited nor refined in their present form) up the hierarchy until reaching the nodes where they are introduced. For each of these classes, a node is created that contains properties common to both the superclasses of the initial hierarchy and to the newly introduced subclass. The inheritance links are then redirected to point from the subclass to these additional nodes rather than to the previous ancestors. These reorganization operations also result in the creation of supplementary classes.

A third and final pass is required to suppress redundant nodes from the graph. Basically, all new classes that do not introduce any additional properties relative to their ancestors, or which have only one descendent, are merged with other class definitions.

As an example of the first restructuring step (called the decomposition step), consider the following definition:

```
class DEQUE
    exports
        Put-front, Put-back, Get-front, Get-back.;
    variables
        First      :    INTEGER;
        Last       :    INTEGER;
        Counter    :    INTEGER;
        Store      :    ARRAY[OBJECT];
    methods
        Put-front (item  :    OBJECT)
            begin
                --    Add the new item at the front of the deque.
            end Put-front;
        Put-back (item  :    OBJECT)
            begin
                --    Add the new item at the end of the deque.
            end Put-back;
        Get-front returns OBJECT
            begin
                --    Retrieve the item at the front of the deque.
            end Get-front;
        Get-back returns OBJECT
            begin
                --    Retrieve the item at the back of the deque.
            end Get-back;
    end;
```

Let us suppose that a programmer inherits from this class to build a description for stacks of real numbers:

```
class STACK
    inherits
        DEQUE;
    exports
        Push, Pop;
    redefines
        Put-front, Get-front, Store;
    renames
        Put-front as Push, Get-front as Pop;
    variables
        Store      :    ARRAY[REAL];
    methods
        Push (item      :    REAL)
            begin
                super.Put-front (item);
            end Put-front;
        Pop returns REAL
            begin
                super.Get-front;
            end Get-front;
    end;
```

This subclass actually embodies several abstraction steps: a change in the terminology for the data structure, a change in the interface, and a specialization. To make these various utilizations of inheritance explicit, the decomposition procedure adds two intermediate classes arranged in the following hierarchy:

```
class STACK0
    inherits
        DEQUE;
    exports
        Push, Pop, Get-back, Put-back;
    renames
        Put-front as Push, Get-front as Pop;
    end;

class STACK1
    inherits
        STACK0;
    exports
        Push, Pop;
    end;

class STACK
    inherits
        STACK1;
    exports
        Push, Pop;
    redefines
        Push, Pop, Store;
    variables
        Store    :    ARRAY[REAL];
    methods
        Push (item    :    REAL)
            begin
                super.Put-front (item);
            end Put-front;
        Pop returns REAL
            begin
                super.Get-front;
            end Get-front;
    end;
```

Each one of the three abstraction levels corresponds to a consistent, typical utilization of inheritance:

- STACK_0 identifies the origin of the inheritance path and the renaming of properties acquired from its ancestors. Renaming operations may hint at an inconsistent terminology for specifying classes in the library—contrary to the rules stated in section 5.2—and may justify additional adjustments to the class collection.
- STACK_1 does not redefine any inherited attribute, but restricts the visible interface to operations Push and Pop, so that a deque is actually manipulated as a stack; STACK_1 is a *view* of DEQUE. Notice that since STACK_1 does not redeclare any inherited attribute, it actually corresponds to the general definition of a stack, that can store any kind of objects.
- The last class, STACK, is a *specialization* of the more general STACK_1 definition: instances from STACK can only handle objects of type REAL.

As an example of subclassing that warrants a reorganization of the existing hierarchy, consider the following situation:

```
class LARGE-DEQUE
     inherits
          DEQUE;
     exports
          Put-front, Put-back, Get-front, Get-back;
     redefines
          First, Last, Put-front, Put-back, Get-front, Get-back;
     rejects
          Store;     --   There is no longer any need for this variable;
                     --   LARGE-DEQUE uses a linked list instead of an array.
     variables
          First      :    NODE;
          Last       :    NODE;
     methods
          --
          --   The code in every method, rather than dealing with a fixed width array,
          --   takes advantage of the possibility to have nodes of the deque allocated
          --   dynamically.
          --
          Put-front (item  :    OBJECT)
               begin
                    --   Add the new item at the front of the deque.
               end Put-front;
          Put-back (item  :    OBJECT)
               begin
                    --   Add the new item at the end of the deque.
               end Put-back;
          Get-front returns OBJECT
               begin
                    --   Retrieve the item at the front of the deque.
               end Get-front;
          Get-back returns OBJECT
               begin
                    --   Retrieve the item at the back of the deque.
               end Get-back;
     end;
```

LARGE-DEQUE provides services similar to those of DEQUE, except for their implementation. It is clear that there is an abstract class underlying the description of both DEQUE (which should perhaps be called FIXED-SIZE-DEQUE) and LARGE-DEQUE. The reorganization algorithm is able to infer from the redefinitions and rejections performed by the latter class that the subclassing operation cannot be decomposed into normal inheritance relationships, and that the original hierarchy must be fixed.

```
class DEQUE₀
     --
     --   This class describes an abstract deque.
     --
     exports
          Put-front, Put-back, Get-front, Get-back.;
     variables
          First      :    OBJECT;
          Last       :    OBJECT;
          Counter    :    INTEGER;
     methods
          Put-front (item  :    OBJECT)
               begin
```

```
                  deferred;
              end Put-front;
          Put-back (item  :      OBJECT)
              begin
                  deferred;
              end Put-back;
          Get-front returns OBJECT
              begin
                  deferred;
              end Get-front;
          Get-back returns OBJECT
              begin
                  deferred;
              end Get-back;
      end;

  class LARGE-DEQUE
      --
      --      This class implements a variable sized deque
      --
      inherits
          DEQUE$_0$;
      exports
          Put-front, Put-back, Get-front, Get-back;
      redefines
          First, Last, Put-front, Put-back, Get-front, Get-back;
      variables
          First       :     NODE;
          Last        :     NODE;
      methods
          Put-front (item  :     OBJECT)
              begin
                  --      Add the new item at the front of the deque.
              end Put-front;
          Put-back (item  :      OBJECT)
              begin
                  --      Add the new item at the end of the deque.
              end Put-back;
          Get-front returns OBJECT
              begin
                  --      Retrieve the item at the front of the deque.
              end Get-front;
          Get-back returns OBJECT
              begin
                  --      Retrieve the item at the back of the deque.
              end Get-back;
      end;

  class DEQUE
      --
      --      This class implements a fixed sized deque.
      --
      inherits
          DEQUE$_0$;
      exports
          Put-front, Put-back, Get-front, Get-back;
      redefines
          First, Last, Put-front, Put-back, Get-front, Get-back;
```

```
variables
    First    :   INTEGER;
    Last     :   INTEGER;
    Counter  :   INTEGER;
    Store    :   ARRAY[OBJECT];
methods
    Put-front (item  :     OBJECT)
        begin
            --   Add the new item at the front of the deque.
        end Put-front;
    Put-back (item  :     OBJECT)
        begin
            --   Add the new item at the end of the deque.
        end Put-back;
    Get-front returns OBJECT
        begin
            --   Retrieve the item at the front of the deque.
        end Get-front;
    Get-back returns OBJECT
        begin
            --   Retrieve the item at the back of the deque.
        end Get-back;
end;
```

DEQUE and LARGE-DEQUE are now *concrete subclasses* of the abstract DEQUE$_0$ class. Notice that the signature of the methods are identical in all classes, that the variable Counter is needed for both DEQUE and LARGE-DEQUE, and that the First and Last attributes present in the abstract class are specialized in its descendents.

7.4 Application

By now, the reader should have a good intuitive understanding of the principles behind our class restructuring approach. In this section, we highlight the essential decomposition and reorganization operations that can be applied to a class hierarchy upon the introduction of a new class definition. Our restructuring method proceeds in several distinct steps, considering only a limited range of class properties at a time. The redefinitions of each property is often broken down into additional substeps, as explained below. We try to analyze class structures in a top-down fashion, from their most general to their most detailed characteristics. Thus, we first deal with classes as interrelated sets of named attributes; we then separate these attributes into public and private attributes. In the next step, we deal with the structure of method signatures, and then with the type of the variables and of the arguments. The last stage of the reorganization deals with method implementations.

Inheritance and renaming. This step only considers the inheritance links from the new class to existing ancestors in the hierarchy. Redefinitions of inherited properties (attributes or interfaces) may not be immediately applied—except for the renaming of attributes. STACK$_0$ exemplifies this reorganization step.

Interface restriction. All inherited attributes that should not appear in the new class's interface are excluded from it—as in the STACK$_1$ class in the previous example—provided some of them remain visible. In this case, a supplementary subclass is created with the new interface definition; this subclass is a *view* of its ancestor. If all inherited attributes become private to the new class, no additional node is inserted. The hierarchy is reorganized so that all inherited attributes that belong to the interface are originally visible in the ancestors; the idea is that no inherited private attribute may be made public, although any inherited public attribute may become private (via a restriction operation). If necessary, the nodes higher up in the hierarchy are reorganized to comply with this constraint.

Class extension. Introduced attributes are inserted into the class definition and those that must be part of the class interface are made public. For example, if STACK defined two additional methods like Empty and Full to query the status of the stack, these would appear in the interface at this point. The implementation of introduced methods is deferred to a lower subclass. If only introduced attributes are part of the class interface, the relationship with its ancestor is called a *modularization.* This relationship captures typical inheritance patterns like the following:

```
class BYTE-FILE
    exports
        Read, Write, Seek, Open, Close;
    ...
    end;

class SEQUENTIAL-FILE
    inherits
        BYTE-FILE;
    exports
        Read-Int, Write-Int, Read-Char, Write-Char, ..., Reset, Rewrite;
    ...
    end;
```

Signature simplification. When input arguments disappear from a method signature, this is an indication that the class requires less information to carry out the same service—perhaps because it assumes some default values for the now superfluous parameters. An additional subclass is created to make this *protocol simplification* explicit. If DEQUE had been defined as:

```
class DEQUE
    exports
        Put, Get;
    ...
    methods
        Put (item  :    OBJECT, location    :    0..1)
            begin
                --    Put item at front or back of the deque according to "location".
            end Put;
        Get (location    :    0..1) returns OBJECT
        begin
            --    Retrieve item at front or back of the deque according to "location".
        end Get;
    end;
```

then STACK would probably have been a simplification of a deque:

```
class STACK
    inherits
        DEQUE;
    exports
```

```
        Put, Get;
    redefines
        Put, Get;
    methods
        Put (item  :     OBJECT)
            begin
                super.Put (item, 0);
            end Put;
        Get returns OBJECT
            begin
                super.Get (0);
            end Get;
    end;
```

Uncoupling. Suppressing input-output arguments from the signature of methods is an important abstraction step: it indicates that the new class encapsulates more state, and that it is no longer necessary to exchange information between this class and its clients. A new class is created where all method redefinitions corresponding to an *uncoupling* are introduced. These methods remain deferred; if there are other redeclarations to carry out on these methods (type of arguments, implementation), they are to be analysed and explicited in another subclass.

At this stage, the hierarchy must be reorganized for all methods for which no simple redefinition of their argument list may be found. Let us take an example:

```
class M
    methods
        Do-Something (a : X, b : Y, c : Z, d : W) ...
    end;

class N
    inherits M;
    redefines Do-Something;
    methods
        Do-Something (a : X, e : U, f : V) ...
    end;
```

Three new classes are introduced in the hierarchy. A first class, common to M and N, defines the Do-Something method and is inherited by both M and N:

```
class MN
    methods
        Do-Something (a : X, i : T) ...
    end;

class M
    inherits MN;
    redefines Do-Something;
    methods
        Do-Something (a : X, i : MT) ...
    end;

class N
    inherits MN;
    redefines Do-Something;
    methods
```

```
          Do-Something (a : X, i : NT) ...
     end;
```

Notice that now M and N specialize a generic parameter called i. The types NT and MT are defined as follows:

```
class MT
     exports
          Set-b, Get-b, Set-c, Get-c, Set-d, Get-d;
     variables
          b    :    Y;
          c    :    Z;
          d    :    W;
     methods
          --
          --   All methods are accessors for the variables.
          --
     end;

class NT
     exports
          Set-e, Get-e, Set-f, Get-f;
     variables
          e    :    U;
          f    :    V;
     methods
          --
          --   All methods are accessors for the variables.
          --
     end;
```

All manipulations of b, c and d (respectively e and f) in the original implementation of Do-Something in class M (respectively N) must be replaced with calls to the appropriate accessors exported by i.

Other simpler kinds of reorganization may be required for arguments that change categories, like input parameters that are redeclared as input-output arguments.

Subtyping. We consider now the redeclaration of argument and variable types. A new class is created for all attributes that are redefined according to a *subtype* relationship. Our notion of subtype is identical to the one that is generally proposed in the literature. Class *A* is a subtype of *B* if and only if:

- The interface of *A* defines at least the methods present in the interface of *B*;
- The return values of *A*'s methods are subtypes of those of *B*'s methods;
- The input arguments of *B*'s methods are subtypes of the input arguments of *A*'s methods;
- Input-output arguments for all methods common to *A* and *B* are identical.

Subtyping is an inheritance relationship with interesting properties: an instance from class *A* can replace an object from class *B*, since all messages sent to it will be understood and the results will correspond at least to what the client of the class is expecting. The behavioural compatibility between classes is assumed as long as the conformance at the level of signatures hold.

Specialization. Specialization is an inheritance relationship with much weaker properties than subtyping, but that conveys important semantic content. In the example of section 7.3, the specialization of STACK_1, a class for managing stacks of objects of any kind, into STACK, a stack for managing real numbers, corresponds to our idea of specialization. Specialization is an important conceptual category among all utilizations of inheritance, and it should be distinguished as such.

Type redeclarations that are neither subtyping nor specializations entail a reorganization of the hierarchy. This reorganization creates intermediate nodes that are subsequently refined to give rise to the previously inherited attributes (whose definition had to be redeclared) on the one hand, and to the new class attributes on the other hand. If possible, new classes are added to the hierarchy to represent the common properties of the inherited and redeclared types. Thus,

```
class M
    variables
        Var  :    A;
    end;

class N
    inherits M;
    redefines Var;
    variables
        Var  :    B;
        --
        --    B is not a specialization or a subtype of A.
        --
    end;
```

results in the following structure, if A and B have no common superclasses:

```
class MN
    variables
        Var  :    GENERIC;
    end;

class M
    inherits MN;
    redefines Var;
    variables
        Var  :    A;
        --
        --    A is a binding of the generic type of Var.
        --
    end;

class N
    inherits MN;
    redefines Var;
    variables
        Var  :    B;
        --
        --    B is also a binding of the generic type of Var.
        --
    end;
```

MN is a *purely generic* class where the type of Var has to be bound in its descendents. If A and B have some superclasses in common, then it is possible to define a class called, say, AB-COMMON, that inherits from these common classes. Class MN defines then Var as being of type GENERIC[AB-COMMON]; this indicates a structure of *constrained genericity*, a technique available in some object-oriented languages, notably Eiffel.

Concretization. The implementation of deferred methods is specified at this stage. In fact, this is done only for methods whose definition cannot be frozen sooner because their implementation depends on attributes that are refined in lower subclasses. The class library is reorganized so as to eliminate unwanted implementations from the current class definition, or to purge it from all rejected attributes—in a manner similar to the second example of section 7.3.

7.5 Evaluation

With our reorganization approach, an inheritance graph may have to be thoroughly modified in order to accommodate a new object definition. In particular, a number of intermediate classes are inserted between the original definitions and the new schema, attributes are shuffled among classes, inheritance links may be redirected, etc.

Additional classes frequently represent shared modules of functionality; they correspond to constructs, such as the "mixins" of LISP, whose main purpose is not describe real-world entities, but rather to support the implementation of other classes. More importantly, the classes introduced during the decomposition, and also during the reorganization processes can serve as a rough estimate for the abstractions that are missing from the modelling of an application domain. Such defects are unavoidable; it is exceptional to achieve a stable, definitive class design without going through several iterations. New classes and inheritance links correspond to the places in the hierarchy warranting redesign.

Of course, the reorganization algorithms should be considered as an aid to design and maintenance, not as a compulsory procedure to apply blindly to a collection of software components. It seems evident that, because these 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 examine the outcome of the reorganization, to adjust it, and perhaps to embark on more comprehensive restructuring activities. When typical evolution and reorganization patterns emerge, they can be catalogued and help guide the design process [31].

However, it remains to be seen whether a reorganization algorithm like the one we propose is really effective as an automatic software engineering assistant, or if simpler and less ambitious tools provide a more adequate functionality for supporting class modelling.

7.6 Assessing the potential of class reorganization

Not all possibilities for reorganizing classes are taken into account by our approach. In particular, there is no facility for splitting methods, restructuring their code, or changing the client/server relationship between classes—for example, there is no automatic transformation of inheritance into delegation. We do not believe that these are shortcomings inherent to our methodology for the following reasons:

- Some of the reorganization issues are already addressed by other approaches. We have already described at some length the Law of Demeter and the dependency restructuring capabilities afforded by this methodology. There are also techniques for producing structured, goto-less programs from unstructured code [36], or for transforming abstract data structures [1][30]. These techniques are not intended for object-oriented environments, but they may find some use in this context.
- The results of our algorithm actually provide useful information for detecting opportunities for further reorganization. For example, modularization relationships hint at possibilities to use variables and delegation instead of inheritance, but transforming class definitions to adopt such a structure is extremely difficult, because of the sensitive issues associated with the binding of the self reference.
- Some of the problems facing software reorganization are simply not tractable. As an example, the attempts to extract the commonalities from the comparison of several procedures that constitute different instances of the same algorithm have met with very limited success [14]. The analysis of a table lookup module and of a root finding method to deduce a general binary search algorithm cannot be automated and requires considerable skill to be carried out manually.

In fact, the rather ad-hoc nature of all class reorganization approaches discussed so far highlights three problems with object-oriented programming:

- The lack of a formal object model prevents the determination of interesting properties of a class hierarchy (for example redundance-free inheritance relationships) and the application of transformations that preserve or improve these properties. Ideally, one would like to have the equivalent of the normal forms and the normalization algorithms available for the relational database model [35].
- We have always dealt with a very general description of classes. Obviously, finding a general way to reorganize any kind of class hierarchy is much more difficult than handling evolution in domain-specific object models, with more restricted structures and behaviours, and richer semantics. Applying reorganization to "scripts" [13], for example, may reveal to be more fruitful than restructuring C++ code.
- Reorganization attacks the object modelling problem backwards, i.e. when imperfections are discovered in a class library. It could perhaps be more profitable to support the initial design phases with better tools and methodologies. A great part of our reorganization algorithms is spent trying to separate different modelling utilizations of inheritance; it would be more interesting to force software developers to respect some design guidelines and work with operations like specialization, subtyping, modularization, etc, instead of letting them resort to the general, powerful, but unwieldy and undisciplined inheritance mechanism.

CHANGE PROPAGATION

Modifications brought to class specifications must be propagated to objects instantiated on the basis of old definitions [41]. Many environments require aborting, recompiling and restarting whole applications when class definitions are modified; in this case, no special mechanisms for updating instances are needed or possible. On the other hand, change propagation is a crucial issue for systems that implement persistent objects (notably object-oriented databases) or that allow dynamic class redefinitions at run-time (like LISP-CLOS). Discarding existing instances is evidently not feasible, since they may be involved in running applications and may contain useful, and perhaps long-lived information. To maintain the overall consistency of the system, all entities must nevertheless conform to the representation determined by the class they belong to. We distinguish three approaches to achieve this result:

- The simplest way to deal with change propagation consists in making sure that class evolution does not affect existing instances in any way. As we show in the following section, change avoidance is in many cases not an altogether unrealistic assumption.
- Instances can be transformed to become conformant with their new class description. Conversion implies that the structure of old entities has to be mapped to a new schema; avoiding loss of information is a delicate issue with such a procedure.
- Rather than physically updating objects, one can wrap them with an interface that filters all accesses to them and takes appropriate actions to make them compatible with their new class definition. Managing conformance relationships between successive class variants is the main problem to consider with the filtering approach.

There is not necessarily a one-to-one correspondence between these update propagation techniques and the class evolution methodologies we have described in the previous sections. Most methodologies try to reduce the costs associated with class evolution by drawing, when possible, on several approaches for adapting instances to class modifications.

8. Change Avoidance

Many class modifications do not actually require that instances be updated or enhanced with a compatibility preserving layer. Detecting when these situations arise is important, since one can then avoid the inconvenience of change propagation without giving up system consistency.

Class tailoring is the prime candidate for change avoidance: tailoring operations are carried out only for the purpose of defining additional subclasses; they do not at all impact previous superclasses. No matter how inherited properties are overridden, the modifications appear and take effect only at the level of the subclasses performing the redeclarations. New subclasses obviously have no associated instances, so there is no need to care about converting or filtering procedures. The language compiler or the interpreter enforces the changes and is able to bypass the characteristics acquired from superclasses efficiently, notably by using dynamic binding. Thus, object-oriented systems avoid updating instances at least when subclassing operations are considered.

Even when a class is directly updated, via class surgery, this does not always imply that instances created when an older schema was in effect must be transformed to comply with the new definition. Many evolution primitives exhibit no side-effects and can safely be applied without reorganizing running applications. Among the modification operations listed in section 3.3, the following bear no consequences on object structures:

- Adding a new class is an operation without any effects, since instances of the class do not exist. The restrictions imposed when inserting a class in the middle of an inheritance hierarchy—such as disallowing the introduction of additional variables—guarantee that subclass definitions, and hence their instances, are not affected by this operation.
- Renaming classes, methods and variables only affects the description of classes, not the structure of instances. This may not true for programs that explicitly manipulate class or attribute names. Objects that pass method names as arguments to other methods, or that store and manage information on class or attribute names, for example by relying on functions like class-of (which, in CLOS, returns the class name of an object), may become invalid after renaming operations.
- Changing the default value of a variable or a shared slot has no effect on instances, since these values pertain to the class definitions, not to the objects themselves.
- The implementation of a method can be changed freely: the code is associated and kept with a class definition, to be shared among all individual entities.
- Because no arbitrary changes to the domain of variables and arguments are allowed, one can guarantee that the values stored within existing objects remain compatible with their new type.

Many of the reorganization algorithms described in the previous section do not destroy information present in the graph when the reorganization process is launched. Even when inheritance links and properties are rearranged, previous class definitions remain valid—so that we can in principle dispense with a conversion of their entities.

9. Conversion

9.1 Issues

Transforming all entities whose class has been modified seems like the most natural approach for dealing with change propagation, and it has in fact been adopted in several object-oriented systems. This technique implies that instances are physically updated so that their structure matches the description of the class they belong to. Two important requirements must be met:

- Because there is in general not a direct or a unique correspondence between old and new class definitions, care has to be taken to avoid losing information.
- The conversion process has to be organized in such a way that it interferes as little as possible with the normal system operations.

A consequence of the first requirement is that ad-hoc reconfiguration procedures have to be programmed to accompany automatic conversion processes—whose capabilities to preserve the semantics of an application domain are obviously limited. The second requirement forces all conversion procedures to behave as atomic transactions (i.e. transformations must be applied completely to the objects involved in the conversion) and puts strong restrictions on their duration.

9.2 Instance transformation

CLOS provides a good example of how automatic conversion can be enhanced by the programmer to take supplementary integrity constraints into account [27]. In CLOS, instances whose class definition has been modified are automatically transformed to conform to their new representation. These transformations are performed according to the rules listed in Table 5. CLOS deletes from the objects all attributes which have been dropped from their class, including their associated accessors, adds and initializes those attributes that have been introduced in the class definition, and adapts the attributes whose status has passed from shared to local (or vice-versa)[1]. This conversion is carried out by a standard function called update-instance-for-redefined-class, which is inherited by every class in a hierarchy and can be customized by the programmer—typically with an :after or a :before daemon. The arguments passed to this function, and available for further processing in the user-specified method, are the following:

	New slot		
Old slot	Shared	Local	None
Shared	preserved	preserved	discarded
Local	initialized	preserved	discarded
None	initialized	initialized	—

Table 5 Default conversions carried out by CLOS on objects after a class modification. A slot corresponds to a variable. Preserved slots have their values left untouched. Discarded slots are removed and their values are lost. Initialized slots are assigned a value determined by the class the instance belongs to.

- The list of names of attributes added to the class definition.
- The list of names of attributes discarded from the class.
- A list containing attribute names with their original value, for all attributes dropped from the class definition as well as those converted from local to shared.
- Optional initialization arguments.

1. A shared slot is a variable shared by all instances of the same class; it is equivalent to a class variable in Smalltalk. A local slot corresponds to a normal instance variable.

This technique enables the programmer to take proper actions to correct and augment the default restructuring and reinitialization procedures provided by CLOS; it is thus possible to determine freely the mapping from an old to a new object schema.

The OTGen system provides a similar kind of functionality for transforming instances affected by a class modification, although this capability is presented to the user as a table-driven interface rather than as a programming feature attached to the inheritance hierarchy [29]. A table lists all class definitions whose instances have to be converted and suggests default transformations to apply—which can of course be overridden or extended by the user. The transformation operations possible with OTGen are as follows:

- Transfer objects belonging to the old class definition to the new database. Unchanged objects are simply copied from a database to another without any conversions.
- Delete objects from the database if their class has been deleted.
- Recursively transform the variables of an object into new values, according to the transformation rules listed here.
- Initialize the variables of an object. When the previous and new types of a variable are incompatible, the default action taken by OTGen consists in assigning a special nil value to it. The user can override the standard behaviour of the system by providing its own initial value, or perhaps giving a formula to compute the new value (for example to convert a number to an equivalent string of characters).
- Perform context-dependent changes. One may initialize variables based on previous information stored in the objects, or separate the instances from a class into two other categories based on the information they contain.
- Move information between classes, for example by shuffling variables among classes, without losing associated information.
- Introduce new objects for classes created while updating the hierarchy, and initialize their variables on the basis of information already stored in the database.
- Change local values to shared values.

Providing a framework to handle the most common transformations certainly eases the task of the programmer. It is however difficult to guarantee that such a pre-determined set of primitives effectively covers all possibilities for object conversion. When complex adaptations cannot be expressed with these operations, one is eventually forced to resort to special-purpose programs as in CLOS.

9.3 Immediate and delayed conversion

An important constraint with conversion concerns the time at which objects must be transformed.

Immediate conversion consists in transforming all objects at once, as soon as the corresponding class modifications are committed. This solution does not find much favour in practice, because it may entail the full unloading and reloading of the persistent object store and long service interruptions if a significant number of entities has to be converted. Furthermore, it raises major problems in distributed environments, where controlling the transformation of objects dispersed over several machines is far from straightforward. On the other hand, this technique provides ample opportunities for optimizing the storage and access paths to objects as part of the conversion process. Immediate conversion has been implemented in the GemStone object-oriented database system [43].

Lazy conversion consists in adapting instances on an individual basis, but only when they are accessed for the first time after a class modification. This method requires keeping track of the status of each object. When successive revisions are carried out on the same class, the system must record each associated conversion procedure, to be able to transform objects that are referenced after a long period of inactivity. Lazy conversion does not incur the drawbacks of system shutdown imposed by immediate conversion, at the price of degraded response time when instances are initially accessed after a class modification. This problem should be particularly apparent when a series of conversions must be sequentially applied to a dormant object, i.e. one that has not been used while its class was repeatedly being revised. Lazy conversion is nevertheless an appealing approach for applications with short-lived instances, that are rapidly garbage-collected and therefore do not even need to be converted. Lazy conversion attracts a lot of interest and is already proposed as the standard mechanism for CLOS. In order to compare the respective merits of immediate and lazy conversion, the O_2 system will eventually implement both techniques [50].

9.4 Evaluation

Conversion, and in particular lazy conversion, seems like a very attractive technique for propagating changes in an object-oriented system. It requires the programming of transformation functions, even when the environment supports automatic conversion, but there appear to be no other alternatives for resolving intricate compatibility conflicts. When the conversion of instances is infeasible, scope restriction techniques borrowed from the filtering approach may prove helpful.

10. Filtering

10.1 Issues

Conversion enforces the consistency of instance representation by physically reorganizing the objects involved in a class modification. Whether it is immediate or deferred, this operation entails non-negligible processing costs and may sometimes be superfluous: under some circumstances, one may not need to convert instances—because they have become obsolete due to class modification, or because they represent information that is not allowed to be modified for legal reasons, like accounting records. In these situations, it is preferable to ensure a partial compatibility between old and new object schemas, so that certain important applications may still use them, but without striving for making them perfectly interchangeable.

Filtering (or screening, as it is often called in the literature) is a general framework for dealing with this problem, most often used in combination with version management. It proceeds by wrapping a software layer around objects. The layer intercepts all messages sent to the enclosed object; these messages are then handled according to the object's version, to make it conform to the current or to a previous class description, or to cause an exception to be raised when an application uses an object with an unsuitable definition. Three major issues must be examined with this approach:

- How does one characterize the degree of compatibility between different class versions?
- How can one map instances from a class version to another?
- How far can a filtering mechanism hide class changes to the users?

10.2 Version compatibility

Fundamentally, filtering is a mechanism for viewing entities of a certain class version as if they belonged to another version of the same class. From the predecessor-successor relationship between versions, we identify two types of compatibility [2]:

- A version C_i is backwards compatible with an earlier version C_j if all instances of C_j can be used as if they belonged to C_i.
- A version C_i is forwards compatible with a later version C_j if all instances of C_j can be used as if they belonged to C_i.

In the first case, applications can use old instances as if they originated from new definitions. With the second form of compatibility, old programs can manipulate entities created on the basis of later versions.

Each class C is associated with a partial ordering of versions $\{ C_i \}$. We assume that, at any point in time, some C_i is considered the valid version of class C. Very often, the valid version corresponds to the most recent version of the class. Building on these definitions, we say that a class version C_i is *consistent* with respect to version D_j of another class D ($C \neq D$) if one of the following conditions is satisfied [2]:

- D_j was the currently valid version of D when C_i was committed. This is the usual situation; C_i references up-to-date, contemporaneous properties of D.
- D_k was the currently valid version of D when C_i was committed, D_j is a later version of D, and D_k is forwards compatible with D_j. Here C_i references an obsolete definition of D, but the forwards compatibility property allows it to work with instances created according to the new schema.
- D_k was the currently valid version of D when C_i was committed, D_j is an earlier version of D, and D_k is backwards compatible with D_j. Here C_i is supposed to manipulate an up-to-date representation of D; thanks to the backwards compatibility, it is nevertheless able to use instances generated from old versions.

10.3 Filtering mechanisms

Most primitives for class evolution destroy compatibility between successive versions and require the development of filters to compensate for their effects. The operations that causes problems when invoked on a non-compatible object are fairly primitive, and can be classified in a limited number of categories. Adding or removing attributes generates access violations when an object attempts to invoke a deleted method, or to read from or write into a non-existent variable. Changing variable or parameter types causes exceptions when assigning or passing an illegal value to a variable or a method argument, when retrieving an unknown value from a variable, or when a method returns unexpected results. These effects are summarized in Table 6.

A simple way to deal with this problem is to replace each access primitive with a routine specifically programmed to perform the mapping between different class structures. Thus, for each variable that violates compatibility constraints, one provides a particular procedure for accessing it in reading mode, and another procedure for accessing it in writing mode. These procedures may perform various transformations, like mapping the variable to a set of other attributes [2]. For example, if the "birthday" attribute of a person class has been replaced with an "age" variable, one has to provide the following procedures to ensure backwards compatibility:

- A read accessor that determines the age of a person based on the time elapsed between his recorded birthday and the current date.
- A write accessor that stores the age of a person as a birthday, computed on the basis of the current date and the age given as argument to the accessor.

Similarly, one must define two symmetrical operations to guarantee forwards compatibility:

- A read accessor for the birthday computes its value from the current time and the age stored in the object.
- Instead of directly storing the birthday, a write accessor records the age of the person, determined from the birthday given as argument to the accessor and the current date.

More generally, one can define so-called substitute functions for carrying out these mappings between objects with different structures as follows:

- A substitute read function $RC_{ij}A(I)$ is given an instance I of version i of class C. It maps the values of a group of attributes from this object to a valid value of attribute A of version j of C. In other words, it makes instances of class version C_i appear as if they contained the attribute A of class version C_j for reading operations.
- A substitute write function $WC_{ij}A(I,V)$ is given an instance I of version i of class C, and a value V for attribute A of C_j. It maps the value V into a set of values for a group of attributes defined in C_i. In other words, this function makes instances of class version C_i appear as if they could store information in attribute A, although this information is actually recorded in other variables.

Needless to say, it may be quite difficult in practice to find strictly equivalent translations from one class definition to another.

A second approach favours the use of handlers to be invoked before or after a failed access to the attribute they are attached to—a technique resembling LISP :before and :after daemons, and which has been implemented in the ENCORE system [46]. Pre-handlers typically take over when attempting to access a non-existent attribute, or when trying to assign an illegal value to

Scope of change	Compatibility	Consequences
add a variable	backwards	undefined variable in old objects
delete a variable	forwards	undefined variable in new objects
extend the type	backwards	writing illegal values into old objects
	forwards	reading unknown data from new objects
restrict the type	forwards	writing illegal values into new objects
	backwards	reading unknown data from old objects
add a method	backwards	undefined method in old objects
delete a method	forwards	undefined method in new objects
extend argument type	backwards	passing illegal values to old objects
	forwards	getting unknown data from new objects
restrict argument type	forwards	passing illegal values to new objects
	backwards	getting unknown data from old objects
change argument list	backwards and forwards	similar to dropping and adding a method

Table 6 Consequences of class changes on version compatibility. The middle column indicates which kind of compatibility is affected by a modification, the right column describes the exceptions raised when accessing an object from the old or the new class definition.

it. A pre-handler may perform a mapping like those carried out by the substitute functions, coerce its argument to a valid value, or simply abort the operation. A post-handler executes when an illegal value is returned to the invoking object; a common behaviour in this case consists in returning a default value.

10.4 Making class changes transparent

Where should filters be defined? As originally stated, the technique based on handlers requires global modifications in all versions of the same class. More precisely,

- Whenever an attribute is added to a class, pre-handlers for the attribute must be introduced in all other versions of the class.
- Pre-handlers must be added to a version that suppresses attributes from a class definition.
- When a version extends the domain of an attribute, pre and post-handlers must be introduced in all other versions.
- When the domain of an attribute is restricted, the class version redeclaring the attribute type must be wrapped with a pre handler and a post-handler.

This solution is obviously inelegant: it requires that old class definitions be adjusted to reflect new developments, it entails a lot of cross-checking between version definitions and leads to a combinatorial explosion of handler complexity that is avoided only at the cost of introducing special kinds of inheritance links in the class hierarchy.

The model of substitute functions allows one to exploit the derivation history for mapping between versions that have no direct relationships. Thus, one can map a version C_i to another version C_j if there exist either substitute functions for them ($RC_{ij}X$, $WC_{ij}X$, where X denotes an attribute of C_j), or a succession of substitute functions that transitively apply to them (i.e. there are substitute functions for mapping between C_i and C_k, then C_k and C_l, and eventually C_l and C_j for example). Depending on compatibility properties, one can even relate class definitions placed in different derivation paths in a version hierarchy. Furthermore, substitute functions are defined just in the newer versions; previous class definitions remain unchanged.

When compatibility between versions cannot be achieved, one may install scope restrictions that isolate objects pertaining to different definitions from each other:

- A forward scope restriction makes instances from a new version inaccessible to objects from older versions.
- A backward scope restriction makes instances from older versions unreachable from objects of new versions.

By relying on scope restrictions and compatibility relationships, it is possible to partition the instance set of a class in such a way that operations may be applied to any object regardless of its version. Naturally, interoperability decreases with such a scheme, since the entities from different versions of the same class can no longer be referred to and accessed as members of one large pool of objects.

10.5 Evaluation

From our discussion, it appears that filtering cannot really fulfil its objective of making class changes transparent without considerable complexity and overhead. The programmer must not only develop a series of special-purpose functions for mapping between the variants of a class, but must also accept a degradation of application performance as these handlers accumulate, replacing the originally simple and efficient accessors. In practice, this complexity does not appear fully warranted: with lazy conversion, for example, one has also to define ad-hoc procedures for transforming entities from one version to another, but these procedures are called only once for every object; their execution is therefore not as expensive as the systematic run-time checks and exception raising implied by screening techniques. On the positive side, filtering provides a rigorous framework for defining and dealing with compatibility issues in object-oriented systems.

Screening has been implemented in some systems, but there its application scope is notably reduced. ORION does not immediately convert instances affected by a class change so as to avoid reorganizing the database [4]; when an instance is fetched, and before its attributes are accessed, deleted variables are made inaccessible (after, if needed, the physical destruction of the objects they refer to). Default values are automatically supplied to account for the introduction of new properties. Rearrangements of inheritance patterns are reflected by hiding unwanted properties and supplying default values for newly inherited ones. In Eiffel, methods can be tagged as obsolete, thus effectively providing a two-level kind of versioning. Obsolete methods can still be invoked, but they no longer appear in the documentation produced by the system, and references to them generate warnings at compile-time.

11. Conclusion

Object-oriented development reveals its iterative nature as successive stages of subclassing, tailoring, class modification, version creation and reorganization allow software engineers to build increasingly general, reusable and robust classes. We expect therefore software information systems to take advantage of a large spectrum of tools and techniques for managing class evolution. Even without comprehensive object-oriented CASE environments, software developers will certainly draw significant benefits from partial capabilities for mastering change in class hierarchies—in the same way as programmers rely on the Smalltalk browser to inspect and reuse class collections, although this specific tool does not seem to scale well for large hierarchies. What remains to do, then, is to validate the existing approaches in the context of real industrial or commercial development and production environments.

It appears clearly that an object-oriented environment would draw a maximum advantage from the integration of the various evolution management approaches described in this paper. For example, reorganization methodologies could be used in combination with versioning to try different class designs in a secure and disciplined fashion, without losing information on previous arrangements of the class hierarchy. Some techniques however do present significantly more potential for solving or alleviating the problems of class evolution than others.

- Tailoring will remain a standard approach for adjusting inherited properties to the needs of new applications—the concept of subclassing would lose most of its power in the absence of attribute redefinition. Because free tailoring can lead to chaotic inheritance structures, we expect this set of techniques to be severely constrained in the context of a class design methodology.

- Class surgery appears to be a prime candidate for integration with structured, language-sensitive development tools. Future object-oriented environments will likely provide language-sensitive tools. Thus, instead of using a standard Unix text editor to create and correct their programs, software engineers will probably rely on a browser or a graphical editor to review and modify their code, much like in the Smalltalk [21] or in the ET++ [19] environments. Attaching integrity constraints and invariant preserving checks to such tools would greatly enhance their functionality and should not raise insurmountable difficulties.
- Versioning appears indispensable in the context of *software communities*, where very large class collections must be shared by groups of programmers over long periods of time [20].
- Coordinating the work of several programmers, taking into account the needs of customers using different hardware platforms and software environments, keeping track of design decisions and recording all information needed to build and debug a particular software release also require comprehensive capabilities for version management.
- Reorganization methods should play a prominent role during the design process. Their potential is currently limited by the lack of a rigorous object model and the generality of object-oriented mechanisms that prevents more application specific (and more interesting) reorganizations. Further work is needed in this area to arrive at algorithms that capture the semantics of a class hierarchy and embody advanced design criteria. The current rules used by reorganization methods are either too general and vague ("split large classes into smaller components") or too focused on specific details of class structures (thus, our algorithms are incapable of recognizing possible "frameworks" in a collection of classes to be reorganized, although they detect and decompose interface redefinitions into several consistent steps).
- We expect the tools and the compilers in an object-oriented environment to be capable of implementing change avoidance techniques whenever possible. It is our conviction that lazy conversion will prevail as the technique for propagating change to instances. Screening seems cumbersome to use and might entail unacceptable overhead when applied in its full extent.

We have demonstrated that class evolution is actually an aspect of the more general problem of class design. Techniques for managing class modification and reorganization would therefore benefit substantially from advances in design methodologies, the determination of sound programming styles, and the disciplining of object-oriented mechanisms, in particular inheritance.

References

[1] Serge Abiteboul and Richard Hull, "Restructuring Hierarchical Database Objects," Theoretical Computer Science, no. 62, pp. 3–38, North-Holland, 1988.

[2] Matts Ahlsén, Anders Björnerstedt, Stefan Britts, Christer Hultén and Lars Söderlund, "Making Type Changes Transparent," SYSLAB report 22, SYSLAB-S, University of Stockholm, Stockholm, 26 February 1984.

[3] Robert Balzer, "Evolution as a New Basis for Reusability," in *Proceedings of the Workshop on Reusability in Programming*, pp. 80–82, Newport RI, September 1983.

[4] Jay Banerjee, Won Kim, Hyoung-Joo Kim and Henry F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in *Proceedings of the ACM-SIGMOD Conference on Management of Data*, eds. Umeshwar Dayal and Irv Traiger, pp. 311–322, Association for Computing Machinery, San Francisco, 27–29 May 1987.

[5] David Beech and Brom Mahbod, "Generalized Version Control in an Object-Oriented Database," in *Proceedings 4th IEEE International Conference on Data Engineering*, pp. 14–22, Los Angeles, February 1988.

[6] Anders Björnerstedt and Stefan Britts, "AVANCE: An Object Management System," in *OOPSLA 88 Proceedings*, ed. Norman Meyrowitz, pp. 206–221, San Diego, 25–30 September 1988.

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

[8] Alexander Borgida, "Modelling Class Hierarchies with Contradictions," SIGMOD Records, vol. 17, no. 3, pp. 434–443, ACM, September 1988.

[9] Alexander Borgida and Keith E. Williamson, "Accommodating Exceptions in Databases, and Refining the Schema by Learning from them," in *Proceedings VLDB 1985*, eds. Alain Pirotte and Yannis Vassiliou, pp. 72–81, Stockholm, 21–23 August 1985.

[10] Eduardo Casais, "Reorganizing an Object System," in *Object Oriented Development*, ed. D. C. Tsichritzis, pp. 161–189, Centre Universitaire d'Informatique, Genève, 1989.

[11] Hong-Tai Chou and Won Kim, "A Unifying Framework for Version Control in a CAD Environment," in *12th VLDB Conference Proceedings*, pp. 336–344, Kyoto, 25–28 August 1986.

[12] Brad J. Cox, *Object-Oriented Programming—An Evolutionary Approach*, Addison-Wesley, Reading, Massachusetts, 1986.

[13] L. Dami, E. Fiume, O. Nierstrasz and D. Tsichritzis, "Temporal Scripts for Objects," in *Active Object Environments*, ed. D. C. Tsichritzis, pp. 144–161, Centre Universitaire d'Informatique, Genève, 1988.

[14] Nachum Dershowitz, "Programming by Analogy," in *Machine Learning: an Artificial Intelligence Approach (vol. II)*, eds. Ryszard S. Michalski, Jaime G. Carbonell and Tom M. Mitchell, pp. 395–423, Morgan Kaufmann oublishers, Los Altos (CA), 1986.

[15] V. Dhar and M. Jarke, "Dependency Directed Reasoning and Learning in Systems Maintenance Support," IEEE Transactions on Software Engineering, vol. 14, no. 2, pp. 211–227, IEEE, February 1988.

[16] Jim Dietrich and Jack Milton, "Experimental Prototyping in Smalltalk," IEEE Software, vol. 4, no. 3, pp. 50–64, IEEE, May 1987.

[17] Gerhard Fischer, Andreas C. Lemke and Christian Rahtke, "From Design to Redesign," in *Proceedings of the 9th International Conference on Software Engineering*, pp. 369–376, IEEE, Monterey CA, 30 March–2 April 1987.

[18] D. H. Fishman, J. Annevelink, D. Beech, E. Chow, T. Connors, J. W. Davis, W. Hasan, C. G. Hoch, W. Kent, S. Leichner, P. Lyngback, B. Mahbod, M. A. Neimat, T. Risch, M. C. Shan and W. K. Wilkinson, "Overview of the IRIS DBMS," in *Object-Oriented Concepts, Databases, and Applications*, eds. Won Kim and Frederic H. Lochovsky, pp. 219–250, Addison-Wesley/ACM Press, 1989.

[19] Erich Gamma, André Weinand, and Rudolf Marty, "Integration of a Programming Environment into ET++: A Case Study," in *Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP 89)*, pp 283–297, Nottingham, 10–14 July 1989.

[20] S. Gibbs, D. Tsichritzis, E. Casais, O. Nierstrasz, and X. Pintado, "Class Management for Software Communities," in *Object Management*, ed. D. C. Tsichritzis, Centre Universitaire d'Informatique, 1990.

[21] Adele Goldberg, *The Smalltalk Programming Environment*, Addison-Wesley, Reading, Massachusetts, 1984.

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

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

[24] Matthias Jarke and Thomas Rose, "Managing Knowledge about Information System Evolution," SIGMOD Records, vol. 17, no. 3, pp. 303–311, ACM, September 1988.

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

[26] J. Karimi and B. R. Konsynski, "An Automated Software Design Assistant," IEEE Transactions on Software Engineering, vol. 14, no. 2, pp. 194–210, IEEE, February 1988.

[27] Sonya E. Keene, *Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS*, Addison-Wesley, Reading, Massachusetts, 1989.

[28] Glenn Krasner, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, Massachusetts, 1984.

[29] Barbara Staudt Lerner and A. Nico Habermann, "Beyond Schema Evolution to Database Reorganization," research paper on databases, Carnegie Mellon University, Pittsburgh, February 1990.

[30] Nicole Lévy, *Outils d'aide à la construction et à la transformation de types abstraits algébriques*, Thèse de $3^{ème}$ cycle, Université de Nancy, 1984.

[31] Qing Li and Dennis McLeod, "Object Flavor Evolution through Learning in an Object-Oriented Database System," in *Proceedings of the 2nd International Conference on Expert Database Systems*, ed. Larry Kerschberg, pp. 241–256, George Mason University, 25–27 April 1988.

[32] K. Lieberherr, I. Holland and A. Riel, "Object-Oriented Programming: an Objective Sense of Style," in *OOPSLA 88 proceedings*, pp. 323–334, 25 30 september 1988.

[33] Karl J. Lieberherr and Ian M. Holland, "Assuring Good Style for Object-Oriented Programming," IEEE Software, pp. 38–48, IEEE, September 1989.

[34] Henry Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," in *Proceedings of OOPSLA 86*, pp. 214–223, 29 September–2 October 1986.

[35] Y. E. Lien, "Relational Database Design," in *Principles of Database Design*, ed. S. Bing Yao, pp. 211–254, Prentice Hall, Englewood Cliffs, 1985.

[36] B. Maher and D. H. Sleeman, "Automatic Program Improvement: Variable Usage Transformations," ACM Transactions on Programming Languages and Systems, vol. 5, no. 2, pp. 236–264, ACM, April 1983.

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

[38] Bertrand Meyer, "The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design," in *TOOLS 89 proceedings*, pp. 13–23, Paris, 13–15 November 1989.

[39] David A. Moon, "Object-Oriented Programming with Flavors," in *Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '86)* Special Issue of SIGPLAN Notices, vol. 21, no. 11, pp. 1–8, Portland, Oregon, 2 September–29 October 1986.

[40] K. Narayanaswamy, "Static Analysis-based Program Evolution Support in the Common Lisp Framework," in *Proceedings of the 10th International Conference on Software Engineering*, pp. 222–230, IEEE, Singapore, 11–15 April 1988.

[41] G. T. Nguyen and D. Rieu, "Schema Change Propagation in Object-Oriented Databases," in *IFIP 89 proceedings*, ed. G. X. Ritter, pp. 815–820, Elsevier Science Publisher B.V. (North-Holland), 1989.

[42] O. M. Nierstrasz, "Object-Oriented Concepts," in *Active Object Environments*, ed. D. C. Tsichritzis, pp. 1–17, Centre Universitaire d'Informatique, Genève, 1988.

[43] D. Jason Penney and Jacob Stein, "Class Modification in the GemStone Object-Oriented DBMS," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)* Special Issue of SIGPLAN Notices, vol. 22, no. 12, pp. 111–117, Orlando, FL, 4–8 October 1987.

[44] Steve Putz, "Managing the Evolution of Smalltalk-80 Systems," in *Smalltalk-80: Bits of History, Words of Advice*, ed. Glenn Krasner, pp. 273–286, Addison-Wesley, Reading, Massachusetts, 1984.

[45] Markku Sakkinen, "Comments on the Law of Demeter and C++," SIGPLAN Notices, vol. 23, no. 12, pp. 34–44, ACM, 1988.

[46] Andrea H. Skarra and Stanley B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," in *Research Directions in Object-Oriented Programming*, eds. Bruce Shriver and Peter Wegner, pp. 393–415, The MIT Press, Cambridge, Massachusetts, 1987.

[47] Dave Thomas and Kent Johnson, "Orwell: a Configuration Management System for Team Programming," in *OOPSLA 88 Proceedings*, pp. 135–141, 25–30 September 1988.

[48] Peter Wegner and Stanley B. Zdonik, "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like," in *ECOOP 88 Proceedings*, eds. Stein Gjessing and Kristen Nygaard, Lecture Notes in Computer Science, pp. 55–77, Springer Verlag, Oslo, 15–17 August 1988.

[49] André Weinand, Erich Gamma, and Rudolf Marty, "ET++: An Object-Oriented Application Framework in C++," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '88)* Special Issue of SIGPLAN Notices, vol. 23, no. 11, pp. 46–57, November 1988.

[50] Roberto Zicari, "Schema Updates in the O_2 Object-Oriented Database System," report 89-057, Politecnico di Milano, Dipartimento di Elettronica, Milano, October 1989.