Chapter 8

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 for a wide range of applications or domains. Tools and methodologies are therefore needed to cope with the complexity of designing, updating and reorganizing class collections. We present a typology of techniques for controlling change in object-oriented systems, illustrate their functionality with selected examples and discuss their advantages and limitations.

8.1 Object Design and Redesign

8.1.1 The Problem

Nowadays, it is generally assumed that the mechanisms provided by object-oriented languages — namely classification, encapsulation, inheritance and delayed binding — together with a comprehensive set of interactive programming tools, provide the basic functionality required for the large-scale production of highly reusable software components. However, software developers working with an object-oriented system are frequently led to modify extensively or even to reprogram supposedly reusable classes so that they fully suit their needs. This problem has been documented during the design of the Eiffel [31] and Smalltalk [21] hierarchies, the construction of user interfaces [20], the development of libraries for VLSI-design algorithms [2], and the development of object-oriented frameworks for operating systems [23].

The first difficulty with object-oriented development is achieving a correct initial modelling of an application domain. 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. The problem is compounded by the versatility of the inheritance mechanism, which can serve to denote specialization relationships, to en-

Eduardo Casais, “Managing Class Evolution in Object-Oriented Systems,” *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 201-244, Prentice Hall, 1995.

Reproduced with the permission of the Publisher, Prentice Hall (a Pearson Education company). This work is protected by copyright and may not be reproduced other than when downloaded and viewed on a single Central Processor Unit (CPU) for private use only. It is not otherwise to be reproduced or transmitted or made available on a network without prior written permission of Prentice Hall. All other rights reserved.

force typing constraints, or to share implementations. Inadequate inheritance structures, missing abstractions in a hierarchy, overly specialized components or deficient object modelling may seriously impair the reusability of a class collection. Such defects must be eliminated through an evolutionary process to improve the robustness and the reusability of a library [20][22].

Even when a class collection embodies stable abstractions that have been reused successfully a number of times, repeated reorganizations of the library may still be unavoidable. Paradoxically, the high degree of reusability of a library may cause it to undergo major reorganizations when developers attempting to take advantage of its functionality stretch its range of application to new domains, thus imposing additional constraints on the library and invalidating the assumptions that drove its original design.

Software reuse also 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.

8.1.2 The Solutions

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

- *Tailoring* consists in slightly adapting class definitions when they do not lead to easy subclassing. Most object-oriented languages provide built-in constructs for making limited adjustments on class hierarchies.
- *Surgery.* Every possible change to a class can be defined in terms of 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 coordinate the modelling of variants in complex application domains.
- *Reorganization* of a class library is needed after significant changes are made on it, like the introduction or the suppression of classes. Reorganization procedures use information on “good” library structures to discover imperfections in a hierarchy and to suggest alternative designs.

A second problem, related to class evolution, is that instances must be updated after their representation is modified. Restarting a program and discarding existing instances is not always feasible, since objects may be involved in running applications and may contain useful, long-lived information. This is especially true for environments implementing persistent objects. We consider in detail three techniques to tackle this issue:

- *Change avoidance* consists in preventing any impact from class modifications on existing instances, for example by restricting the kind of changes brought to classes.

- *Conversion* physically transforms objects affected by a class change so that they conform to their new class definition.
- *Filtering* hides the differences between objects belonging to several variants of the same class by encapsulating instances with an additional software layer that extends their normal properties.

The remainder of the chapter explains the principles behind these approaches, referring when appropriate to the research prototypes or industrial products that implement them, and illustrating their functionality with simple examples.

8.2 Class Tailoring

8.2.1 Issues

Quite often, object-oriented 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 which need to be enforced in specialized subclasses [9]. Typically, one prefers an optimized implementation of a method to the general and inefficient algorithm defined in a superclass. Similarly, a variable with a restricted range may be more appropriate than one admitting any value. Tailoring mechanisms alleviate these problems by allowing the programmer to replace unwanted characteristics from standard classes with properties better suited to new applications.

8.2.2 Language Mechanisms

Object-oriented languages have always provided simple constructs for tailoring classes. We present here an overview of the tailoring mechanisms provided by the Eiffel language [30]. Similar mechanisms are available 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; a special undefine clause in Eiffel 3 allows the programmer to turn an inherited method into a deferred definition in a 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 (or the new post-condition) is weaker (or stronger) than the original one.

- *Interfaces* are not statically defined in Eiffel. An attribute declared as private in a superclass may be made accessible in a subclass; conversely, a previously visible attribute may be excluded from the subclass interface.

The following excerpt from the Eiffel 2.1 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, go_to_child,
        delete_child, change_child, attach_to_parent, change_right, change_left,
        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
        ...
    end -- class BINARY_TREE [T]
```

Sometimes, 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 [33]. 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 results 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.

8.2.3 Evaluation

Tailoring techniques are useful in performing small adjustments on 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 implemented efficiently within compilers.

On the other hand, overreliance on tailoring may quickly lead to incomprehensible structures overloaded with special cases, which are, 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. When signature compatibility is not respected, or when the semantics of a method can be radically altered, polymorphism becomes impossible; an instance of a class may no longer be used where an instance of a superclass is allowed. Changing attribute representations also cancels 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 [10]. Unfortunately, the techniques we have described in this section do not really help detect design flaws in object descriptions.

8.3 Class Surgery

8.3.1 Issues

Whenever changes are brought to the modelling of an application domain, corresponding modifications must be applied to the classes representing real-world concepts. These operations disturb a class hierarchy much more profoundly than tailoring: instead of overriding some inherited properties when new subclasses are defined, the structure of existing classes themselves must 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 [3][4][27][32][35]. There, the available methods determine the consequences of class changes on other definitions and on existing instances, as well, so that possible integrity violations can 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 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 analyzed. In general, additional reconfiguration procedures have to be applied in order to preserve schema invariants. It is for example illegal to delete an attribute from a class *C* if this attribute is really inherited from a superclass of *C*. If the attribute can be deleted, it must also be recursively dropped from all subclasses of *C*.
4. Finally, the effects of schema changes are reflected on the persistent store; instances belonging to modified classes are converted to conform to their new description.

We base our discussion on class surgery mainly on the research performed around the object-oriented database systems GemStone, ORION, O_2 and OTGen, although evolutionary capabilities based on this technique have been proposed for many other systems. We defer the description of instance conversion techniques to the section on change propagation.

8.3.2 Schema Invariants

Every class collection contains a number of integrity constraints that must be maintained across schema changes. These constraints, generally called schema invariants in the literature, impose a certain structure on class definitions and on the inheritance graph.

- *Representation invariant.* This constraint 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 acyclic graph (so that classes may not recursively inherit from themselves), possibly restricted to be a tree, and having as root a special predefined class usually called OBJECT.
- *Distinct name invariant*. All classes, methods and variables must be distinguished by a unique name.
- *Full inheritance invariant*. A class inherits all attributes from its superclasses, except those that it explicitly redefines. Naming conflicts occurring because of multiple inheritance are resolved manually, or by applying some default precedence scheme.
- *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 (or of a method argument) redefined in a subclass must be consistent with its domain as specified in the superclass. In all systems this means that the new type must be a subclass of the original one.
- *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; instances can only be deleted when they are no longer accessible. OTGen requires that two references to the same object before modification also point to the same entity after modification.

Schema invariants supported by four object-oriented database systems are summarized in table 8.1.

8.3.3 Primitives for Class Evolution

Updates to a schema are assigned to a relevant category in a predetermined taxonomy. 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. Schema evolution taxonomies are compared in table 8.2.

- *The insertion of an attribute*, whether it is a variable or a method, is an operation that must be propagated to all subclasses 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. In all systems, instances of all modified schemas are assigned an initial value for their additional variables that is either specified by the user or the special nil value.

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

Table 8.1 *Schema invariants of four object-oriented database systems. Some constraints (like the representation invariant) are implicit in most models.*

- *Deleting an attribute* is allowed only if the variable or method is not inherited. Because of the full inheritance and representation invariants, the attribute must also be dropped from all subclasses of the definition where it is originally deleted. If a subclass, or the class itself, inherits another variable or a method with the same 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.
- *Attribute renaming* is forbidden if the operation gives rise to ambiguities in the class or in its subclasses, or, in GemStone, if the attribute is inherited.
- The *type* of a variable (or of a method argument) can rarely be arbitrarily modified because of the subtype relations imposed by the compatibility invariant. In ORION and GemStone, the domain of a variable can be generalized. 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 supported; moreover, type changes are not propagated to subclasses. Instances violating new type constraints have their variables reset to nil.
- Properties like the default value of a variable or the body of a method can also be modified. 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. In addition, ORION provides operations to update shared variables and special aggregation links.
- *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

Scope of change	GemStone	O_2	ORION
Instance variables			
add a variable	✓	✓	✓
remove a variable	✓	✓	✓
rename a variable	✓	✓	✓
redefine the type of a variable	✓	✓	✓
change the inheritance origin			✓
change the default value			✓
modify other kinds of variables			✓
Methods			
add a method		✓	✓
remove a method		✓	✓
rename a method		✓	✓
redefine the signature		✓	
change the code		✓	✓
change the inheritance origin			✓
Classes			
add a class	✓	✓	✓
remove a class	✓	✓	✓
rename a class		✓	✓
modify other class properties	✓		
Inheritance links			
add a superclass to a class		✓	✓
remove a superclass		✓	✓
change superclass precedence			✓

Table 8.2 *A comparison of schema evolution taxonomies.*

new class to the leaves of a hierarchy is trivial — possible conflicts caused by multiple inheritance are solved with standard precedence rules. GemStone allows for 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 how inherited attributes are superseded, and these redeclarations must comply with subtyping compatibility rules.

- *Removing a class* causes inheritance links to be reassigned from the class's superclasses to its subclasses. All instance variables that have the deleted class as their type are assigned the suppressed class's superclass as their new domain. GemStone assumes that a class which is being discarded no longer defines any property and that no associated instances exist in the database. O_2 forbids class deletion if it results in dangling references in other definitions, if instances belonging to the class still exist, or if 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, such as the indexable classes in GemStone, and their corresponding manipulation primitives.
- *Adding a superclass* to a schema is illegal if the inheritance graph invariant cannot be preserved. In particular, no circuits may be introduced in a hierarchy. The consequences of this operation are analogous to those of introducing attributes in a class.
- *The deletion of a class S* from the list of superclasses 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 that 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 superclasses; 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.

8.3.4 Completeness, Correctness and Complexity

Three issues have to be addressed to ensure that class surgery captures interesting capabilities:

- *Completeness*: does the set of proposed operations actually cover all possibilities for schema modifications?
- *Correctness*: do these operations really generate class structures that satisfy all integrity constraints?

- *Complexity*: is it possible to detect violations of schema invariants and subsequently regenerate a schema conforming to these invariants in an efficient way?

The first two 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 that 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 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. Since 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 changes in the scope of attributes. If the method supersedes an inherited routine, subclasses referring to the previous method may become invalid. Not surprisingly, maintaining behavioural consistency across schema changes is an undecidable problem [39]. Dataflow analysis techniques, like those that are used by some compilers to check for type violations in object-oriented programs, can help detect the parts of the code that become unsafe because of schema updates, but they are typically pessimistic and might reject legal programs as incorrect [14]. Enriching the set of schema invariants to detect more (semantical) inconsistencies requires careful selection to avoid turning an efficient test procedure for constraint satisfiability into an NP, or even an undecidable problem [26][39]. As a consequence, all aforementioned systems capture relatively simple structural constraints with their schema invariants and give little support to update methods upon class alterations [41].

8.3.5 Evaluation

Decomposing all class modifications into update primitives and determining the consequences of these operations has several advantages. During class design, this approach helps developers detect the implications of their actions on the class collection and maintain consistency within 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 structured editor where each editing operation corresponds to a schema manipulation primitive like those of ORION or GemStone [32]. Such an environment would nevertheless fall short of providing 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.

8.4 Class Versioning

8.4.1 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 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. The incompatibilities between variants of the same class hierarchy were sufficient to hinder the further exchange of software, or at least to severely reduce its portability.

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 software developers may correct their 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.

Because *adhoc* techniques do not scale well for large, distributed programming environments, current approaches favour a 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:

- How is version management organized with respect to software development?
- 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?

8.4.2 The Organization of Version Management

An environment for version management is divided into several distinct working spaces, each one providing a specific set of privileges and capabilities for manipulating different kinds of versions [15][24]. Three such domains are generally recognized in the literature:

- A private working space supports the development activities of one programmer. The information stored in the programmer's private environment, in particular the software components he or she is currently designing or modifying, is not accessible to other users.
- All classes and data produced during a project are stored in a corresponding domain that 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 domain 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 domain. 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 domains and possibly private domains. They are considered stable and cannot be modified, but they can be deleted by their owner, i.e. the project administrator or the user of a private domain. 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 any other kind of version. It belongs to the user who created it and it is stored in his or her private domain. Transient versions can be updated, deleted and promoted to working versions.

The principal characteristics of version types are summarized in table 8.3.

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 the domain as addi-

Characteristics of version types	Transient	Working	Released
Location			
public domain			✓
project domain		✓	
private domain	✓	✓	
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 8.3 *Principal characteristics of version types. Some systems consider only two kinds of versions (transient and released) and two levels of domains (private and public) for managing their visibility.*

tional transient versions. In order to try different designs for the same class, or to save the result of the programming activity, programmers may derive new transient versions from those they is currently working on, while simultaneously promoting the latter to working versions. When a programmer achieves a satisfactory design for a software component, he or she installs it as a working version in the project domain. Of course, these working versions can subsequently be copied by 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 for 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 and can use simpler checkin/checkout or optimistic locking techniques.

8.4.3 Version Identification

Class identity is an essential problem to deal with. 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 domain name, must be provided to identify a component unambiguously [24]. 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 good solution to bind version references in interactive queries in object-oriented databases.
- Its most recent version at the time the component which made the reference was created. This is the preferred option for dealing with dynamic references in class definitions.
- A default class definition specified by the administrator in charge of the domain. 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 domain where the reference is initially discovered to be unresolved; the hierarchy of domains is then inspected upward until the appropriate definition is found. Thus, to bind an incomplete reference to a class made in a project domain (i.e. a reference consisting only in the class name, without additional information), the system first examines the class hierarchy in the current domain; if this domain does not contain the class definition referred to, the search proceeds in the public repository. No private domain is inspected, for stable versions are not allowed to refer to transient versions that could be in the process of being revised. Similarly, dynamic references to classes in the public domain cannot be resolved by looking for unreleased components in a project domain. Naturally, dynamic binding can be resolved at the level of a private domain 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 herarchy. When a version is copied or installed in a domain, 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 acyclic graph [8].

The information on derivation dependencies is generally associated with the generic version of a class version set. Version management systems like IRIS and AVANCE implement a series of primitives for traversing and manipulating derivation graphs [5][8]. 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 a significant storage and processing overhead. This is why in most systems one is required to explicitly indicate that classes are versionable by making them subclasses of a special class from which they inherit their properties of versions — that is often called Version, as in AVANCE and IRIS.

8.4.4 Versioning and Class Evolution

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 should be released. Such actions must be based on design knowledge that is best mastered by the software developers themselves. Thus, the automatic generation of new versions triggered 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 [8].

- 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 delegation (i.e. classes containing 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 method implementations 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 re-establishing 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 [15]. Thus, a class should never have a "last modification" date that exceeds the "approved modification" date of the versions referring to it. 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 program-

mer 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.

Building consistent configurations of classes and instances, and 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 prohibit objects from older versions from referring to instances of future variants. We describe in more detail how to achieve these effects in the section devoted to update propagation.

8.4.5 Evaluation

Versioning is an appealing approach for managing class development and evolution. Recording the history of class modifications during the design process has 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 (e.g. 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 [38].

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, methods for version management still lack some support for design tasks: at what point does a version stop being 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 into several object-oriented environments, including Orwell [38], AVANCE [7], ORION [3] and IRIS [19].

8.5 Class Reorganization

8.5.1 Issues

The lessons drawn from the construction of collections of reusable classes have led to the formulation of some principles that serve to improve object-oriented libraries [22].

The first principle is to make sure that components are really polymorphic. This can be achieved in a number of ways:

- Adopt a uniform terminology for related classes and standardize the methods making up their interface [31].
- Eliminate code that explicitly checks the type of an object. Rather than introducing 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.
- Decrease the number of arguments in a method, either by splitting the method into several simpler procedures, or by creating a class to represent a group of arguments that often appear 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.

A second set of rules aims to increase the degree of abstraction and generality of classes:

- Factorize 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 to reduce the dependency of methods on the internal class representation [29]. This can be achieved by resorting to special accessors instead of referring directly to variables.
- Ensure that inheritance links express clear semantic relationships such as specialization, or even better, relationships with known mathematical properties like conformance or imitation [40].

Finally, reorganizations should improve the modularization of functionality in a library:

- Split large classes into smaller, cohesive classes that are more resilient to change.
- Separate groups of methods that do not interact. Such sets of methods represent either totally independent behaviour or different views of the same object, which are perhaps better represented by distinct classes.
- Uncouple methods from global attributes or internal class properties by sending messages to parameters instead of to self or to instance variables.

These guidelines are very general; the problem is therefore to formulate these empirical rules rigorously and to make them amenable to a subsequent automation.

8.5.2 Refactoring

8.5.2.1 Issues and Techniques

Refactoring is an approach that extends basic class surgery primitives with advanced redesign mechanisms [23]. Refactoring is based on an object model that is specifically tailored to represent and manipulate the rich structure of components developed with an object-oriented programming language. The schema invariants of class surgery are extended with additional constraints for preserving behaviour, and the preconditions for modification operations are made more precise or more restrictive to avoid introducing behaviour and referential inconsistencies in a class collection. The approach proposed in [34] is intended to support refactoring specifically for C++ libraries. Four important operations are discussed in detail.

- Distributing the functionality of a class over multiple subclasses by splitting methods along conditional statements. Let us consider a hypothetical class that checks the rights of users to access a system during weekends and normal working days:

```
class ACCESS-CONTROL
    methods
        CheckPrivileges
            begin
                -- some general code …
                if date = Sunday or date = Saturday then
                    -- restricted access on week-ends …
                else
                    -- usual checks during normal working days …
                end-if
            end CheckPrivileges; …
    end;
```

  ACCESS-CONTROL is specialized in as many classes as there are branches in its CheckPrivileges method; CheckPrivileges is itself decomposed so that, in each subclass, it contains only the code corresponding to one branch of the original conditional statement. The common part of all CheckPrivileges variants is left in ACCESS-CONTROL.

```
class ACCESS-CONTROL
    methods
        CheckPrivileges
            begin
                -- some general code …
            end CheckPrivileges; …
    end;

class CONTROL-WEEK-END
    inherit ACCESS-CONTROL;
    methods
        CheckPrivileges
            begin
                super.CheckPrivileges;
```

```
                        -- restricted access on week-ends …
                end CheckPrivileges; …
        end;
    class CONTROL-WORKING-DAYS
        inherit ACCESS-CONTROL;
        methods
            CheckPrivileges
                begin
                    super.CheckPrivileges;
                    -- usual checks during normal working days …
                end CheckPrivileges;
        end;
```

- Creating an abstract superclass. This operation analyses two classes, extracts their common properties, which are placed in a new component, and then makes both initial classes subclasses of the new definition. The extraction of similarities between two classes is not performed automatically and relies on heuristics to detect common structures in method signatures and implementations. Additional renaming of variables and methods, reordering of method parameters and transformations of method implementations may be carried out to achieve a satisfactory result. However, contrary to the incremental reorganization algorithm described in section 8.5.4.3, refactoring does not propagate through the inheritance graph.
- Transforming an inheritance relation into a part-of relation. The following example shows a class SYMBOL-TABLE that inherits functionality from HASH-TABLE.

```
    class HASH-TABLE
        methods
            Insert …
            Delete …
        end;
    class SYMBOL-TABLE
        inherit HASH-TABLE; …
        end;
```

Rather than being a subclass of HASH-TABLE, SYMBOL-TABLE can refer to an instance of HASH-TABLE via a part-of relation. This requires severing the inheritance link between both classes, introducing a variable of type HASH-TABLE in SYMBOL-TABLE, and adding a series of procedures in SYMBOL-TABLE for delegating the invocations of methods previously inherited from HASH-TABLE to this new variable. In our simplified example, the refactoring does not change the superclass. In general, it may be necessary to introduce special operations in the superclass to encapsulate accesses to its variables, and to change the methods declared in the subclass so that they manipulate these variables through these operations.

```
    class SYMBOL-TABLE
        variables
            store          :     HASH-TABLE; …
        methods
            Insert (…)
```

```
                begin
                    store.Insert (...);
                end Insert;
            Delete (...)
                begin
                    store.Delete (...);
                end Delete; ...
        end;
```

- Reshuffling attributes among classes. This operation is intended to improve the design of classes representing aggregations, where a component of an aggregation can only belong to or be referred to by one object. Redistributing variables denoting aggregation elements in a behaviour-preserving way is feasible only when several strong conditions on referencing patterns are satisfied. References to the migrated variables are updated or replaced with invocations to appropriate accessors.

8.5.2.2 Evaluation

Refactoring is one of the most interesting approaches for providing software developers with high-level, intuitive operations supporting complex redesign activities. Refactoring embodies some of the empirical guidelines derived from actual experience with class evolution; it would therefore be appealing to integrate such a toolkit of operations in an editing and browsing environment. This approach is not without limitations though; the decision to carry out specific refactorings, the optimization goals and the selection of the classes to modify are left entirely up to the programmer. Thus, refactoring exhibits the same shortcomings as class surgery. The automatic approaches discussed in the following sections are based on systematic strategies that are probably more adequate in the context of large, complex libraries. As with any other restructuring method, refactoring faces intractability problems when trying to achieve all possible transformations or to preserve behaviour. For example, all interesting situations where a method could be split among subclasses cannot be detected, and, in fact, the conditional expressions considered are only of a very elementary nature.

8.5.3 Restructuring Interattribute Dependencies

8.5.3.1 Issues

Avoiding unnecessary coupling between classes and reducing interattribute dependencies are two important prerequisites for well-designed objects. Two major issues have to be addressed:

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

A possible solution to this problem has been proposed by Lieberherr *et al.* [29] under the name of "Law of Demeter", together with a small set of techniques for mechanically transforming object definitions so that they comply with this law [12].

8.5.3.2 The Law of Demeter

The Law of Demeter distinguishes three types of interattribute 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 if 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 if 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.

The "class form" of the law states that methods may only access entities belonging to their preferred-supplier classes. The "object form" of the law does not consider the classes a method depends on, but rather the objects this method sends messages to. In this context, a preferred-supplier object is an instance that is either a variable introduced by the class where the method is defined, or an argument passed to the method, or an object created by the method, or the pseudo-variable self (identifying the object executing the method). The "object form" of the law prohibits references to instances that are not preferred-suppliers of a method. In its *weak* version, the law considers the classes of inherited variables (or the variables themselves, in the "object form" of the law) as legitimate preferred-suppliers. The *strict* version does not consider the classes of inherited variables (or inherited variables) as legitimate preferred-suppliers.

8.5.3.3 Application and Examples

We illustrate the main reorganization aspects dealt with by the Demeter approach for a group of simple object descriptions [12][29]. Let us consider the following partial class definitions:

```
class LIBRARY
    variables
        Catalog             :    CATALOG; …
    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
        Optical-Disk        :    CD-ROM;
```

```
            Microfiches            :    MICROFICHE; …
        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] …
        end;
    class BOOK
        variables
            Title                           :    STRING; …
        end;
```

These definitions obviously do not conform to the law: the method Search-book in LIBRARY accesses internal components of Catalog (the attributes Microfiches and Optical-Disk); it sends messages to these variables and receives as a result objects that are neither components of LIBRARY nor instances of a preferred-supplier class of LIBRARY. We also note that the algorithm for retrieving all references stored on the optical disk 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 makes the querying methods of LIBRARY immune to alterations in the internal structure of the catalogue — for example the replacement of the microfiches with an additional CD-ROM file. In doing so, we have to take care that LIST [BOOK], the type of the result of methods Search-book in MICROFICHE and CD-ROM, is not a preferred-supplier of CATALOG. The introduction of the auxiliary method Merge-refs in CATALOG solves this problem and makes the dependency between classes CATALOG and LIST [BOOK] explicit. Finally, ensuring the proper encapsulation of

BOOK objects requires that their variables be manipulated through special-purpose accessors; CD-ROM is adjusted accordingly.

```
class LIBRARY
    variables
        Catalog             :    CATALOG; …
    methods
        Search-book (title : STRING) returns LIST [BOOK]
            begin
                return (Catalog.Search-book (title));
            end Search-book; …
    end;

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;

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; …
    methods
        Get-Title returns STRING
            begin
                return (Title);
            end Get-Title; …
    end;
```

8.5.3.4 Evaluation

The Law of Demeter nicely captures some issues dealing with encapsulation and coupling; although a fully formal model that would mathematically justify its underlying assumptions is still lacking [36], its application to the design of modular class libraries has been found to be beneficial [29]. However, putting the Law of Demeter into practice raises several difficulties [36]. It cannot be completely enforced with languages, such as CLOS or Smalltalk, that allow expressions to be constructed dynamically and then executed at run-time. In general, the "class form" of the law does not seem to be fully effective for untyped languages; since objects are untyped, violations of the law cannot be discovered by a static inspection of the source code, but must be monitored during program execution.

As far as typed languages are concerned, applying the Demeter principles is not always straightforward either. First, there are some special cases where the spirit of the Law of Demeter is violated, although all the dependencies formally respect all the Demeter rules stated in section 8.5.3.2. 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; for example, translating the law of Demeter into equivalent terms for C++ is far from trivial, because of the hybrid model of this language and the need to take constructs like friend functions into account.

8.5.4 Restructuring Inheritance Hierarchies

8.5.4.1 Issues

A frequent problem during the design of inheritance hierarchies is that programmers overlook intermediate abstractions needed for establishing clean subclassing dependencies, and develop components too specialized to be effectively reusable. Several approaches have been proposed to automate the detection and correction of such defects in inheritance hierarchies. They are distinguished by the way they address a few fundamental issues:

- What is the scope of the reorganization applied to an inheritance graph?
- What are the criteria driving the reorganization?
- What properties are preserved across reorganizations?

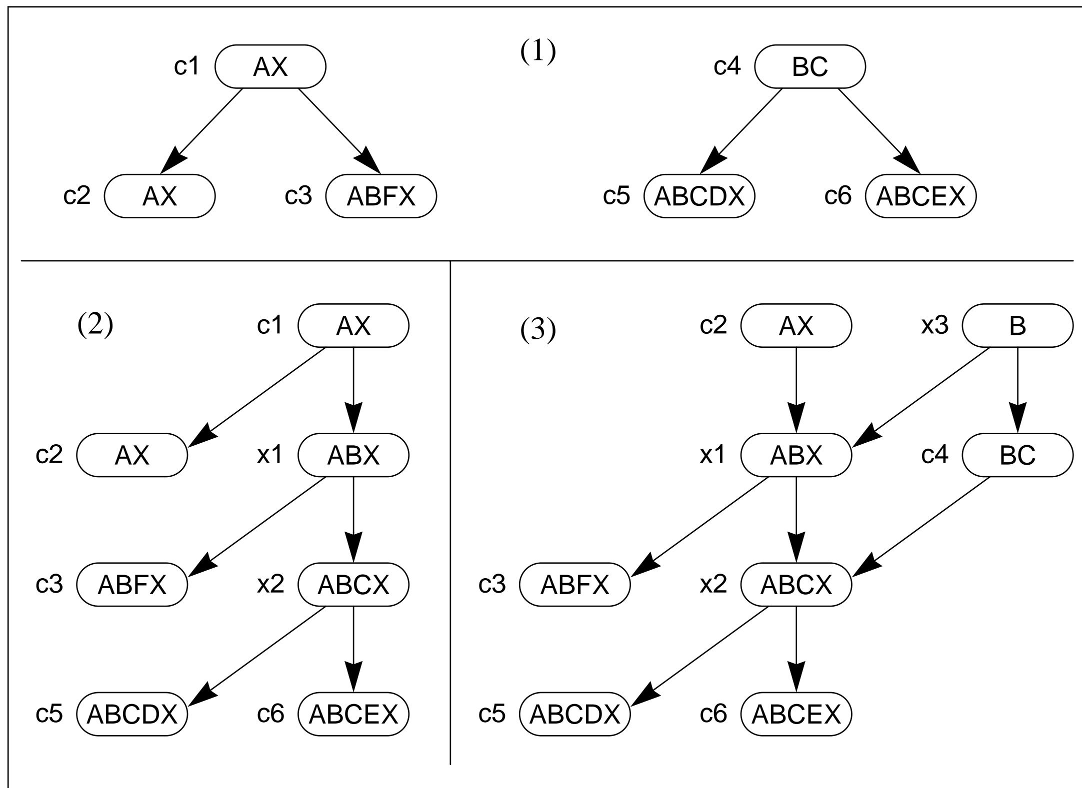

Figure 8.1 *Reorganizing a redundant, non-connected hierarchy (1); capital letters represent attributes. (2): after applying the Demeter algorithm; (3): after applying the algorithm described in [12]. Class c2 is the concrete counterpart of abstract class c1; similar definitions are merged in (3), but not in (2). Class c4 is preserved in (3), but considered as superfluous in (2).*

The differences between inheritance reorganization methods are best summarized by grouping these approaches into global and incremental reorganization techniques.

8.5.4.2 Global Reorganization

Global reorganization approaches produce optimal inheritance graphs, without attribute redundancy and with a minimum number of classes and inheritance links, from pre-existing hierarchies (figure 8.1). These techniques can be fully automated. They work globally, analyzing and recasting an entire class collection at a time.

The approach proposed in the context of the Demeter project is based on a formalism that distinguishes between abstract classes, which can be inherited but not instantiated, and concrete classes, which can be instantiated but cannot be used as superclasses. Classes correspond to the vertices in a graph. The edges of the graph denote either inheritance relationships between classes, or part-of relationships between classes and their (typed) attributes [28]. This model forms the basis for global reorganization algorithms whose goal

is to optimize the structural characteristics of an inheritance graph (i.e. to minimize the number of classes and relations in a library). Redefinitions and attribute structures are not taken into account. The formal properties of these algorithms have been investigated in detail [28]:

- Transforming a hierarchy to suppress redundant part-of edges, i.e. forcing classes to inherit common attributes from a shared superclass, is in *P*.
- Minimizing the overall number of edges is *NP*-complete. When the final hierarchy is actually a tree, efficient (polynomial) algorithms exist for optimizing the hierarchy.

A different method is based on an object model that allows classes to inherit from concrete superclasses [12]. The corresponding algorithm proceeds by flattening all class definitions present in a hierarchy, then factoring out common structures, relinking all class definitions through inheritance, and finally eliminating redundant inheritance links and auxiliary class definitions. Contrary to the Demeter approach, this algorithm does preserve all definitions that actually differ in the library before the reorganization, it takes redefinitions into account and it can be tailored to avoid repeated inheritance in the final hierarchy.

None of the global algorithms deal with interattribute dependencies or with the preservation of behavioural properties. Global algorithms do not always produce identical results because of their varying assumptions and goals — as is shown clearly in figure 8.1.

8.5.4.3 Incremental Reorganization

Adding a subclass is a major step in the development of an object-oriented library, warranting an evaluation, and possibly an improvement of the hierarchy. The evaluation can be restricted to the relationships between the new class and its superclasses, and the reorganization can be limited to the location where the new class is introduced. The incremental factorization algorithm proposed in [11] is driven by the analysis of redefinition patterns between a new class and its superclasses. It attempts to optimize the inheritance graph within reason while keeping the disturbances to the original library to a minimum. Behavioural properties can be maintained to a certain extent and classes present in the hierarchy before the reorganization are not deleted [12]. The algorithm transforms a hierarchy automatically to eliminate unwanted subclassing patterns, to pinpoint places requiring redesign and to discover missing abstractions. It can take into account renaming and structural transformations similar to those discussed in 8.5.2.

The incremental reorganization algorithm extracts the properties shared by several classes and isolates them in a new, common superclass. Figure 8.2 shows a fragment of the Eiffel library where class CIRCLE inherits from ELLIPSE. This subclassing operation is accompanied by a partial replacement of ELLIPSE's behaviour. Simultaneously, CIRCLE changes its superclass's interface in a way that corresponds neither to a restriction (which would be expected in a specialization relationship) nor to an extension (characteristic of subtyping relationships). A transformation of the hierarchy eliminates this unnatural subclassing pattern by inserting an intermediate definition containing the properties common

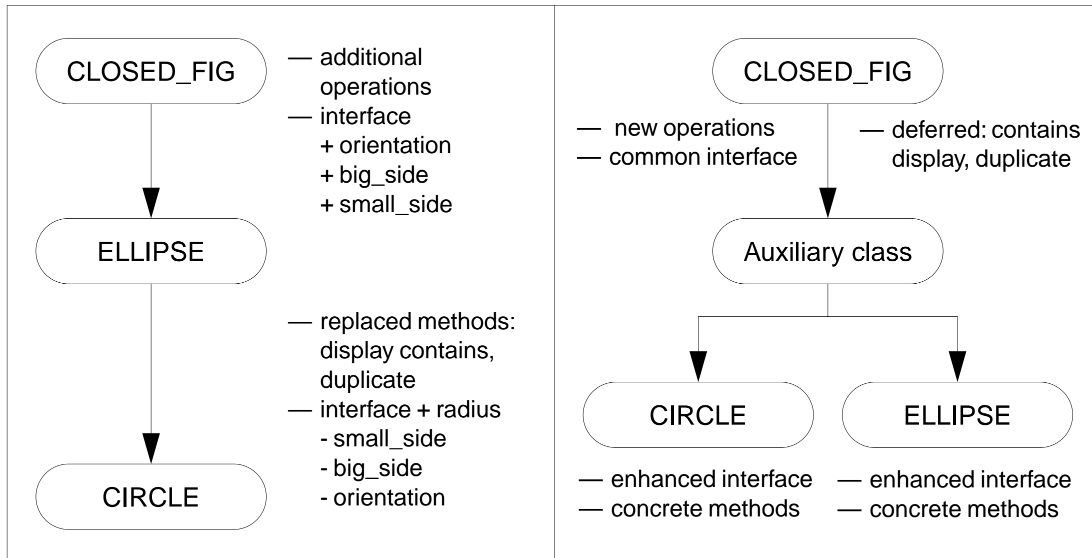

Figure 8.2 *Factorizing inheritance relationships.*

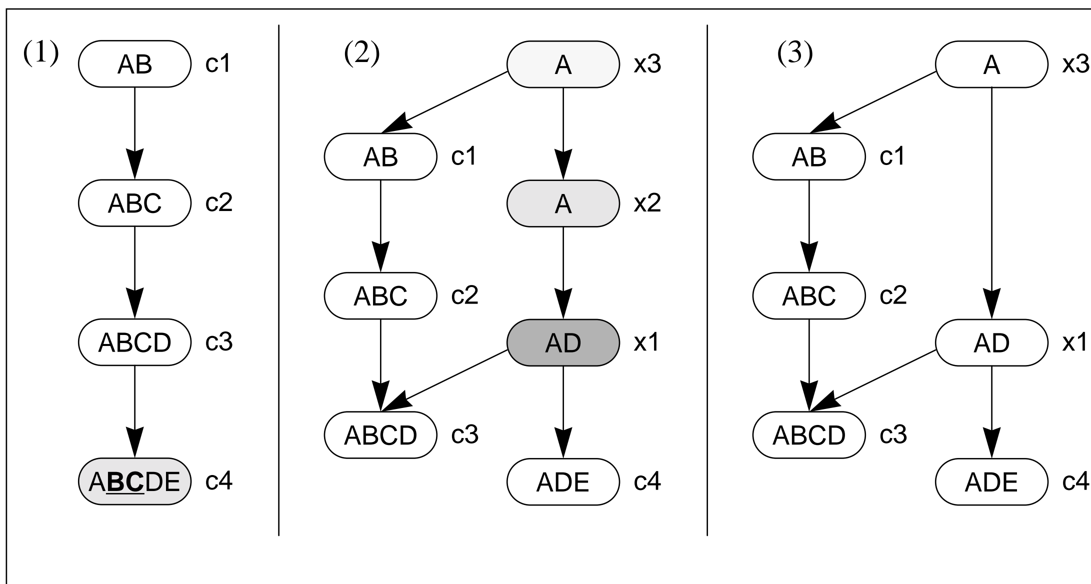

Figure 8.3 *The new class c4 rejects attributes B and C from c3; this triggers an incremental reorganization of the hierarchy whose final result is depicted in (3).*

to both CIRCLE and ELLIPSE, and by making these two classes subclasses of the new auxiliary node.

In more complex situations, the factorization propagates as high up in a hierarchy as is needed to eliminate unwanted subclassing patterns and introduces auxiliary definitions

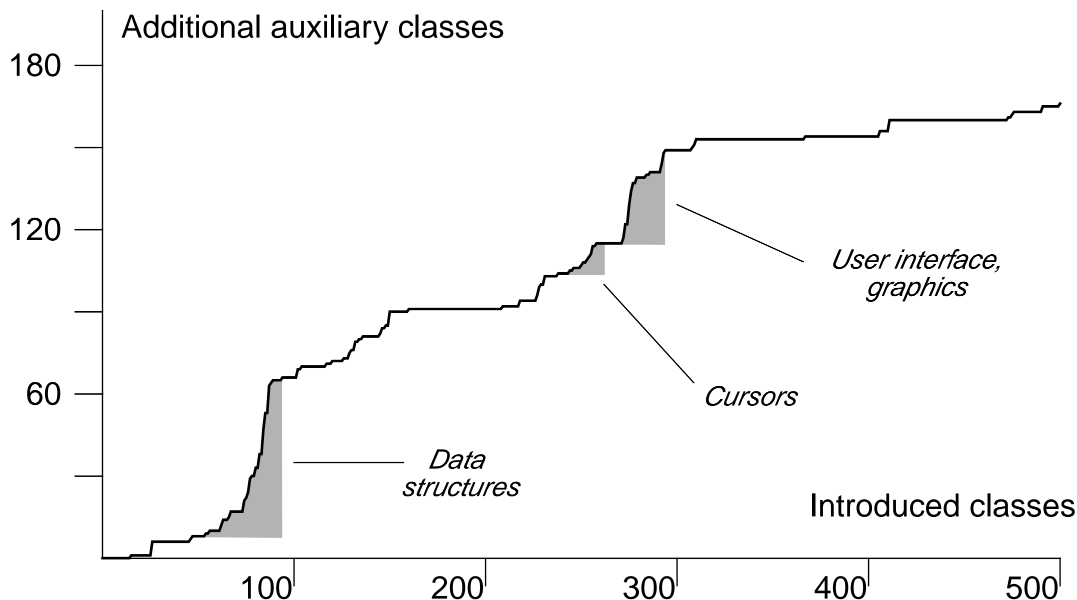

Figure 8.4 *Restructuring the Eiffel 2.3 library. A few groups of classes responsible for clustered reorganizations are highlighted. Overall, the incremental factorization of Eiffel 2.3 adds 166 auxiliary definitions to the library.*

along the way. A last simplification phase suppresses redundant auxiliary nodes and links (figure 8.3).

8.5.4.4 Application of Incremental Reorganization

The incremental reorganization algorithm of [11] is one of the rare approaches whose effectiveness has been quantitatively assessed on the basis of large-scale experiments involving the reorganization of versions 2.1 and 2.3 of the Eiffel library (98 and 500 classes respectively). Starting from an empty hierarchy, Eiffel classes were added one by one to the library, triggering incremental reorganizations whenever redefinition patterns amounting to the rejection of inherited methods were detected (figure 8.4). This study brought to light several interesting results [13]:

- A large majority (63%) of the problems uncovered by the reorganization algorithm were caused by the utilization of inheritance for code sharing and by an inadequate modularization of functionality leading to other improper subclassing relationships.
- In 21% of the cases, the outcome of the reorganization corresponds to what one would expect from a manual redesign of the library. The restructuring patterns of the incremental algorithm closely match empirical observations on the evolution of object-oriented libraries [2], as well as small-scale reorganizations of a limited subset of the Smalltalk hierarchy [17].

- In 33% of the cases, the incremental algorithm detects, but is not able to correct, many actual design problems in a library that are best solved by other kinds of reorganizations, such as transforming inheritance links into part-of relationships.
- The algorithm is also useful for evaluating and comparing the quality of object-oriented libraries, especially when it is combined with other incremental techniques that are sensitive to naming patterns [13]

8.5.4.5 Evaluation

Global reorganizations are a prerequisite when the goal is to put a hierarchy into a "normal form" free from redundancy. However, global revisions may thoroughly transform a library. The results are therefore difficult to grasp and to utilize, particularly with libraries comprising hundreds of classes. Incremental factorization, on the other hand, limits its scope to the inheritance paths leading to one new class — an approach that also guarantees better performance in an interactive environment. Besides, it is doubtful that a global reorganization can achieve significant results without additional processing to extract the structural similarities between class interfaces or method signatures that are hidden because of diverging naming and programming conventions [31][34]. Maintaining behavioural properties is a problem with both global and incremental reorganizations [6][12][39] and, anyway, many design problems cannot be solved through adjustments of subclassing relationships alone. Inheritance reorganization techniques must therefore be enhanced with other methods such as refactoring to support redesign activities effectively. Automatic approaches are nevertheless essential to reduce the search space for redesign operations on large libraries to a manageable size before applying interactive, user-driven surgery or refactoring operations.

8.6 Change Avoidance

8.6.1 Confining the Effects of Evolution

In principle, modifications of class specifications must be propagated to objects instantiated on the basis of old definitions, so as to maintain the overall consistency of the system. Nevertheless, in many cases instances need not be updated or enhanced when their class is modified. Detecting when these situations arise is important, since one can then avoid the inconvenience of change propagation without giving up system consistency.

Change avoidance is easily combined with class tailoring. Tailoring operations are carried out only for the purpose of defining additional subclasses; no matter how inherited properties are overridden, the modifications appear and take effect only at the level of the subclasses performing the redeclarations. New classes obviously have no associated instances, so there is no need to care about filtering or conversion procedures. Thus, object-oriented systems avoid updating instances when subclassing operations are considered.

Several other evolution primitives exhibit no side-effects and can safely be applied without reorganizing running applications. Among the surgery operations listed in section 8.3.3, the following have no consequences on object structures:

- Renaming classes, methods and variables only affects the description of classes, not the structure of instances, although this may not always be true for programs that explicitly manipulate class or attribute names.
- 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 instances.
- 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.

8.6.2 Physical Structures

A technique for confining the effects of class evolution consists of uncoupling the logical object model from its physical representation, so that instances may be implemented in a way immune to change. Transposed files exhibit such desirable characteristics [18].

In traditional database systems, the state of an object (i.e. the set of all its variables) is usually stored in one record (methods are shared and stored in a separate area). Every class of a hierarchy is associated with a file which is used as a persistent storage space for its entities, with each record of a file containing the state of a particular entity (figure 8.5). When a variable is added to a class definition, additional space must be allocated for the corresponding class and its subclasses; the instances affected by the modification are subsequently copied into the new storage zones. When a variable is suppressed from a class, special procedures are required for reclaiming unused storage space, a process that generally entails unloading and reloading entire class extents.

Transposed files associate one file with each variable of a class. Each record contains the value of the variable for a particular instance. The complete representation of a class is thus spread among several files. One reconstitutes the state of an object by first accessing the values of its various variables in their respective files, and then grouping them together in the main memory for processing. All values for the variables of an object are stored in records located at the same rank in the various files; this is made possible by deriving this rank directly from the identifier assigned to every object in the system. A simple scheme is to use a pair ⟨class-identifier, rank⟩ to identify objects. Because file management systems generally allocate disk space not by records but by blocks, a level of indirection is needed to access the value of a variable. On the other hand, such a structure facilitates the insertion of objects whose identifiers are not strictly sequentially determined (blocks corresponding to unused identifiers need not be reserved), and the release of space after the last object

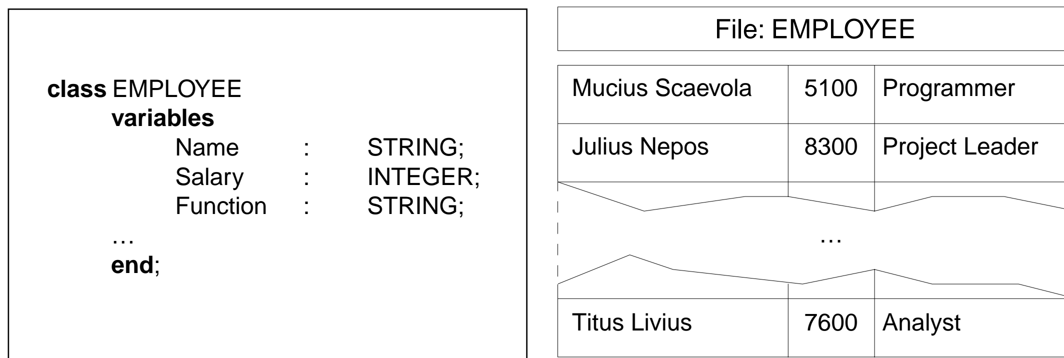

Figure 8.5 *Traditional storage technique for a hypothetical* EMPLOYEE *class.*

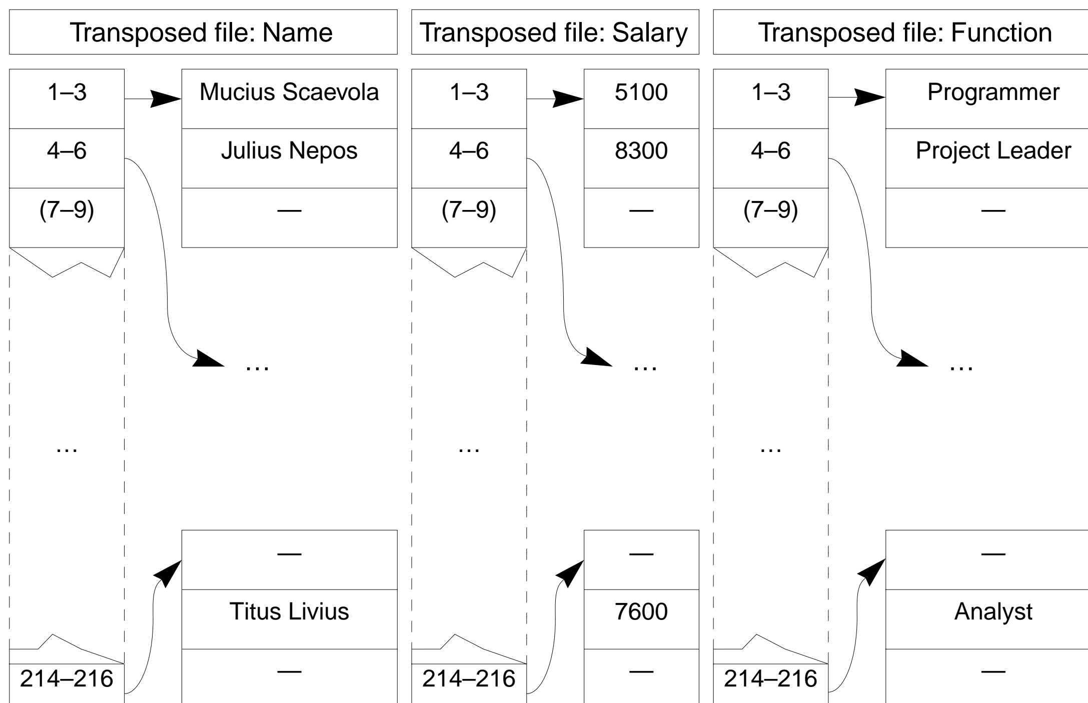

Figure 8.6 *Using a transposed file organization for storing class* EMPLOYEE*. Rank 2 contains all information relative to employee "Julius Nepos", rank 215 the data relative to "Titus Livius". No instance corresponds to ranks 7–9, so the corresponding block is not allocated.*

associated with a particular block is deleted. Resource waste is therefore reduced. The diagram of figure 8.6 represents the simplified structure of a transposed file.

Transposed files provide an efficient kernel for implementing many of the class surgery primitives described in section 8.3.3, for example:

- Adding a variable to a class does not require reformatting the existing records to make room for the new attribute. Instead, an additional file is reserved to contain the supplementary variable that is initialized to some default value, such as nil or 0, for existing instances.
- Suppressing a variable is achieved by deleting the corresponding file and returning all the space it occupies to the system. No compaction of the database is required.
- A subclass definition comprises all its superclass files plus some additional files. If a reorganization of the hierarchy results in the destruction of the subclass, all files for the attributes it introduces are deleted, but not those corresponding to the variables of its superclass. All instances of the subclass automatically become members of the superclass, without one having to execute any procedure to save, reformat and transfer the objects from one class to the other. The class-identifier part of all object identifiers must nevertheless be updated to remain consistent across changes.

Transposed files have proved very useful in domains such as statistical and econometric information systems. They have therefore been implemented in special-purpose database systems geared towards supporting these categories of applications. Their application in semantic and object-oriented database systems is currently a field of active research [18].

8.7 Conversion

8.7.1 Issues

Transforming all entities whose class has been modified seems like the most natural approach to dealing with change propagation. 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 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 evidently limited. The second requirement forces all conversion procedures to behave as atomic transactions (transformations must be applied completely to the objects involved in the conversion) and puts strong restrictions on their duration.

8.7.2 Instance Transformation

CLOS provides a good example of how automatic conversion can be enhanced by the programmer to take supplementary integrity constraints into account [25]. Conversions are performed according to the rules listed in table 8.4. CLOS deletes from objects all at-

Old slot	New slot		
	shared	local	none
shared	preserved	preserved	discarded
local	initialized	preserved	discarded
none	initialized	initialized	—

Table 8.4 *Default conversions carried out by CLOS on objects after a class modification. A slot corresponds to a variable. Preserved slot values are 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. This table is reproduced from [25].*

tributes that have been deleted in their class, including their associated accessor methods; it 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). These conversions are carried out by a standard function called update-instance-for-redefined-class that is inherited by every class in a hierarchy and can be customized by the programmer. Arguments such as the list of attributes added to the class, or the list of attributes discarded from the class or converted from local to shared, with their original values, are passed to this function. This allows the programmer to take proper actions to correct and augment the default restructuring and reinitialization procedures provided by CLOS, and thus 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 through a table-driven interface rather than as a programming feature attached to the inheritance hierarchy [27]. A table lists all class definitions whose instances have to be converted and suggests default transformations that apply, which can of course be overridden or extended by the user. The transformation operations possible with OTGen are as follows:

- Transfer objects which belong to the old class definition to the new database. Unchanged objects are simply copied from a database to another.
- Delete objects from the database if their class has been deleted.
- Initialize the variables of an object. When the old and new types of a variable are incompatible, the default action taken by OTGen consists of assigning the nil value to

the variable. The user can override the standard behaviour of the system by providing its own initial values.

- Change local variables to shared variables.
- Perform context-dependent changes. One may initialize variables based on previous information stored in the objects, or partition 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.

Providing a framework to handle the most common transformations certainly eases the task of the programmer. It is difficult, however, to guarantee that such a predetermined 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 routines.

8.7.3 Immediate and Delayed Conversion

A major 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 have to be converted. 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 [35].

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 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. Lazy conversion 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 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. This technique has been proposed as the standard mechanism for CLOS. A version of the O_2 system implements both techniques [41], applying immediate conversion to instances present in main memory at the time of the modification and resorting to lazy conversion for objects residing in secondary storage [4].

8.7.4 Evaluation

Conversion, and in particular lazy conversion, is 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 are 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.

8.8 Filtering

8.8.1 Issues

Under some circumstances, one may not need to physically 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 an application may still use them, but without striving to make them perfectly interchangeable.

Filtering (or screening) is a general framework for dealing with this problem. It is most often used in combination with version management. This can be done 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 pop up 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 class versions?
- How can one map instances from a class version to another?
- How far can a filtering mechanism hide class changes from the users?

8.8.2 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 [1]:

- 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 the 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. 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 [1]:

- 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.

8.8.3 Filtering Mechanisms

The operations that cause problems when invoked on a non-compatible object can be classified in a limited number of categories. For example, deleting a method generates access violations when an object attempts to invoke the deleted method. These effects are summarized in table 8.5.

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 procedure that returns the variable's value, and another procedure for changing its value. These procedures perform various transformations, like mapping the variable to a set of other attributes [1]. 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 the 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. 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

Scope of change	Compatibility	Consequences
add a variable	backwards	undefined variable in old objects
delete a variable	forwards	undefined variable in new objects
extend variable type	backwards	writing illegal values into old objects
	forwards	reading unknown data from new objects
restrict variable 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 8.5 *Consequences of class changes. 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.*

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.

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 that has been implemented in the ENCORE system [37]. Pre-handlers typically take over when attempting to access a non-existent attribute, or when trying to assign an illegal value to 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 is activated when an illegal value is returned to the invoking object; a common behaviour in this case consists in returning a default value.

8.8.4 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 [37]. 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 of a class.
- When a version extends the domain of an attribute, corresponding pre- and post-handlers must be introduced in all other versions of the class.
- 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 rather inelegant: it requires that old class definitions be adjusted to reflect new developments and leads to a combinatorial explosion of handler complexity.

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 only 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 more recent versions.

Scope restrictions and compatibility relationships make it possible to partition a class extension 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.

8.8.5 Evaluation

Screening has been implemented in some systems, but its application scope there is notably reduced. ORION does not immediately convert instances affected by a class change so as to avoid reorganizing the database [3]. 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 new inherited attributes.

From our discussion, it appears that filtering cannot 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, and it is most adequate during prototyping, when class modifications may be cancelled just after being tested. Recent approaches provide improved mechanisms derived from database views that encompass filtering techniques and that can also be suitable as modelling tools during application development [16].

8.9 Conclusion

Object-oriented development reveals its iterative nature as successive stages of subclassing, class modification and reorganization allow software engineers to build increasingly general and robust classes. We therefore expect object-oriented CASE systems to take advantage of the large spectrum of tools and techniques available to manage the various aspects of class evolution (see table 8.6).

Approach	Actual impact on instances	In charge of controlling change propagation	Implementation
change avoidance			
confinement	logical	system	side-effect free operations
storage structures	physical	system	transposed files
conversion	physical	programmer	conversion routines
filtering	logical	programmer	handlers/wrappers

Table 8.6 *The main characteristics of change propagation techniques.*

It is appealing to envision an environment where software engineers build new classes out of reusable components, tailor them to suit their needs, and launch exploratory incremental reorganizations to detect the places in their code most likely to require further re-

Approach	Scope	Phase in library development	Enforced properties
tailoring	attributes; interfaces	extension	syntactical constraints
surgery	attributes; inheritance links; classes	redesign	schema invariants
versioning	classes	extension	configuration consistency
reorganization			
refactoring	classes; attributes; method structures; inheritance links	redesign	schema invariants; preservation of behaviour
interattribute dependencies	method structures	redesign	preservation of behaviour
inheritance (global)	classes; inheritance links	redesign	preservation of class structures; global optimality of hierarchy
inheritance (incremental)	classes; inheritance links; interfaces; method structures	extension	preservation of class structures; local optimality of hierarchy; preservation of behaviour

Table 8.7 *The main characteristics of evolution management techniques. Attributes refer to methods as well as to variables; method structures correspond to the signature and the implementation of methods.*

visions. Software developers may then refine the outcome of automatic reorganizations with class surgery primitives and perhaps embark on comprehensive refactoring activities. The results of different reorganizations and their subsequent adjustments are kept as versions of the hierarchy, that can be further modified, tested, debugged and possibly cancelled by the programmers (see table 8.7). Filtering makes it possible to test the correctness of various class definitions without having to carry out numerous conversions. When a satisfactory design for a new component and its related classes is achieved, it can be frozen and publicly released as the new version of the class library, while the other temporary versions are discarded. If necessary, instances from modified classes can then be definitely converted to conform to their new definitions.

Some approaches have been partially implemented and already appear, albeit in isolation, in some object-oriented systems; we hope that integrated tools suitable for supporting class evolution in industrial and commercial environments will become available in the near future.

References

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

[2] Bruce Anderson and Sanjiv Gossain, “Hierarchy Evolution and the Software Lifecycle,” in *Proceedings 2nd TOOLS Conference*, ed. J. Bézivin, B. Meyer and J.-M. Nerson, Paris, 1990, pp. 41–50.

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

[4] Gilles Barbedette, “Schema Modification in the $LISPO_2$ Persistent Object-Oriented Language,” in *Proceedings 5th ECOOP Conference*, ed. P. America, *Lecture Notes in Computer Science*, vol. 512, Springer-Verlag, Geneva, 15–19 July 1991, pp. 77–96.

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

[6] Paul L. Bergstein and Walter L. Hürsch, “Maintaining Behavioral Consistency during Schema Evolution,” in *Object Technologies for Advanced Software (First JSSST International Symposium)*, *Lecture Notes in Computer Science*, vol. 742, Springer-Verlag, Nov. 1993, pp. 176–193.

[7] Anders Björnerstedt and Stefan Britts, “AVANCE: An Object Management System,” *ACM SIGPLAN Notices* (special issue on OOPSLA ’88), vol. 23, no. 11, Nov. 1988, pp. 206–221.

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

[9] Alexander Borgida, “Modelling Class Hierarchies with Contradictions,” *SIGMOD Record* (special issue on SIGMOD ’88), vol. 17, no. 3, Sept. 1988, pp. 434–443.

[10] Alexander Borgida and Keith E. Williamson, “Accommodating Exceptions in Databases, and Refining the Schema by Learning from them,” in *VLDB 1985 Proceedings*, ed. A. Pirotte and Y. Vassiliou, Stockholm, 21–23 August 1985, pp. 72–81.

[11] Eduardo Casais, “An Incremental Class Reorganization Approach,” in *Proceedings 6th ECOOP Conference*, ed. O. Lehrmann Madsen, *Lecture Notes in Computer Science*, vol. 615, Springer-Verlag, Utrecht, June 29 – July 3 1992, pp. 114–132.

[12] Eduardo Casais, “Managing Evolution in Object-Oriented Environments: An Algorithmic Approach,” Ph.D. Thesis, Université de Genève, Geneva, 1991.

[13] Eduardo Casais, “Automatic Reorganization of Object-Oriented Hierarchies: A Case Study,” *Object-Oriented Systems*, vol. 1, no. 2, Dec. 1994., pp. 95–115

[14] Fabiano Cattaneo, Alberto Coen-Porisini, Luigi Lavazza and Roberto Zicari, “Overview and Progress Report of the ESSE Project: Supporting Object-Oriented Database Schema Analysis and Evolution,” in *Proceedings 10th TOOLS Conference, Versailles*, ed. B. Magnusson and J.-F. Perrot, Prentice Hall, 1993, pp. 63–74

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

[16] Stewart M. Clamen, “Type Evolution and Instance Adaptation,” Technical Report CMU-CS-92-113, Carnegie-Mellon University, Pittsburgh, June 1992.

[17] William R. Cook, “Interfaces and Specifications for the Smalltalk-80 Collection Classes,” *ACM SIGPLAN Notices* (special issue on OOPSLA ’92), vol. 27, no. 10, Oct. 1992, pp. 1–15.

[18] Thibault Estier, Gilles Falquet and Michel Léonard, “F2: An Evolution Oriented Database System,” Cahiers du CUI no. 69, Centre Universitaire d’Informatique, Genève, January 1993.

[19] D. H. Fishman, J. Annevelink, D. Beech, E. Chow, T. Connors, J. W. Davis, W. Hasan, C. G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, 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*, ed. W. Kim and F. H. Lochovsky, Frontier Series, Addison-Wesley/ACM Press, 1989, pp. 219–250.

[20] Erich Gamma, "Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design," Dissertation, Universität Zürich, August 1991.

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

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

[23] Ralph E. Johnson and William F. Opdyke, "Refactoring and Aggregation," in *Object Technologies for Advanced Software (First JSSST International Symposium)*, *Lecture Notes in Computer Science*, vol. 742, Springer-Verlag, Nov. 1993, pp. 264–278.

[24] Randy H. Katz, "Towards a Unified Framework for Version Modelling in Engineering Databases," *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990, pp. 375–408.

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

[26] Hyoung-Joo Kim, "Algorithmic and Computational Aspects of OODB Schema Design," in *Object-Oriented Dabatases with Applications to CASE, Networks and VLSI CAD*, ed. R. Gupta and E. Horowitz, Prentice Hall, 1991, pp. 26–61.

[27] Barbara Staudt Lerner and A. Nico Habermann, "Beyond Schema Evolution to Database Reorganization," *ACM SIGPLAN Notices* (special issue on OOPSLA '90), vol. 25, no. 10, Oct. 1990, pp. 67–76.

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

[29] Karl Lieberherr, Ian Holland and Arthur Riel, "Object-Oriented Programming: an Objective Sense of Style," *ACM SIGPLAN Notices* (special issue on OOPSLA '88), vol. 23, no. 11, Nov. 1988, pp. 323–334.

[30] Bertrand Meyer, *Eiffel: The Language*, Object-Oriented Series, Prentice Hall, 1992.

[31] Bertrand Meyer, "Tools for the New Culture: Lessons from the Design of the Eiffel Libraries," *Communications of the ACM*, vol. 33, no. 9, Sept. 1990, pp. 68–88.

[32] Shamkant B. Navathe, Seong Geum, Dinesh K. Desai and Herman Lam, "Conceptual Design for Non-Database Experts with an Interactive Schema Tailoring Tool," in *Proceedings of the 9th Entity–Relationship Conference*, ed. H. Kangassalo, Lausanne, 8–10 Oct. 1990, pp. 3–20.

[33] *Objective-C Compiler Version 4—User Reference Manual*, StepStone Corporation, Sandy Hook, 1988.

[34] William F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[35] D. Jason Penney and Jacob Stein, "Class Modification in the GemStone Object-Oriented DBMS," *ACM SIGPLAN Notices* (special issue on OOPSLA '87), vol. 22, no. 12, Dec. 1987, pp. 111–117.

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

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

[38] Dave Thomas and Kent Johnson, "Orwell: a Configuration Management System for Team Programming," *ACM SIGPLAN Notices* (special issue on OOPSLA '88), vol. 23, no. 11, Nov. 1988, pp. 135–141.

[39] Emmanuel Waller, "Schema Updates and Consistency," in *DOOD'91 Proceedings*, ed. C. Delobel, M. Kifer and Y. Yasunaga, *Lecture Notes in Computer Science*, vol. 566, Springer-Verlag, Dec. 1991, pp. 167–188.

[40] Franz Weber, "Getting Class Correctness and System Correctness Equivalent — How to Get Covariance Right," in *Proceedings 8th TOOLS Conference, Santa Barbara*, ed. R. Ege, M. Singh and B. Meyer, Prentice Hall, 1992, pp. 199–213.

[41] Roberto Zicari, "A Framework for Schema Updates in an Object-Oriented Database System," in *Building an Object-Oriented Database System — The Story of* O_2, ed. F. Bancilhon, C. Delobel and P. Kanellakis, Morgan Kaufmann, 1992, pp. 146–182.