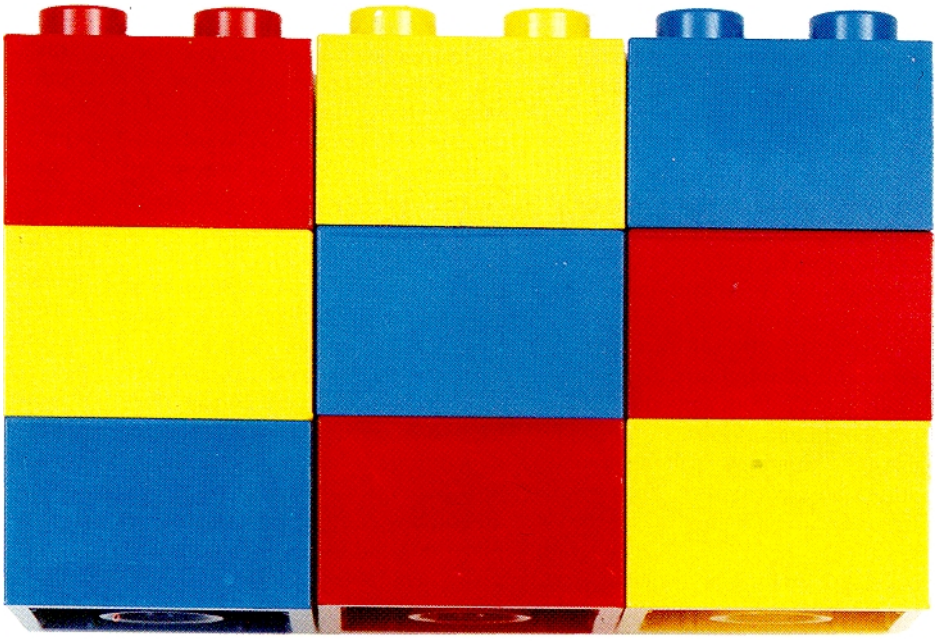
Simon Gibbs, Dennis Tsichritzis, Eduardo Casais,

CLASS MANAGEMENT FOR SOFTWARE COMMUNITIES

Oscar Nierstrasz, and Xavier Pintado

Object-oriented programming may engender an approach to software development characterized by the large-scale reuse of object classes. Large-scale reuse is the use of a class not just by its original developers, but by other developers who may be from other organizations, and may use the classes over a long period of time. Our hypothesis is that the successful dissemination and reuse of classes requires a well-organized community of developers who are ready to share ideas, methods, tools and code. Furthermore, these communities should be supported by software information systems which manage and provide access to class collections. In the following sections we motivate the need for software communities and software information systems. The bulk of this article discusses various issues associated with managing the very large class collections produced and used by these communities.



Software Communities

Software development and maintenance cause major headaches for most organizations. Although it has been recognized as a problem for many years now, software development still costs too much and induces overruns and delays. Advances have been made over the years, particularly in the area of Computer-Aided Software Engineering (CASE) tools which aim to improve productivity. In spite of these improvements, software development has resisted efforts at mechanization or automation. It is perhaps time to recognize that there is something intrinsically different about software development which does not allow easy automation.

It is widely recognized that software development is not repetitive but requires much creative and intellectually taxing effort. Therefore, it is different from most manufactured products. Nevertheless, we still dream of "software factories" which will cheaply produce high quality software (see [20] for an early expression of this idea). The problem, perhaps, is that we approach software development with the wrong paradigm. If we approach software using a mathematical paradigm, the program resembles a proof of a stated problem (the theorem). The emphasis is on structure, methodical development and proof of correctness. If we approach software with an engineering/manufacturing paradigm, we view the program as a product built by a well-known procedure whose steps have to be streamlined. Over the years, as a result of considerable research activity we have achieved some success using these paradigms. However, the fact that software development and maintenance are still a problem should encourage the search for other paradigms.

One new paradigm is offered by object-oriented programming. This paradigm, when fully applied, promotes a method of development we call *cooperative large-scale reuse.* This method can be illustrated by use of a legal analogy. Suppose a program corresponds to a legal case: its development and maintenance parallel the legal effort associated with building and presenting a legal case. Such an analogy would have been natural if the pioneers of computer science had been lawyers rather than mathematicians and engineers. Note that, within this analogy, it is difficult to talk about the correctness of software, or software factories, for the analogy immediately points out the difficulties in considering correctness of a legal case or a legal case factory.

The most interesting insights, however, come in a positive sense when we consider how lawyers go about building a case. First, they base their arguments on past experience accumulated not only by themselves, but especially by their colleagues. Recording this experience is an integral part of the legal process. Second, a legal case continuously evolves. There is no notion of separating design from implementation or development from maintenance. Instead, each legal case continuously develops (through the appeal pro-

cedure) and links up to previous and eventually future cases. Two outstanding characteristics of legal effort seem, therefore, to be the reusability of past experience and a continuously evolving effort.

We will now draw parallels from the analogy and apply the characteristics of legal effort to software. The two outstanding characteristics of software development and maintenance should be reusability of experience and evolving software. To increase productivity of software development one should reuse past experience, in the same way a lawyer building a legal case uses past ideas, arguments and cases. By the term *past experience* we mean to include requirements, specifications, models, designs and software components. To promote evolving software we should be able to interchange parts, such as documentation, designs, and software components, and link them in various ways, just as a lawyer enhances his case by continuously rearranging his arguments, drawing in new ones and abandoning those that are unsuccessful.

Like legal work, software development and maintenance are intellectually taxing. Both can benefit from proper organization and appropriate use of technology to help manage and locate information. The prevailing software engineering methods tend to cover all phases of software development for every single project, from requirements collection, analysis and specification, all the way to coding. Reuse of experience and software is effectively discouraged by restricting the context to a single application at a time [22]. We argue, on the other hand, that long-term gains in software productivity and reliability can only be achieved by adopting a more global view of software development.

In particular, software development can be viewed as taking place within the context of a *software community.* Just as there are legal communities—groups of lawyers with common areas of legal expertise and a shared history of legal cases—so there should be software communities: groups of people engaged in the development, and also the dissemination and end use, of pieces of software. An essential characteristic of any community is its history: an accumulation of collective past experience. The history of a software community would be the experiences gained in the design, development, use and maintenance of software for particular application domains. For a software community to function efficiently it must learn from and take advantage of this wealth of experience.

In our ideal scenario, applications would be based on generic software components accumulated by a software community familiar with the application domain. To build a new application, a developer could collect requirements according to an existing, well-defined model of the domain, select generic software components according to these requirements, and initialize and compose the selected components to construct the running application. By analogy, lawyers would like to handle all legal cases as though they were slight variations on textbook cases.

Although this scenario is rather idealized, we believe it can be realized to a greater or lesser extent, depending on how well an application domain can be characterized, and on how routine the required applications will be. In fact, commercially available generic software, (such as spreadsheets, relational databases, and hypertext systems), is already proving this scenario workable for certain application domains. Even in cases where clients have very specific requirements, we believe a large part of an application should be *boilerplate,* with only a few software components being designed specifically to meet the new requirements.

To approach this scenario as closely as possible for any given application domain, it is clear that we must support the process of developing generic, reusable software. To this end we must

1. organize and manage software and information about software development,
2. make it easy to find information concerning prior projects that may be relevant to new projects, and
3. provide support for the gradual evolution of software and software components.

Software Information Systems

The use of *software information systems* is one way of achieving the above three goals and improving the efficiency of software communities. A software information system is a repository, likely very large, containing all the information, including documents, designs, and software components, relevant to the functioning of a particular software community. The system should be readily available to members of the community and continuously augmented as software is developed or refined.

To make the notion of a software information system more concrete we shall assume that applications are developed using an object-oriented approach and that individual software components are primarily classes written in an object-oriented programming language. Object-oriented languages, through mechanisms of encapsulation, data abstraction, instantiation, inheritance, genericity, and strong typing, have demonstrated their potential in developing toolkits and libraries of reusable software components. Although we make few assumptions about the nature of the particular mechanisms supported by the language of choice, we feel it reasonable to suppose that object classes and some form of class inheritance will play an important role. A starting point, then, is to consider a software information system as a collection of object classes.

There are a number of advantages to collecting and organizing classes within an information system. First, the classes will be indexed to help with retrieval. Second, by applying quality control procedures to classes added to the system, developers can be more certain of the reliability of classes obtained from the system.

Furthermore, a software information system with knowledge about dependencies between classes can ensure that its contents be complete (missing files or definitions are often problems when reusing software). Finally, by obtaining a class from a repository, developers are more likely to get a standard version rather than a version full of undocumented local modifications.

There has been considerable work in the area of database support for software development [3, 4, 15, 28], primarily in the context of extending programming environments with database facilities for project and configuration management. We view a software information system in a rather different light, as an autonomous service, not necessarily tightly coupled with the programming development tools but, nevertheless, easily accessible by these tools. The closest existing systems of this nature are electronic bulletin boards and the various software repositories scattered over Internet. Such facilities, while useful, are very limited in their functionality.

We will call the task of maintaining a collection of classes *class management.* Class management includes many traditional database management issues such as data modeling, access methods and authorization. Additionally, class management encompasses new issues specific to classes. For instance, as requirements change or designs improve, classes must change; we call this class evolution. When the collection is large, developers may require assistance in finding a class for reuse; we call this class selection. There is the problem of preparing classes for reuse: class packaging. Other class management issues pertain to security and pricing policies. These include keeping the class collection free from viral infection or, when a class is proprietary to particular groups, helping to enforce licensing contraints.

Next we explore the basic issues in class management by discussing approaches to organizing and managing classes so as to support software development and reuse, approaches to browsing and querying a collection of object classes, and techniques for the controlled evolution of object classes and class hierarchies. Our objective is not to propose a design for software information systems, but rather to identify and categorize some of the critical issues that must be addressed when designing these systems.

Class Packaging

Object-oriented programming has been described as a "packaging technology" [9]. *Class packaging* is the problem of representing an object class so that the information needed to use the class can be easily located and incorporated within an application. A straightforward approach to packaging would be to represent classes by source text and store these representations in a file system. The information could be organized using simple mechanisms such as filenaming conventions and directories, and accessed through standard utilities such as editors and file browsers. However, even if the number of classes is small, this representation may present difficulties. For instance, on a UNIX™ system a C++ programmer typically represents a class X by two files: a source file, X.c, and header file, X.h, containing public declarations. Suppose X.h consists of:

```
#include "common.h"
#include "Y.h"
#include "Z.h"
class X : public Y, public Z {
    int       x;
protected:
    void      setx(int);
    int       getx();
public:
              X(int);
              ~X();
};
```

Given X.h, a programmer who wants to make use of class X would have to locate at least the following information:

- the include files common.h, Y.h, and Z.h,
- the source code or object code for the methods X::setx, X::getx, X::X, and X::~X, and
- the source code or object code for methods of the classes Y and Z.

In addition the programmer would have to consider

- whether the names (classes, structures, type definitions, etc.) used in common.h, Y.h, or Z.h, are in conflict with names already in use,
- whether any of common.h, Y.h, or Z.h, in turn refer to other include files,
- if object code is available, whether it is suitable for the run-time environment (processor, operating system) the programmer intends to use,
- if source code is available, whether it is suitable for the development environment (compiler, operating system) the programmer intends to use,
- whether X will be reused directly or refined. In the first case the programmer may want to examine the source of public methods of X; in the second case the programmer may also want source of private and protected methods.

As the number of classes increases, more problems appear with this representation: it becomes difficult to find classes, relationships between classes are not explicitly represented and so must be deduced from the source code, and adding new classes may involve rearranging the file system. By choosing a richer, more explicit representation of class structure, the software information system can be of greater assistance in managing large numbers of classes. For instance, advanced querying and browsing facilities, versioning, and high-level interfaces to development tools all require, to some extent, knowledge of the structure and relationships of classes.

An early example of class packaging can be found in Xerox's PIE (Personal Information Environment) [14]. PIE is an extension of the Smalltalk programming environment in which Smalltalk classes are represented by layered networks. The nodes of these networks contain various chunks of code for the associated class, (see Figure 1 for a

simplified example). Each layer corresponds to a different design of the class (in the example shown, class X has one method in the initial layer and a second method added by the superseding layer). One advantage of representing classes by data structures rather than text is that software can then be integrated with other forms of information. This is illustrated by PIE since it supports the creation of hypertext-like links between nodes containing code and nodes containing documentation.

A more recent example of packaging is found in the Trellis programming environment [27]. As a programmer defines new classes using the Trellis/Owl language, representations consisting of the source code of these classes are added to a database. This information is shared and augmented by the programming tools within the environment, including a cross-referencing tool and a compiler which adds object code and possibly error information. A second advantage of representing classes by data structures, rather than text, is that it is easier to build tools which examine and manipulate classes. Trellis is an open-ended environment where tools can be added or modified. This is, at least in part, a result of the packaging and sharing of class definitions provided by the database.

It is natural to ask what are the characteristics of useful class representations. We believe three things are important: First, the representation should allow a structural decomposition of the class into a number of logical components. Second, the representation should permit the attachment of descriptive information. Third, the representation should support multiple views.

Structural Decomposition. By structural decomposition we mean breaking the representation of a class into a number of interrelated components. In choosing a decomposition for classes written in a particular programming language, one can be guided by the constructs provided by the language. So if the programming language supports class and instance variables, the representation should contain structural components corresponding to both class and instance variables. Similarly, if methods may be private or public it should be possible to capture this distinction within the representation. However, there is a tradeoff between the granularity of structural decomposition and simplicity of the representation: as the representation becomes more finely detailed, its use by tools such as browsers becomes more complex.

Descriptive Attachment. Not all components of the class representation need be derivable from source code. The representation should allow one to attach components corresponding to descriptive attributes. Possible attributes include the author of the class, the date it was written, version and release information, and comments or documentation. For retrieval purposes it is useful to attach textual descriptions of the class. This could be a set of keywords, or descriptors from a software classification scheme such as described in [34].

Multiple Views. Structural decomposition of classes is a very general mechanism which can be used in a number of ways. One use is in versioning, the advantage being that only those components differing from a previous version need be stored. This is demonstrated by PIE. Structural decomposition is also useful for browsing—since the browser can then display or highlight different parts of the class in different ways, and for querying—since it is then possible to express and evaluate queries which refer to different parts of the class. However, the representation of a class may become rather complex. Considering only versioning there are many complications. Versions may have different designs (i.e., different signatures), may refer to different stages of development, their implementations may differ (i.e., different choices for internal data structures and algorithms), and their compilations may differ (i.e., object code for various machine architectures). In order to cope with this complexity it is useful if multiple views of a class are supported.

Some examples of views include the private and public parts of a class, the implementations of a class (see Figure 2), and owner-versus-user views of a class [42]. Other examples of views can be found in the ways various object-oriented languages organize methods. For instance, Smalltalk conventionally groups methods into categories. In CV++ [37], an extension of C++, methods can be grouped into a number of interfaces. Other proposals group methods into roles—each object has a current role and will only respond to methods associated with that role [30, 36]. In such cases one may want to be able to view a class from the perspective of a particular category, interface, or role. Finally, in a multilanguage environment where, for instance, both C++ and Smalltalk classes are needed, it may be useful to have a coarse language-independent view, showing perhaps only class names and method names, in addition to more detailed, language-dependent views. In general, as these examples show, a view mechanism allows classes to be dealt with at different levels of detail and in more flexible ways.

Class Organization

Class packaging deals with the representation of single classes. *Class organization,* on the other hand, deals with the relationships and dependencies that occur in collections of classes. A software information system should capture the relationships between classes for a number of reasons. First, it is needed for reuse; although classes have been proposed as units of code reuse, it is often the case that one class depends on another and so it is not single classes but groups of classes which are reused. Second, knowledge of class relationships can help with browsing since a browser needs to identify related pieces of information. Finally, class relationships can also help to detect inconsistencies or incompleteness. For example, a software information system would be incomplete if it con-

tained a class but not its superclass.

It is useful to distinguish two categories of relationships involving classes. The first, structural relationships, are derivable from source code. Examples include the *SubclassOf* or inheritance relationship, *InstanceOf,* and a *DependsOn* relationship. Relationships of the second category are those which are not derivable from source code; instead these are explicitly defined by some agency external to the software information system. For example, a project could define a relationship for the purpose of collecting the classes which it uses. We now look at some of the issues involved in representing relationships among classes.

SubclassOf (inheritance). Inheritance is one of the standard features of object-oriented languages [44]. Thus we would expect a software information system to keep track of which classes are subclasses of other classes. Representing this relationship itself is straightforward; single inheritance is a *1-n* relationship between classes while an *m-n* relationship is needed for multiple inheritance. An interesting question is to what extent the software information system need model the semantics of inheritance. There are many varieties of inheritance [25, 39]. To take one example, object-oriented programming languages differ on whether the instance variables of a superclass are visible to the methods of a subclass. If we want the software information system to provide a view of a class showing all available instance variables or all available methods, as does the "flat" view of Eiffel [21], then it will be necessary to model some of the semantics of inheritance. Furthermore, such views involve calculating the transitive closure of the *SubClassOf* relationship, so efficient traversal of this relationshp must be possible within the software information system.

InstanceOf. The role of the *InstanceOf* relationship within software information systems requires some clarification. We see software information systems as containing representations of classes, but generally not instances of these classes. Instances would be created and managed by applications constructed using the classes provided by a software information system. However, there are situations when an inter-class *InstanceOf* relationship is useful. Some object-oriented languages contain metaclasses. In this case classes can be viewed as instances and the software information system would need to represent both classes and metaclasses as well as the relationship between the two. A second potential use is in modeling parametric polymorphism. Some object-oriented languages contain constructs which can be expanded into class specifications by binding type parameters. Such polymorphic class specifications could be modelled as metaclasses, in which case the derived class would be an instance of the metaclass.

DependsOn (ClientOf, PartOf). One class may depend on another in a variety of ways: A class may be a *ClientOf* (i.e., invoke) the methods of another class. One class may be *PartOf* a second, as when a class has instances of other classes among its instance variables. In strongly-typed object-oriented languages a class may depend on another by declaring it as the type of a method parameter. These are examples of a general *DependsOn* relationship that identifies

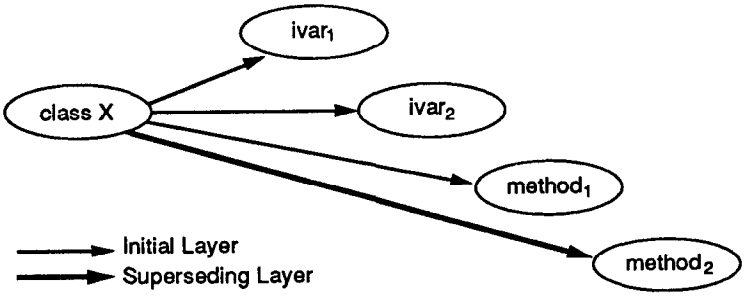


FIGURE 1. PIE Network Layers.

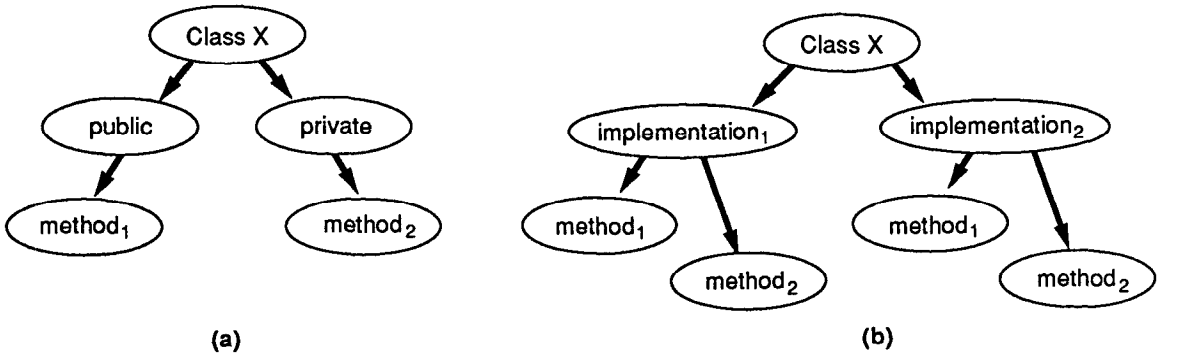


FIGURE 2. Alternative Views.

the various syntactic references between classes. A software information system should be able to determine for a given class, which classes it depends on, and conversely, which depend on it.

These relationships are common to many object-oriented languages. There are other relationships which are more language-dependent, such as the "friend" relationship found in C++ [40]. If one class declares a second as its friend, then the private methods and instance variables of the first class are available to the second. Other examples result from aggregations of classes such as "features" [17] and "frameworks" [45]. Both features and frameworks involve groups of classes: a feature is a language construct that specifies an interface to some group of classes while a framework is a subsystem design based on an inter-working group of classes. In these examples, one class may be related to another via participation in the same feature or framework. In general, any language-dependent software information system may have to represent a number of additional relationships derived from the language concerned. Figure 3 shows an example of a more extensive group of relationships used to represent a C++ class collection.

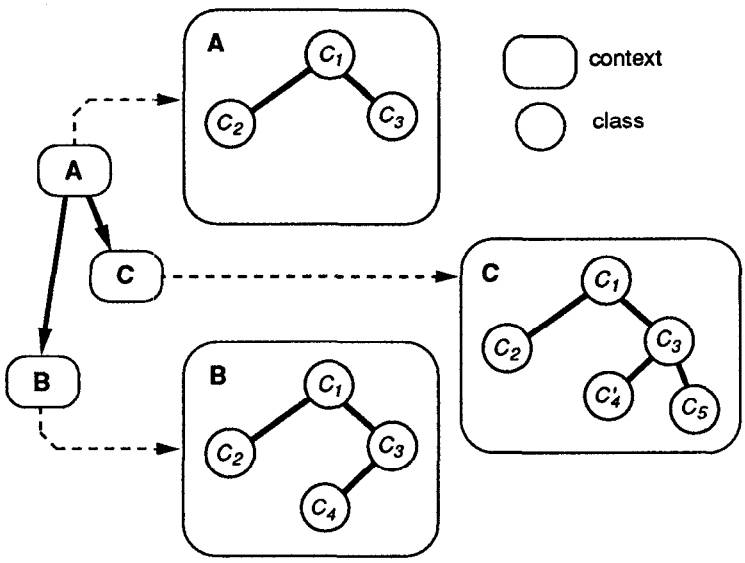


FIGURE 3. C++ Class Relationships.

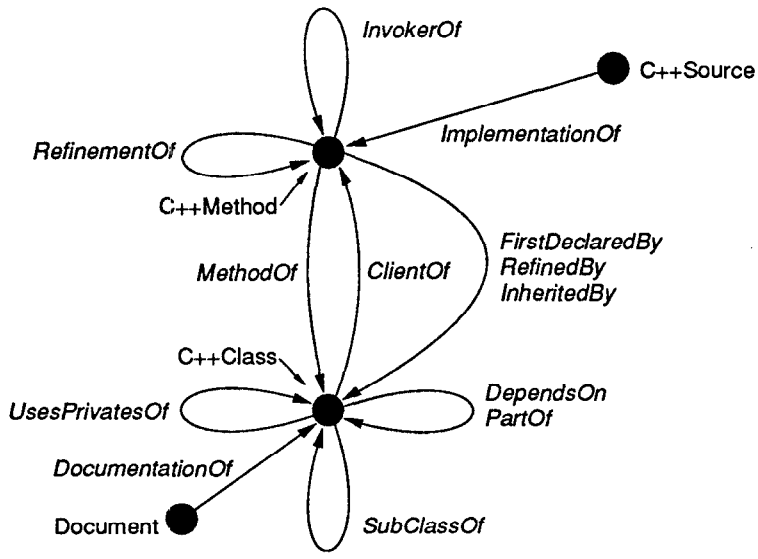


FIGURE 4. Class Hierarchies within Contexts.

In addition to structural relationships such as *SubClass, InstanceOf,* and *DependsOn,* class organization also requires relationships not derivable from source code. These include relationships that associate documentation and other design information with classes. The nature of these relationships depends on many factors such as the procedures for adding a class to the software information system and documentation conventions and formats. For example, Figure 3 shows a simple "DocumentationOf" relationship between C++ classes and documents. In practice, however, a more refined and versatile inter-linking of classes and documentation is likely to be necessary.

In addition to organizing classes in terms of inter-class relationships, it may be useful to have more abstract groupings of the class collection. In many object-oriented programming languages the class name space is essentially flat. This can be problematic in a multi-user environment since a monolithic class hierarchy constrains the designer of new objects to avoid name clashes. A simple example would be a CAD programmer who wants to provide a "Window" object class for use in architectural applications but is unable to because of a conflict with a user-interface "Window" class. A more subtle form of this problem may also occur in object design. There is a tendency for the initial choice of object classes within a given application domain to prescribe the design of future applications for the domain. It can be difficult for a designer to break out of the prescribed design by class specialization:

1. inheritance is now working against the designer, and
2. the designer really wants a reorganization of the class hierarchy.

As a result, the class hierarchy may become a rigid constraining structure that hampers innovation and evolution.

For large software information systems it appears that a single class hierarchy is just too simple. What is needed is a a *context mechanism,* so, for instance, the object classes deriving from a particular design for a particular domain can be grouped together. One possible solution may be context hierarchies, each context corresponding to a class name space. As an example, Figure 4 shows three contexts: A, B and C. The class hierarchy visible within a given context consists of those classes visible within the context's parent and any additional classes defined within the context in question. For instance, context B includes classes C_1, C_2 and C_3 from its parent, A, and the locally-defined class C_4. A map of the context hierarchy, such as the small tree appearing in the left of Figure 4, provides a high-level global view of the class collection.

Class Selection and Exploration

We now discuss the general problem of retrieving information from a class collection. There are many programming situations where retrieval is necessary. A user (such as a programmer or application developer) may, for example, be looking for a specific class—perhaps the class of complex numbers or a particular version of a window class. Alternatively, the user may be looking for functionality that is provided by any of a number of classes in the system, or simply trying to get a feel for the scope of the class collection. We can divide these retrieval activities into two groups: *class selection* and *class exploration.* Class selection refers to the situation in which the user has fairly specific selection criteria, such as the name of a class or method, or an area of functionality. With class exploration, on the other hand, the user is not interested in individual classes but rather in the relationships among classes and the overall organization of the collection. This is the case, for instance, when a programmer is implementing a new application and wants to determine which classes may be relevant to the application. The two methods commonly used for retrieval are querying and browsing. Querying is useful when search criteria are known, it is thus more appropriate for selection—while browsing is more appropriate for class exploration.

Class browsers

Currently most programming environments do not contain extremely large numbers of classes—thus a single tool, a class browser, is used for both selection and exploration. This approach is exemplified by the Smalltalk-80 browser [13] which allows a user to browse through the class inheritance hierarchy, display instance variables and methods, and determine which classes send or receive a given message. Classes are grouped by functionality into possibly overlapping categories, and it is possible to browse through categories of classes and methods. The Smalltalk browser has been extended in many ways. For instance with the PIE browser [14], it is possible to associate textual components to classes, categories and other entities of the system in order to help in the understanding of the system. The PIE browser also provides multiple views. It is possible, for example, to present the user with a set of views adapted to different application domains. One such view might correspond to a development project where classes are being developed incrementally and thus should be kept hidden from other users not involved in the development effort. The ability to define partial views can reduce the complexity of the system as it appears to a particular user.

Most of the existing browsers have been tested on small- or medium-scale software projects. Although extrapolating their usefulness is not an easy task, it is natural to ask whether the Smalltalk approach is scalable and whether it will be able to cope with the potential size of software information systems. We believe that current browsers are unlikely to be adequate for selection when class collections increase in size by a few orders of magnitude.

As the size of the class collection increases, class selection becomes more difficult and query facilities are of greater benefit. There has been relatively little work in the area of class selection, although information retrieval techniques may be applicable [10]. One proposal that appears promising is the software classification scheme developed by Prieto-Diaz and Freeman [34]. This scheme uses a six-tuple of *facets* or descriptive attributes, to classify software components according to such things as functional area, medium and system type. Furthermore, a conceptual distance based on facet values can be used to estimate the match of a component to a particular query.

Another question is whether browsing is sufficient for users who are interested in exploring the functionality of a class collection. The primary navigational structure used by browsers based on the Smalltalk approach is the inheritance hierarchy. However, in most object-oriented programming languages, the semantics of inheritance is not sufficiently constrained for it to give useful insight into the functionality of subclasses. This is illustrated by the following examples:

- A subclass may add behavior to that of its superclass.
- A subclass may provide the same interface as its superclass but reimplement the methods.
- With multiple inheritance, a subclass may override a method from one superclass with that from another.

In general, it is possible that classes related by inheritance provide dissimilar functionality while classes unrelated by inheritance may provide similar functionality, so merely knowing the inheritance relationships between classes gives little indication of how the functionality of a subclass differs from its superclass or why the subclass appears where it does in the hierarchy. Typically the

user will resort to comparing the code belonging to the two classes. However, determining the structure and dependencies of a set of classes by examining the code is difficult [41] and contrary to encapsulation.

The problem of guiding a user engaged in exploring the class space is similar to the problem of providing navigational assistance in hypermedia environments, a subject that has received much attention recently [43]. Possible features that could be integrated in a class browser are global views of the organization of the system and navigation charts that help users visualize their position and the structure of the surrounding space.

Affinity browsing

Another approach to guiding exploration is by providing means for determining the similarities between classes, their interfaces and their functionality. In this case the "nearest neighbors" of a class are not simply its super and subclasses but rather those classes which it somehow resembles. We call this *affinity browsing*. The principal assumption of this approach is that in a software information system containing a large collection of inter-dependent classes, the relationships among these classes are complex and can be viewed in many ways.

The affinity browser [32] is an attempt to integrate navigational aspects of conventional browsing with query capabilities. The affinity browser provides the user with a set of two-dimensional views, each displaying some relationship among a set of classes. One view could be based on the usual inheritance relationship while another could portray a grouping of classes based on their relevance to some query. An affinity function, which defines the intensity of a relationship, is associated with each view. When the view is displayed, distances between classes convey their affinity (i.e., pairs of classes with strong affinity are displayed close together) while those with less affinity lie further apart. For example, classes that implement similar functionality, or have similar signatures, could have a higher affinity, and would then cluster together when displayed.

In order to apply affinity browsing to class exploration we need to define affinity functions for classes. Clearly there are many such functions, some more useful than others. Some potential candidates include the distance between two classes on the inheritance hierarchy, the conceptual distance between two classes using some classification scheme such as facets, the textual similarity of the signatures of two classes, the amount of code shared between two classes, or a measure based on class dependency (where two classes are similar if they depend on the same classes).

As a specific example of an affinity function and view generation, consider Figure 5 which shows the inheritance structure of a set of classes, $C = \{C_0, ..., C_8\}$, and the methods defined (and redefined) by each class. Assume that classes recursively inherit methods from their superclasses. Let $M(X)$ be the set of methods in the interface to class X. For instance

$$M(C_7) = \{a, b, e, f, i, j, o, p\}.$$

We want to define an affinity function that conveys the extent to which classes provide similar functionality. As a candidate function, suppose we define $A(X, Y)$, the affinity between class X and class Y, as

$$A(X, Y) = \frac{card(M(X) \cap M(Y))}{card(M(X) \cup M(Y))}$$

where *card()* is a function that returns the cardinality of a set. For example, to evaluate $A(C_3, C_4)$, we have

$M(C_3) = \{a, b, g, h\}$ and
$M(C_4) = \{a, b, i, j\}$, so
$card(M(C_3) \cap M(C_4)) = 2$ and
$card(M(C_3) \cup M(C_4)) = 6$. Hence,
$A(C_3, C_4) = 1/3$.

Of course other definitions are possible and it may be necessary to perform a few iterations before one obtains views which convey a good intuition of the underlying relationship. To illustrate this point, the above function could be modified slightly in the case of redefinition of an inherited method (such as method a of class C_1). Suppose we want to emphasize that redefined functionality differs from inherited functionality. Let m be the inherited method and m' be its redefinition. In the case where both m and m' appear in $M(X) \cup M(Y)$ then in the affinity calculation we consider $m = m'$ in $M(X) \cap M(Y)$ while in $M(X) \cup M(Y)$ we take $m \neq m'$. This produces a slight reduction of the affinity between classes where one redefines a method of the other.

Figure 6 depicts a typical view generated by the affinity browser using the previously defined measure of affinity. The highlighted class, C_4, is the *current class*. The *Inspect Window* displays the names of the classes within the view, these can be selected to obtain further information about each class.

The affinity browser promotes the local exploration of the class space. The user selects a class, it becomes the current class, and the tool displays the classes that are within a user-defined affinity neighborhood (i.e., those that have an affinity with the current class that is greater than a user-defined limit). Selecting a new current class causes a shift in the neighborhood; new classes enter the view while others disappear. Views can be connected in the sense that they can be constrained to have the same current class. Each view then provides a different exploration context; they are centered on the same class but have different neighborhoods since different affinity functions are involved.

It should be pointed out that given a measure of affinity it is not possible, in general, to generate a two-dimensional representation that satisfies all the affinity constraints. The view layout algorithm [31, 33] attempts to find a good approximate solution. For example, it does not assign the same weight to each affinity constraint. It assumes that it is more important to provide an accurate representation of affinity between the current class and the other classes of the view than between two arbitrary classes.

Class Evolution

Issues

Classes developed with an object-oriented language frequently undergo considerable reprogramming before they become readily reusable in a wide range of applications or domains. There are a number of reasons for this phenomenon:

- Experience shows that stable, reusable classes are not designed from scratch, but are "discovered" through an iterative process of testing and improvement [16].
- Classes are difficult to arrange in predefined taxonomies.
- Because user's needs are rarely stable, additional constraints and functionalities have to be constantly integrated into existing applications.
- Reusing software raises complex integration problems when teams of programmers share classes that do not originate from a common, standard hierarchy.

To apply such powerful techniques as inheritance, genericity, and delayed binding efficiently, real-world concepts have to be properly encapsulated as classes so they can be specialized or combined in a large number of programs. Inadequate inheritance structure, missing abstractions in the hierarchy, overly specialized classes or deficient object modeling may seriously impair the reusability of a class collection. The collection must therefore evolve to eliminate such defects and improve its robustness and reusability.

Several approaches, ranging from class tailoring to class reorganization, have been proposed to improve class collections. We will now describe some relevant techniques developed recently for controlling evolution in object-oriented environments, and discuss their respective merits.

Class tailoring

Object-oriented languages have always provided simple constructs for tailoring class hierarchies, notably by allowing the redefinition of inherited properties. The body of a method, for example, can be completely modified in a subclass, although its name and its signature remain identical. Therefore, it is possible to implement specialized or optimized versions of the same method, rather than using the general, and perhaps inefficient algorithm defined in a superclass. Some languages, such as Eiffel, allow the type of inherited variables, parameters and function results to also be overridden, provided the new type is compatible with the old one [21]. With the object-oriented variants of LISP, the programmer can choose how to combine inherited methods in a new class [24].

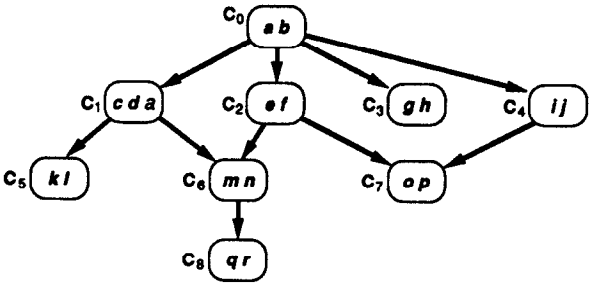


FIGURE 5. Inheritance Structure of a Set of Classes.

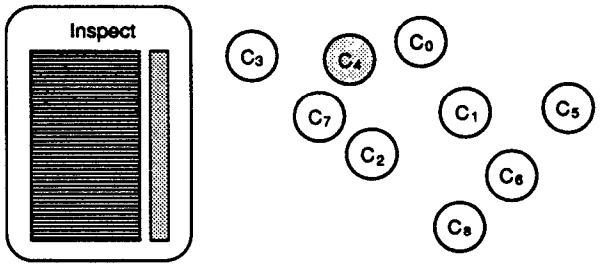


FIGURE 6. Affinity Browser Display.

A similar, but more formal approach is described in [7]. The author proposes a mechanism for excusing abnormal cases that arise when modeling an application domain, and that do not fit with the existing class hierarchy. For example, a system for managing information on students may have to cope with the case of people who did part of their studies in foreign countries with different grading schemes and academic titles. Contradictions between the definition of the "foreign student" class and its superclass ("normal student") must be explicitly acknowledged. The explicit redefinition of inherited attributes according to a formal model integrating excuses with inheritance facilitates the detection of type violations and the correct handling of database queries (without overlooking exceptional entities). Moreover, exceptions are handled locally, and do not require the factoring of common properties into numerous intermediate classes.

These techniques are useful for performing limited adjustments to a class collection, but they do not provide any help for detecting design flaws. Over-reliance on tailoring and excuses may quickly lead to an incomprehensible specialization structure, overloaded with special cases and difficult to manage efficiently with current database technology. Such a situation is generally a strong indication that the hierarchy does not contain the proper abstractions and that it should be reorganized.

Class surgery

Whenever changes are brought to the modeling of an application domain, corresponding modifications must be applied to the classes representing real-world concepts. Modifying a class hierarchy is a delicate operation because of the multiple connections

between class definitions that must be taken into account to guarantee the consistency of the hierarchy.

This problem also arises in the area of object-oriented databases. There, the available techniques [1, 29] first determine a set of integrity constraints that a class collection must satisfy. For example, all instance variables of a class should bear distinct names, no loops are allowed in the hierarchy, the attributes defined in a class should be inherited by all its subclasses, and so on. In a second step, a taxonomy of all possible updates to the system is established. These changes concern the structure of classes, like "add a method," "rename a method," or "restrict the domain of a variable"; they may also refer to the hierarchy as a whole, as with "suppress a class," or "add a superclass to a class."

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. Generally, additional reconfiguration procedures have to be applied in order to preserve integrity constraints. It is, for example, illegal to suppress an attribute from a class C if this attribute is really inherited from a superclass of C; if the attribute can be suppressed, it must also be recursively dropped from all subclasses of C, or possibly replaced by another variable with the same identifier inherited through another subclassing path. As another example, deleting a class S from the list of ancestors of another class C is not allowed if this operation leaves the inheritance graph disconnected. If the operation does not cause any problems, the inheritance links are reassigned to point from C to the superclasses of S. Of course, the properties of S no longer belong to the representation of C, nor to those of its subclasses.

Decomposing all class modifications into update primitives and determining their consequences brings several advantages. During class design, this approach helps developers detect implications of their actions on the class collection and maintain the consistency of class specifications. During application development, it guides the propagation of changes to where the class is reused. For example, renaming an instance variable of a class, changing its type or defining a new default value, has no impact on an application using the class. Changing or deleting methods, on the other hand, generally leads to changes in applications.

Depending on the class model and on the integrity constraints, a software information system may provide different forms of class surgery. This approach, however, limits its scope to local, primitive kinds of evolution; it forms a solid framework for defining "well-formed" class modifications, but it gives no guidance as to when these modifications should be performed.

Class versioning

Versioning is a particularly appealing technique for managing class development and evolution. It enables programmers to try different paths when modeling complex application domains and to record the history of class modifications during the design process. Versioning also helps in keeping track of various implementations of the same class for different software environments and hardware platforms.

A basic problem to deal with concerns the identity of classes. It is no longer enough to refer to a class by its name, since the name might correspond to many versions of the same class. An additional version number must be provided to identify unambiguously the class referred to. When this version number is absent, a default class is assumed: the very first version of the class referred to, or its current version, or its most recent version when the software component making the reference was created.

If only the most recent version can give rise to new versions, there is in principle no need for an elaborate structure to keep track of the history of classes: their name and version number suffice to identify their relationship to each other. The case where versioning is not sequential, (i.e., where new versions can be derived from any previous version), requires that the software information system record a hierarchy of versions somewhat similar to the traditional class hierarchy.

Another difficulty arises because of the superimposition of versioning on the inheritance graph. For example, when creating a new version 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 dealing with this kind of problem. If the interface of a class is changed, then new versions should be created for all its subclasses and all its dependent classes. If only nonpublic parts of the class are modified, such as methods visible only to subclasses, or the types of instance variables, then versioning can be limited to its existing subclasses. If only the implementations of the class's methods are changed, no new versions for other classes are required.

Application developers may want to consider objects instantiated from previous class versions as if they originated from the current version, or they may want to forbid objects from an old version to refer to instances of future versions. These effects are rarely achieved by fully automatic means. For every new version, one must program special functions for mapping between old and new class structures [6, 38]. These functions filter the messages sent to objects, so that proper actions can be taken, like translating between method names, returning a default value when accessing a non-existent variable, or simply aborting an unsuccessful operation.

In spite of their overhead, class versioning techniques have proved indispensable in important domains like CAD/CAM and office information systems. They have therefore been integrated in object-oriented systems, such as Orwell [42] AVANCE [5], ORION [1], and IRIS [2].

Class reorganization

Class evolution is intimately linked with class design. Suppose programmers build applications chiefly in a bottom-up fashion by reusing existing classes. Classes may then require adaptations so that they fully suit the needs of software developers. This is achieved by redefining or suppressing attributes (instance variables and methods), reimplementing methods, changing class interfaces, etc. Such modifications indicate that the current hierarchy is not satisfactory: if classes cannot be reused as they are, if subclasses cannot be derived from other classes without considerable tailoring, then one needs to look for missing abstractions, to make some classes more general, to increase modularity, in short, to reorganize, at least in part, the hierarchy. Tools that automatically restructure a class collection and suggest alternative designs can reduce considerably the efforts required for carrying out these tasks.

One solution is to algorithmically restructure the hierarchy when introducing new classes by creating intermediate nodes, shuffling attributes among them, and rearranging inheritance paths, so as to avoid the need for explicitly redefining or rejecting attributes [8]. In the example of Figure 7, we want to insert a class that inherits attributes A and D, introduces E, but suppresses attributes B and C. The second part of Figure 7 shows how the graph has to be modified to accommodate class ADE; notice that two intermediate classes are required for its integration in the hierarchy. These additional classes represent shared modules of functionality; they correspond to constructs, such as the "mixins" of Lisp with Flavors [23], whose main purpose is not to describe real-world entities, but rather to support the implementation of other classes. More importantly, the classes introduced during the reorganization process can serve as a rough estimate for the abstractions that are missing from the modeling of an application domain. Such defects are unavoidable; it is exceptional to achieve a stable, definitive class design without going through several iterations. New classes and inheritance links correspond to the places in the hierarchy warranting redesign.

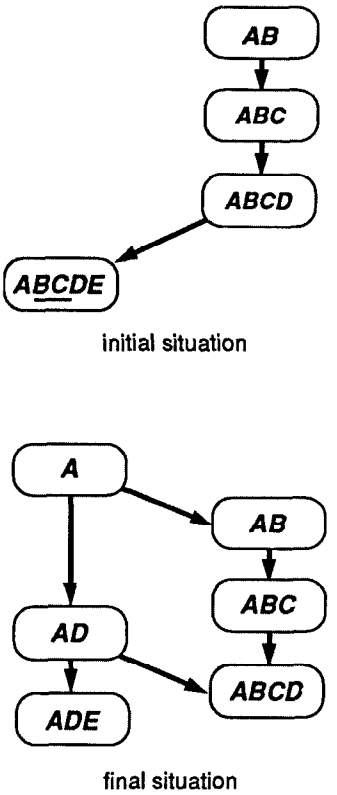


FIGURE 7.
Reorganizing a Class Hierarchy.

This approach works incrementally and preserves the structure of all original classes, except for their inheritance links. It can be extended to take into account information on types, on mutual dependencies between attributes, and on multiple inheritance. When typical evolution patterns emerge, they can help guide the design process [18].

An analogous technique is used to fully recast a class hierarchy, by getting rid of obsolete classes or unwanted versions. Global restructuring algorithms keep as much information as is needed to reconstruct all original classes, if needed; they try to enforce some properties, like allowing an attribute to be introduced at only one point in the hierarchy [8].

Reorganization can also improve the quality of classes. Some class design methods prohibit certain kinds of references to the attributes of objects [19]. Thus, a method should never access variables that do not belong to the class where it is defined or are not passed to it as parameters. Such unsafe expressions can be detected and replaced with appropriate method calls automatically. By eliminating unnecessary dependencies, classes should encapsulate functionality more tightly and show better resilience to change.

Reorganization algorithms appear useful for detecting missing abstractions, for proposing generalizations of very specialized classes, and for cleaning up a hierarchy. However, because they perform strictly structural transformation on object descriptions, their results require user intervention to compensate for the lack of knowledge concerning the application domain and the concepts embodied in the class collection.

Object-oriented development has an iterative nature and successive stages of subclassing, class tailoring, class modification, version creation and reorganization are needed to build increasingly general, reusable and robust classes. We expect, therefore, software information systems to take advantage of a spectrum of tools and techniques for managing class evolution.

Conclusion

In the preceding sections we have argued that object-oriented programming, augmented by the availability of large class collections, leads to a new method of software development which encourages the design and reuse of generic components by communities of software developers.

In establishing this method there appear to be three sets of issues which must be addressed. First, there are basic questions related to the design of systems for maintaining the class collection—what we have called software information systems. Second, we need to understand how to integrate such systems with software development methods. And, third,

there is the question of establishing the appropriate infrastructure to assure wide accessibility of these systems.

We have been more concerned with the first set of issues; in particular we have focused on class management, or how to organize and maintain large class collections. We have looked at various alternatives for representing classes and their relationships, for assisting developers to select classes, and for allowing the class collection to evolve over time. There has been little experience working with very large, shared class collections and so we plan to evaluate some of the techniques described above. Currently we are implementing a prototype, called Xos, or "external object system" which has been specifically designed for modeling object classes [11, 12]. Xos allows application development tools to concurrently create, query and modify class representations. We plan to use Xos to capture a large C++ hierarchy and then evaluate various querying and browsing facilities, such as affinity browsing, and experiment with class reorganization algorithms.

Regarding the role of software information systems and class collections in the development life cycle, it is useful to distinguish between two kinds of development activity: component development and application development. The former consists of designing and implementing reusable or generic components while the latter consists of constructing applications from primarily predesigned components. For reuse to occur there must be an increased emphasis on the development, evaluation and refinement of components, as opposed to final products or applications. Furthermore, tools must be provided that aid in configuring existing components into new applications.

We are exploring this approach by participating in Ithaca [35], a large European ESPRIT project, the aim of which is to build an environment to support the development of object-oriented applications in a variety of application domains. The environment includes an object-oriented language with database support, a software information base (SIB) which stores and manages information concerning reusable software and its intended use, a selection tool for browsing and querying the SIB and a variety of application development tools built around the SIB. Among these tools is a *visual scripting tool* for interactively constructing running applications from visual representations of packaged application objects [26].

Finally, we believe that the greatest benefits of large-scale class reuse will occur when software information systems are publicly available resources rather than confined within single organizations. Despite facilities such as electronic mail and bulletin boards, software development is still too isolated an activity. The past decade has seen the establishment of on-line services in areas such as finance and travel. These services are decentralizing and interconnecting workers in many occupations. Using the class as a unit of interchange, software development may also become a more open, networked, cooperative activity. This raises a number of pragmatic issues, some of which we have alluded to in this article. For instance, if proprietary software is placed in publicly accessible systems will it be possible to ensure that licensing and copyright conditions are met? Who will operate these systems and what services will be provided? How will they be accessed? These pragmatic issues, in addition to the technical problems of class management, must be addressed before large-scale reuse of object classes can be realized.

Acknowledgments.
The authors would like to thank their past and present colleagues at the Centre Universitaire d'Informatique and the members of the Ithaca project for helping voice the questions which led to this paper. The authors would also like to thank the reviewers for their many constructive comments.

References

1. Banerjee, J., Kim, W., Kim, H.-J., and Korth, H.F. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM (SIGMOD) Conference on the Management of Data.* (San Francisco, California, May 27-29). ACM, New York, (1987), pp. 311-322.
2. Beech, D., and Mahbod, B. Generalized version control in an object-oriented database. In *Proceedings of the 4th IEEE International Conference on Data Engineering.* (Feb. 1988).
3. Bernstein, P. Database system support for software engineering. In *Proceedings of the International Conference on Software Engineering.* (1987), pp. 161-178.
4. Biggerstaff, T., Ellis, C., Halasz, F., Kellog, C., Richter, C., and Webster, D. Information management challenges in the software design process. MCC Tech. Rep. STP-039.87. 1987.
5. Björnerstedt, A., and Britts, S. AVANCE: An object management system. In *Proceedings of OOPSLA '88* (Sept. 1988). pp. 206-221.
6. Björnerstedt, A., and Hultén, C. Version control in an object-oriented architecture. In *Object-Oriented Concepts, Databases and Applications.* W. Kim and F. Lochovsky, Eds. Addison-Wesley/ACM Press, 1989, pp. 451-485.
7. Borgida, A. Modeling class hierarchies with contradictions. In *Proceedings of the ACM SIGMOD Conference on the Management of Data* (Chicago, June 1-3). ACM, New York (1988), pp. 434-443.
8. Casais, E. Reorganizing an object system. In *Object Oriented Development,* D. Tsichritzis Ed., Centre Universitaire d'Informatique, Université de Genève, 1989.
9. Cox, B.J. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.
10. Frakes, W.B., and Gandel, P.B. Classification, storage, and retrieval of reusable components. In *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval,* (Cambridge, Mass., June 25-28). ACM, New York (1989) pp. 251-254.
11. Gibbs, S. Querying large class collections. In *Object Management,* D. Tsichritzis Ed. Centre Universitaire d'Informatique, Université de Genève, 1990.
12. Gibbs, S. and Prevelakis, V. Xos: An overview. In *Object Management,* D. Tsichritzis Ed. Centre Universitaire d'Informatique, Université de Genève, 1990.
13. Goldberg, A. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Mass., 1984.
14. Goldstein, I.P., and Bobrow, D.G. A layered approach to software design. Rep. CSL-80-5, Xerox, 1980.

15. Hudson, S.E., and King, R. Object-oriented database support for software environments. In *Proceedings of the ACM SIGMOD Conference on the Management of Data* (1987), pp. 491-503.
16. Johnson, R.E., and Foote, B. Designing reusable classes. *J. Object Oriented Programming* (June-July 1988), 22-35.
17. Kaiser, G.E., and Garlan, D. MELDing data flow and object-oriented programming. In *Proceedings of OOPSLA '87* (Oct. 1987), pp. 254-267.
18. Li, Q., and McLeod, D. Object flavor evolution through learning in an object-oriented database system. In *Proceedings of the 2nd International Conference on Expert Database Systems,* (Tysons Corner, Virginia, April 25-27, 1988), 241-256.
19. Lieberherr, K.J., and Holland, I.M. Assuring good style for object-oriented programming. *IEEE Softw.* (Sept. 1989), 38-48.
20. McIlroy, M.D. Mass produced software components. In *Software Engineering,* P. Naur and B. Randell Ed. NATO Science Committee (Oct. 1968), 138-150.
21. Meyer, B. *Object-Oriented Software Construction.* Prentice Hall (1988).
22. Meyer, B. The new culture of software development: Reflections on the practice of object-oriented design. In *TOOLS'89,* 13-23.
23. Moon, D.A. Object-oriented programming with flavors. In *Proceedings of OOPSLA '86* (Sept. 1986), pp. 1-8.
24. Moon, D.A. The common LISP object-oriented programming language. In *Object-Oriented Concepts, Databases, and Applications.* W. Kim and F. Lochovsky Ed. Addison-Wesley/ACM Press, 1989, 49-78.
25. Nierstrasz, O.M. A survey of object-oriented concepts. In *Object-Oriented Concepts, Databases, and Applications.* W. Kim and F. Lochovsky Ed. Addison-Wesley/ACM Press, 1989, 3-21.
26. Nierstrasz, O.M., Dami, L., de Mey, V., Stadelmann, M., Tsichritzis, D., and Vitek, J. Visual Scripting: Towards interactive construction of object-oriented applications. In *Object Management,* D. Tsichritzis Ed. Centre Universitaire d'Informatique, Université de Genève, 1990.
27. O'Brien, P.D., Halbert, D.C., and Kilian, M.F. The Trellis programming environment. In *Proceedings of OOPSLA '87* (Oct. 1987), pp. 91-102
28. Penedo, M.H., and Stukle, E.D. PMDB: A project master database for software engineering environments. In *Proceedings of the International Conference on Software Engineering.* (1985), pp. 150-157.
29. Penney, D.J., and Stein, J. Class Modification in the GEMSTONE object-oriented DBMS. In *Proceedings of OOPSLA '87* (Oct. 1987), 111-117.
30. Pernici, B. Objects with roles. In *Proceedings of the ACM Conference on Office Information Systems* (Apr. 1990), pp. 205-215.
31. Pintado, X., and Fiume, E. Grafields: Field-directed Dynamic Splines for Interactive Motion Control. *Vol. 13, no 1, Computers & Graphics.* pp. 77-82, Pergamon Press (1989).
32. Pintado, X., Tsichritzis, D. An Affinity Browser. In *Active Object Environments.* D. Tsichritzis Ed. Centre Universitaire d'Informatique, Université de Genève, 1988.
33. Pintado, X., Tsichritzis, D. Satellite: A Visualization and navigation tool for hypermedia. In *Proceedings of the ACM Conference on Office Information Systems* (Apr. 1990), pp. 271-280.
34. Prieto-Diaz, R., and Freeman, P. Classifying software for reusability. *IEEE Softw.* (Jan. 1987), 6-16.
35. Pröfrock, A., Tsichritzis, D., Müller, G., and Ader, M. ITHACA: An overview. In *Proceedings of the European Unix Users' Group (EUUG) Conference,* (Spring 1990), pp. 99-105.
36. Reenskaug, T., and Nordhagen E. The Description of Complex Object-Oriented Systems: Version 1. Senter for Industriforskning, Oslo, 1989.
37. Schilling, J.J., and Sweeney, P.F. Three steps to views: Extending the object-oriented paradigm. In *Proceedings of OOPSLA '89* (Oct. 1989), pp. 353-361.
38. Skarra, A.H., and Zdonik, S.B. The management of changing types in an object-oriented database. In *Research Directions in Object-Oriented Programming.* The MIT Press, Cambridge, Massachusetts, 1987, 393-415.
39. Snyder, A. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA '86* (Sept. 1986), pp. 38-45.
40. Stroustrup, B. *The C++ Programming Language.* Addison-Wesley (1986).
41. Taenzer, D., Ganti, M., and Podar, S. Problems in object-oriented software reuse. In *Proceedings of ECOOP 89 Conference,* (July 1989), Cambridge University Press, pp. 25-38.
42. Thomas, D., and Johnson, K. Orwell: A configuration management system for team programming in *Proceedings of OOPSLA '88* (Sept. 1988), pp. 135-141.
43. Utting, K., Yankelovich, N. Context and orientation in hypermedia networks. *ACM Transactions on Office Information Systems* 7, 1 (Jan. 1989), 58-84.
44. Wegner, P. Dimensions of object-based language design. In *Proceedings of OOPSLA '87* (Oct. 1987), pp. 168-182.
45. Wirfs-Brock, R.J., and Johnson, R.E. A survey of current research in object-oriented design. In *CACM,* (see this issue) Sept. 1990.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques-Software libraries; K.6.3 [**Management of Computing and Information Systems**]: Software Management

General Terms: Design, Languages, Management

Additional Key Words and Phrases: Class libraries, class management, reuse, software communities, software information systems

About the Authors:

SIMON GIBBS is an assistant professor at the University of Geneva. His research interests include database support for software development, computer-supported cooperative work, multimedia systems, and semantic data modeling.

EDUARDO CASAIS is a research assistant at the University of Geneva, from which he received a "licence" in business-oriented informatics. He is currently working for his Ph.D. on class reorganization in object-oriented systems.

OSCAR NIERSTRASZ is currently an assistant professor at the University of Geneva. His current research interests include computational models for object-oriented concurrency, and interactive tools for object-oriented application construction.

XAVIER PINTADO is a research assistant at the University of Geneva. He holds a "diplôme" in electrical engineering and a "licence" in information systems. He is currently working for his Ph.D. on class selection and exploration within an object-oriented graphics approach.

DENNIS TSICHRITZIS is a professor of computer science and director of the Centre Universitaire d'Informatique at the University of Geneva. His research interests include object-oriented environments, databases and office information systems.

Authors' Present Address: Centre Universitaire d'Informatique, 12 rue du Lac, Geneva 1207, Switzerland. simon@cui.unige.ch; casais@cui.unige.ch; oscar@cui.unige.ch; pintado@cui.unige.ch; dt@cui.unige.ch.