Chapter 9
Integrated Office Systems[1]

O.M. Nierstrasz
D.C. Tsichritzis

Introduction

New techniques are sorely needed to aid in the development and maintenance of large application systems. The problem with traditional approaches to software engineering is well in evidence in the field of office information systems: it is costly and difficult to extend existing applications, and to get unrelated applications to "talk" to each other. The object-oriented approach is already being tentatively applied in the modeling of "office objects" and in the presentation of these entities to users as such in "desktop" interfaces to office software. In order to fully exploit the approach to achieve integrated office systems, we need to use object-oriented programming languages, object-oriented run-time support, and object-oriented software engineering environments.

We can view the fundamental idea behind the object-oriented approach as that of *encapsulation*: object-oriented languages and systems exploit encapsulation in various ways in an attempt to enhance productivity through, for example, reusability mechanisms such as class inheritance. (See [Nierstrasz 1988] in this book for an elaboration of this view.) The importance of object-oriented techniques for software engineering has long been established: encapsulation has been successfully used by programmers ever since macros and subroutines were invented. Encapsulation is important not only for the purpose of decomposing large software problems into manageable pieces, but can also be exploited to achieve software maintainability, reusability, and even "rapid prototyping", as object-oriented systems like Smalltalk [Goldberg and Robson 1983] have demonstrated.

Commitments to existing software, languages and operating systems have prevented application developers from taking full advantage of object-oriented techniques until now. The possibilities are especially in evidence in the domain of office information systems, since there is often a close and natural correspondence between the entities these systems manipulate and the real-world "objects" they refer to.

Office information systems are a paradigm of a new kind of application. For a long time the applications supported by computer systems were well-defined and reasonably well-structured. In such environments it was realistic to expect that we could define the "problem" inherent in the application, then do the requirements analysis and follow on with implementation. In office information systems the "problem" often cannot be readily defined. Office

[1]In *Object-Oriented Concepts, Databases and Applications*, ed. Kim and Lochovsky, ACM Press and Addison Wesley, 1989, pp. 199-215.

procedures, being generally goal-oriented, are often ill-defined and exhibit large numbers of exceptions. Furthermore, they expand in scope and evolve in time, and thus suffer from constantly changing requirements and assumptions. We are therefore faced with an ill-structured application that is not clearly-defined.

Object-oriented techniques are especially suited to deal with such applications. Each object is well-defined, but we are not expected to start with a definition of an overall object population behaviour. We can change object behaviour in time and we can add new objects. We have an environment where the whole is represented by the behaviour of the parts. We claim that the object-oriented approach is well-suited for overall application development. In addition, traditional approaches, such as structured programming, cannot cope with applications like office information systems. Hence, for such applications, object-oriented techniques are not only important but necessary.

The particular needs of office information systems can be divided into the following main subtopics:

1. office modeling concepts
2. application development
3. application maintainability
4. user interface issues

The modeling issues are: identification, modeling and representation of "office objects", formulation of routine office procedures, encapsulation of (non-routine, or semi-routine) office tasks, and encapsulation of office knowledge.

Application development necessitates better languages, better run-time support and better tools for developing applications (such as object design tools).

Application maintainability not only refers to routine maintenance but also to the ability to easily extend applications, to allow them to migrate to new environments (i.e., "porting"), and to allow different applications to communicate with one another.

The user interface issues are diverse, but some of the most important include the representation of system entities to the user, natural paradigms for interacting with and manipulating those entities, and "standard" interfaces across a range of applications on a per-user basis.

We shall focus on each of these issues and show how object-oriented techniques have contributed to software solutions. We shall argue that the adoption of object-oriented languages, systems and tools is essential for developers of integrated office systems.

It is clearly possible to develop applications in an object-oriented fashion without using any object-oriented tools, programming constructs or run-time support. However, the extent to which an application developer may exploit these techniques will be severely limited if these tools are not available, as we hope to demonstrate.

1 Office Modeling Concepts

The initial phase of application development involves modeling. Modeling is by definition a process of abstraction, since the point of a model is that one be able to concentrate on "important" properties and ignore the "unimportant" ones. Similarly, application development is a process of abstraction, except that the "important" properties are those that the user of the system will be aware of, and the "unimportant" ones are the implementations of the software components. Object-oriented approaches help us in this abstraction process by encouraging us to decompose problems into semi-independent objects.

There are various object models in use today, but the one that has the most relevance for applications programming is that which views objects as instances of abstract data types. In this model, the abstract interface to an object is a set of named operations that clients of that object may invoke. The static data (i.e., the *instance variables*) and the implementations of the operations (the *methods*) are hidden. The specification of the object's interface may include the invocation syntax of the operations, the argument and return types, and possibly other static properties of the object or its operations (for example, a formal or informal description of the semantics of an operation invocation).

This particular object model will support object-oriented techniques such as instantiation, parameterization and class inheritance, to mention a few. There are also object models that are useful for capturing other aspects of office information systems (objects as part hierarchies; objects as sets of rules or facts), which we shall touch on in the discussion that follows. In one sense, however, the abstract data type object model is the most general, since we can always access the static properties of other object models through an operational interface.

1.1 Office objects

It has been claimed that the best way to computerize the office is to *begin* by mimicking office functions in software [Ellis and Nutt 1980]. Not only does this provide application developers with a (relatively) well-defined initial goal, in terms of simulating the real objects of the office environment and their functionality in software, but it supposedly eases the cost of training office personnel to use the software, since they will be initially introduced to as few new concepts as necessary.

Furthermore, it can hardly be disputed that people are used to working in terms of manipulating and coordinating "objects". The objects we see in the office are mainly encapsulations of information: forms, letters, records, databases, reference manuals, and so on. The nature of the information they contain may vary a great deal, however. Many of the office "objects" encapsulate not only information but also methods for its manipulation.

Computerized office systems initially concentrated on encapsulating well-structured, routine entities, such as forms and records. Database management systems have proven invaluable for managing large numbers of similarly structured entities. Electronic forms systems exploit the regular structure of forms to provide forms-oriented interfaces to database systems [Tsichritzis et al. 1982; Tsichritzis 1982].

Traditional database models were typically not object-oriented, however, since they exposed database records as passive entities without any indication of their intended use. It was up to the application software to decide how the database was to be used. When new applications were built, there was no implicit guarantee that the database would continue to be used in a consistent fashion, since there was no mechanism for abstracting the proper usage of the database entities. This problem was remedied to some extent by the introduction of *database constraints* and *triggers*, which provided a way of discovering when things went wrong. They stopped short, however, of providing a way to specify what operations were valid to perform on objects. Current work on object-oriented databases attempts to remedy this shortcoming (see the subsequent chapters in this book).

Several object-oriented techniques are appropriate for the specification of electronic forms. The most obvious are classification and instantiation, since that is how paper forms are defined. Similarly, data abstraction is appropriate for parts of forms, since there are distinct categories of form fields that behave in very different ways [Gehani 1982]. Some examples are:

- optional, modifiable fields
- obligatory, non-modifiable fields
- automatic fields
- "virtual" fields

An example of an obligatory, non-modifiable field is an employee number. An automatic field, for example, the form creation date, is filled in by the system and stored with the rest of the form. Virtual fields, on the other hand, require no storage, being re-computed whenever they are displayed. Prices and totals on an order form might be handled in this way. Forms can be defined as conglomerations of fields of various field types, where the permissible set of field types is extendible.

Inheritance can play an important role in designing new forms, since many forms share common parts. For example, we can define a basic order form that provides certain standard fields required by, say, the accounting department. We may then define new order forms that inherit these standard fields, but may include extra information used only locally by certain departments. This kind of inheritance between form types is analogous to class inheritance in object-oriented languages such as Smalltalk, if we interpret the fields of a form as its public instance variables and methods.

Other kinds of office objects can also be defined using these techniques. Document types are now commonly viewed as *part hierarchies* [Horak 1985; ECMA 1983; Furuta et al. 1982]. As with forms, one may define new types of document parts as need arises. Furthermore, these parts may be made up of other parts, for example, chapters are made up of sections and paragraphs, and diagrams are made up of graphical entities. Documents themselves are not normally made up of a static combination of these parts. Certain document types may exhibit some common structure; for example, standard business letters must contain certain parts in fixed positions, but the body of a document is frequently quite flexible. We can furthermore draw an analogy between document types and object classes, and correspondingly define forms of inheritance that permit document types to share part structures. The document structure can be defined either using a formal grammar, or equivalently by defining for each part how it may be expanded in terms of permissible sub-parts. New document types may be defined by inheriting and refining the structure of existing types.

Part inheritance is concerned with the sharing of parts or values between object instances. (Notice that class inheritance can be viewed as part inheritance applied to class objects.) We make use of part inheritance when, for example, we create new document instances that inherit (copy) text from existing documents. More interesting is the case where document types are determined implicitly by the contents of the document, as is the case with text-formatting languages in which contents and formatting statements are freely mixed: here we can dynamically change the "type" of a document by copying parts of existing documents to the one we are editing. This kind of activity is supported by WYSIWYG document editors that allow copying of parts between different documents, or reuse of an existing document as a template for a new one. The danger is that undisciplined use of part inheritance leads to a profusion of document types and a loss of regularity.

A third kind of inheritance that is very important for documents is what we shall call *scope inheritance*. With class inheritance, all future instances will have the properties inherited by their class. With part inheritance, only the current instance inherits parts from another object. With scope inheritance, certain properties of an object are determined by its current scope, or environment. When that scope changes, those properties of that object will implicitly change. For example, if a piece of text is moved from the body of a section into a quotation, its display characteristics (text width, points size, etc.) may change, although it is still the same object. The difference between these different kinds of inheritance is in the nature of the properties inherited, and when the inheritance occurs (see also chapter 1 of this book).

Documents can be seen as a generalization of the concept of a form, since their structure is more flexible. At some point the paradigm of a document takes over that of an object, because any object can be seen as a document if there is a way of presenting it and a way of "editing" it (i.e., interacting with it). This phenomenon is already apparent in the work done on "electronic books" and "hypertext" [Cook 1984; Meyrowitz 1986; Weyer and Borning 1985]. (A "document", it would seem, is anything that has words in it, and doesn't change too often, even if there are many different ways of looking at it!)

Finally, there is a set of office objects that are more dynamic than forms or documents, and can be classified as "office tools". These are objects that can be used to manipulate other objects. Some common examples are mailboxes, spreadsheets and editors. Here too is the potential for exploiting class inheritance. Tools depend on the fact that the objects they manipulate have certain common properties. A mailbox can hold anything, as long as it is a message. A generic editor can be used to edit any object that can be displayed, and understands the edit operations. The properties required by a tool can be encapsulated in the operations of an object class, and inherited by any class that is to be compatible with that tool. The implementation of those operations may vary, since objects may have different internal structure, but this should be transparent to a generic tool.

1.2 Office procedures

Thus far we have concentrated on passive office objects that have counterparts in the physical office. The operations that are valid for manipulating these objects may be encoded as part of their object classes, if an object-oriented approach is used. By *office procedures* we mean the routine sequences of operations that are used to manipulate office objects [Hogg et al. 1985].

Office procedures are also objects that office workers should be able to manipulate, since "routine" procedures may have a life span lasting anywhere from a few minutes to the lifetime of the office, and may be routine only for a single worker or for everybody in the company. Office workers should not, however, be expected to learn how to program in order to specify or use computerized office procedures.

Attempts to provide users with the ability to automate routine functions have mostly concentrated on "programming-by-example" techniques for encoding office procedures [Zloof 1982; Halbert 1984]. With this approach, one first builds a system containing electronic representations of office objects and functions. Such a system is then enhanced by permitting users to "teach" the system to perform routine functions by stepping through the procedures using examples. For example, an office worker who uses an electronic forms system to coordinate order forms and inventory forms can teach the system to notify him when matching forms appear by stepping through the procedure to find matching forms, and indicating the action to be taken. The advantage of this approach is that the user specifies procedures using the same interaction mechanisms as are used to manipulate the office objects manually. It is not

necessary to learn a completely new language.

Tools for specifying office procedures have also focused on the event-driven behaviour of office work. Office procedures are typically *triggered* upon completion of some awaited event: the arrival of a message, the completion of a form, the modification of a document. The by-example approach can be used to specify triggers by allowing the user to give an example of a situation that will trigger the procedure. Alternatively, it is possible to describe the conditions using a language that models information flow using *augmented Petri nets* [Zisman 1978; Barron 1982] or other formalisms that decompose procedures into triggered steps, for example, finite automata or production rule systems.

Event-driven behaviour lends itself well to graphical representations (such as those commonly used to depict Petri nets and finite automata). This suggests that graphical programming may be a promising approach for enabling office workers to specify certain kinds of office procedures; for example, users could specify trigger conditions and actions, and see the local or global effects in terms of a graphical display of the resulting event-triggering behaviour. Semi-automatic processing and routing of forms in a large community of users could be depicted as a flow diagram induced by users' local office procedures. Users could then use the graphical representation to navigate through the network of interacting procedures and either modify the graph directly to establish different triggering scenarios, or use the graph to access the procedures in a different form. The graph would be dynamically updated by analysing the interaction between procedures, using techniques such as those described in [Nierstrasz 1985].

1.3 Office tasks and office knowledge

Whereas routine office procedures are triggered under well-defined circumstances and result in specific actions to be taken, *office tasks* are goal-oriented and cannot necessarily be encoded according to a precise procedure to be followed. Examples of tasks as opposed to procedures are managing a meeting, or developing a new product.

The execution of an office task typically consists of many phases during which office objects are manipulated according to routine, mechanizable procedures, or according to non-routine, knowledge-intensive procedures. Either kind of procedure is event-driven, but only the former may be fully automated. The latter, while not mechanizable, can benefit from triggering (e.g., by notifying office workers what work is ready to be done next), and can benefit from decision support tools [Bui and Jarke 1986] and from knowledge-based approaches [Croft and Lefkowitz 1988].

Encapsulation techniques are important for decomposing office knowledge. Conceptual modeling techniques have traditionally decomposed knowledge into object-like entities, and have found it useful to apply such notions as instantiation, classification, structuring (aggregation) and inheritance [Mylopoulos and Levesque 1983]. Similarly, rule-based approaches for encod-

ing knowledge using languages like Prolog have benefited from the object-oriented paradigm by encapsulating knowledge or beliefs inside objects (see [Russinoff 1988] in this book). Conflicting theories can thus be tolerated as long as knowledge objects do not interfere with one another.

"KNOs" are a programming paradigm for encapsulating knowledge and goals as knowledge objects [Tsichritzis et al. 1987]. The principle idea is that KNOs are mobile, active entities that can gather information from their environment, and can exchange rules and knowledge with other KNOs. A system for defining KNOs [Casais 1988] has been implemented using the Lisp flavors package [Moon 1986]. In this system, KNOs belong to one or more KNO classes, and are able to dynamically inherit parts while they execute. The operations of a KNO are encoded as production rules, each consisting of a name, a trigger condition, and a series of actions. The actions may be used to modify a KNO's state, to communicate with other KNOs via an intermediate *blackboard* object, create new KNOs, move to another environment, learn and "unlearn" rules, and create and communicate with *limbs* (subordinate KNOs that may be distributed to other environments).

2 Application Development

Object-oriented techniques and languages help to cut down the cost of developing software in many ways. The most significant way in which they do so is in providing the possibility of a large software base of well-designed, pre-packaged, reusable objects. (For a survey on software reusability, see either [Biggerstaff and Perliss 1984] or [Biggerstaff and Richter 1987].) Object-oriented languages not only adopt the old idea of reusable libraries of subroutines, but they extend the ways in which software can be made reusable by supporting mechanisms such as object instantiation, operator overloading (i.e., "tailoring" of object classes), run-time binding (when appropriate), class inheritance, and parameterization.

A programmer who is familiar with the reusable object base can, in principle, configure a "prototype" application in a fraction of the time it would take to program from scratch, with a minimum of programming effort. Furthermore, it has been shown that object-oriented techniques and mechanisms are in no way inconsistent with strong (static) typing, or with efficient compilation (rather than interpretation). Thus the commonly-held view that objects are "only" good for prototyping is misleading, since these prototypes by no means have to exhibit poor performance.

Despite the clear advantages that object-oriented programming offers the application developer, there are several important issues to be addressed before one can start programming with objects. Fundamentally, of course, we must have an object-oriented programming language available for our use.

This immediately introduces an integration problem. We would like to be able to reuse

existing software and applications that were developed without the benefit of an object-oriented methodology, and integrate them with newly-developed applications. We rarely have the luxury of being able to choose our environment when we have to build an application. Fortunately, there are several possible solutions to this problem, all of which have to do with encapsulation: since the implementation of an object is hidden from us, there is no reason the internal environment of an object must be the same as the environment in which it is used. Encapsulation may therefore be used to hide the implementation language. The practical alternatives are:

1. use an object-oriented extension of an existing programming language (such as C++ [Stroustrup 1986], Objective C [Cox 1986], Lisp flavors [Moon 1986])
2. use an object-oriented language that is translated into the target programming language (such as Eiffel, which is translated into C [Meyer 1986])
3. use an object-oriented language augmented with a package for interfacing to other languages within the programming environment (e.g., calling Fortran libraries from C++, or whatever)

The other fundamental issue is that of run-time support for the object environment. This may entail support for:

- object naming
- persistence
- distribution (e.g., remote communication)
- concurrency
- transaction management
- version management (for objects and object classes)
- security

An environment that provides basic run-time support for objects at this level allows the application developer to concentrate on the configuration of the objects in the intended application, and permits standard solutions to these low-level problems to be adopted across a range of applications.

These fundamental issues aside, the application developer has to solve two difficult problems posed by the object-oriented paradigm:

1. What are the objects of my application?

2. What objects will help me to implement it?

The first is the traditional "decomposition" problem: How can I break my problem down into manageable pieces? The second is the composition problem: What tools are available to me to solve my subproblems? In addition, we have the traditional problems of managing large pieces of software, i.e., version management and software distribution management.

Obviously, there is a certain amount of "black magic" involved in properly decomposing a problem into objects. Even though we may intuitively know what objects we have to deal with, it is often difficult to decide what should be a "property", a "part" or an "acquaintance", just as it is difficult to decide what should be visible and what should be hidden. We call this the *object design problem.*

Techniques for designing "objects" in other areas, such as database schema design and computer-aided design, do not have to deal with the full range of techniques and mechanisms offered by object-oriented languages.

The composition problem is currently answered only through "browsers" that help the programmer to browse through the base of object classes, and through programmer familiarity with the set of reusable object classes available to him. Both of these solutions are inadequate for dealing with extremely large collections of object classes, say in the tens of thousands. The obvious approach of putting the object classes in a database fails because the data are highly dissimilar. What kind of queries would one pose of such a database?

Simply grouping object classes according to subject matter to aid browsing is also inadequate since there are many possible groupings. This is comparable to the problem of browsing through an encyclopedia to find the answer to a loosely-defined question. An "electronic encyclopedia" approach can only work if we not only know where to *start* browsing, but we can also be sure that the appropriate links exist for us to follow. Knowing which object classes may be useful for solving a particular problem is a good example of "expert" knowledge that can only be acquired through experience. Even the original implementor may not be able to anticipate all the uses to which his objects may reasonably be put. This suggests that knowledge about object usage should be incrementally added to the "object class encyclopedia", based on the experience of programmers who have actually used those classes to build applications. Furthermore, we expect that an expert system interface would be the most natural way to pose queries that can be used to direct the browsing task.

There is a final bit of "magic" we have avoided up to now, namely, how do objects become reusable? It should be obvious that one can hardly sit down and design a new object class that will be instantly useful to a host of applications. Rather, we expect that application developers will discover that certain objects that have been tailored for particular applications can, with some careful redesign, be generalized and made useful to other problem domains. Object classes that evolve in this way gradually become more generic and enter the software base of reusable objects. This suggests that our object encyclopedia should not only manage

tried and proven object classes, but that it should keep track of evolving classes, and help us to detect and forge software generality.

3 Application Maintainability

Unfortunately, once we have developed software, the job is far from done. Aside from the tedious work of testing and debugging, we must maintain software because the conditions and requirements for its use change. The application may need to be extended, or ported to a new machine, or modified to solve a slightly different problem, or integrated with somebody else's software, or even hacked apart so that certain parts can be reused. In the field of office information systems the issue of constantly changing requirements is ever more pressing, as software is being used to address more complex and ill-defined problems.

Object-oriented techniques provide solutions to some, but not all, of these problems. Provided that objects' specifications do not change, modification of their internal behaviour should not affect the context in which they are used. For example, porting an application to a new environment should only entail re-implementation of the "machine-dependent" objects.

Encapsulation also provides the possibility of reusability of objects: since objects are "self-contained", it is in principle possible to take them out of one environment and reuse them in another. As we have pointed out, however, it is rarely true that objects designed for one application will be automatically useful for another. Such objects typically evolve through several iterations before they stabilize and acquire a level of generic usefulness. Nevertheless, we believe that encapsulation makes it easier to isolate and evolve generic software.

Adding functionality to an application is often possible through the mechanism of class inheritance. This works only if one can capture this increased functionality by strictly adding on to the objects that already exist in an application. Parts of the application that knew instances of the old parent class will continue to work with the instances of the subclass, and it is even possible to allow instances of both classes to co-exist. For example, a forms system may be upgraded to deal with multimedia forms. The old interface will be able to cope with both kinds of forms, though it will not be able to make use of the new functionality. New operations may, if necessary, violate encapsulation, and have direct access to the static data of the originally defined object, but this may cause difficulties if we later decide to independently evolve the old class, or change its implementation.

Object encapsulation can also be extremely important for integrating applications, if we realize that the object specification language need not be the same as the implementation language. That is, one could conceivably use an object-oriented language to describe encapsulations (the interfaces to objects), and feel free to use other languages in their implementations. There are two important reasons for doing this. First, we may have already invested

a great deal in software that is not written in a particular object-oriented language. Repackaging software in terms of objects is an exercise in defining its valid external interfaces, and in identifying its reusable components. Second, we may find that different languages are more appropriate for implementing different kinds of objects. If we believe that a carpenter should use the best tool for a given task, then it is reasonable to allow a programmer to use the best language for solving a problem. For example, we can imagine an object-oriented interface to a "knowledge object" programmed internally in Prolog.

4 User Interfaces

As computer software is used to accomplish more abstract and complex tasks, the user interfaces to these systems must become correspondingly more sophisticated. The problem is how to provide the user with the most reasonable presentation of the state of the system and its functionalities. Fortunately, as more cycles become available for more complicated computer systems, we can skim off more cycles for more sophisticated interfaces.

Objects provide us not only with a useful paradigm for organizing software, but they also suggest a corresponding interface paradigm: for every software object in the system that we wish to represent or interact with, there can be a corresponding "presentation object" that will be visible to the user on a display. Furthermore, the paradigm of *direct manipulation* will allow the user to "directly" interact with objects by issuing commands to one or more "current" presentation objects that can provide visual feedback regarding resultant side-effects. We see these ideas work not only in document editor-formatters, but also in interfaces with virtual sliders, buttons, levers and so on.

Task switching in a direct-manipulation environment is possible just as it is in the real world: we simply shift our attention from one set of objects to another by, for example, moving between windows. With command-style interfaces the current "mode" is often implicit. We must remember whether we are in *insert* or *delete* mode, or which directory we are in. By contrast, the current context is always explicitly presented in a direct-manipulation interface.

The main difficulty with this approach is that it requires a fair bit of creativity to decide on good paradigms for presentation and for manipulation. How should we represent a mailbox? What feedback should we get when incoming mail is received, or when outgoing mail is delivered? What interaction paradigm is appropriate for performing a database query? (Are command-style interaction paradigms inevitably best for some kinds of tasks, or do we just lack the intuition and insight we need to package them in terms of direct manipulation?)

Another important difficulty is that we must decide between richness and uniformity. A rich user interface is pleasant if it allows us to perform a large number of complex tasks with minimal effort, but it may be more of a nuisance if it takes a long time to learn, and if it is inconsistent with interfaces to other applications we regularly use. A powerful tool is

severely undermined if we can only apply it with a users' manual constantly open on our desk.

Can we use object-oriented techniques to help us build user interfaces to object-oriented applications? If it is truly possible to separate an application from its user interface, then it should be possible to configure an interface for an application after the fact by selecting reusable presentation and interaction objects from a library [Myers 1987]. In this way, applications could be tailored by user interface designers to apply a consistent set of rules across an environment of users. It remains to be seen whether the idea of user interface management systems [Buxton et al. 1983; Foley 1986] can be given new life in the context of object-oriented programming.

Summary

There are very good traditional reasons for exploiting object-oriented approaches in the development of office information systems. Software reusability, maintainability and reliability are the main contributions in the domain of software engineering. Furthermore, the notion of software objects is quite close to that of modeling of office objects. Languages that permit us to directly describe the properties of the objects manipulated by our applications can only help us build our software more quickly and reliably. Certain object models can also make it easier for us to deal with other issues that are important for office information systems, such as concurrency, distribution and persistence. Objects' well-defined boundaries help us to understand exactly *what* must be concurrent or distributed or persistent, and therefore help us to build systems with low-level support for objects that have these properties.

Object-oriented techniques, however, are not only important for developing office system applications, they are *essential* for the success of integrated office systems, as they are for any open, evolving, distributed application domain. Encapsulation is necessary for evolution since it is the fundamental mechanism for software reusability. All other object-oriented mechanisms depend on it (such as inheritance, parameterization, polymorphism, etc.). Incremental development, changing requirements and integration of disparate applications may be difficult to handle in an object-oriented world, but they are next to impossible without it.

We are discovering that the traditional software cycle of requirements analysis, application specification, programming and debugging, and installation, is not appropriate for today's vaguely-defined, ill-structured, rapidly changing applications. Office workers are known for their ability to adapt to day-to-day alterations in loosely-defined office procedures. If we believe that office information systems will be able to provide office workers with meaningful support, then our application systems should be similarly flexible, adaptable, reusable, and amenable to evolution.

Objects encapsulate a locally well-defined behaviour, but make few assumptions about their global context. Object applications can evolve because their behaviour is determined implicitly by the object populations that implement them. When objects are added, modified

or combined, the global behaviour of the system will change as a matter of course.

If these techniques are to pay off, then we must be prepared to evolve our old programming languages, software environments and operating systems. Just as one may employ structured programming techniques when programming in assembler, one may apply object-oriented techniques in our antiquated software environments. On the other hand, it should be clear that we will not be able to take full advantage of objects until our software environments are themselves object-oriented from top to bottom.

References

[Barron 1982] J.L. Barron, "Dialogue and Process Design for Interactive Information Systems using Taxis", Proceedings ACM SIGOA, pp. 12-20, Philadelphia, June 1982.

[Biggerstaff and Perliss 1984] T.J. Biggerstaff and A.J. Perliss (ed.), "Special Issue on Reusability", IEEE Transactions on Software Engineering, vol. SE-10, no. 5, Sept 1984.

[Biggerstaff and Richter 1987] T.J. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions", IEEE Software, vol. 4, no. 2, pp. 41-49, March 1987.

[Bui and Jarke 1986] T.X. Bui and M. Jarke, "Communications Design for Co-oP: A Group Decision Support System", ACM TOOIS, vol. 4, no. 2, pp. 81-103, April 1986.

[Buxton, et al. 1983] W. Buxton, M.R. Lamb, D. Sherman and K.C. Smith, "Towards a Comprehensive User Interface Management System", Computer Graphics, vol. 17, no. 3, pp. 35-42, July 1983.

[Casais 1988] E. Casais, "An Object-Oriented System Implementing KNOs", Proceedings of the Conference on Office Information Systems (COIS), pp. 284-290, Palo Alto, March 1988.

[Cook 1984] P.R. Cook, "Electronic Encyclopedias", Byte, pp. 151-167, July 1984.

[Cox 1986] B.J. Cox, *Object Oriented Programming – An Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.

[Croft and Lefkowitz 1988] W.B. Croft and L.S. Lefkowitz, "Using a Planner to Support Office Work", Proceedings of the Conference on Office Information Systems, pp. 55-62, Palo Alto, CA, 1988.

[ECMA 1983] ECMA, "Office Document Architecture", TC 29/83/56, Fourth Working Draft, 1983.

[Ellis and Nutt 1980] C.A. Ellis and G. Nutt, "Computer Science and Office Information Systems", ACM Computing Surveys, vol. 12, no. 1, pp. 27-60, March 1980.

[Foley 1986] J. Foley (ed.), "Special Issues on User Interface Software", ACM Transactions on Graphics, vol. 5, no. 2-4, 1986.

[Furuta, et al. 1982] R. Furuta, J. Scofield and A. Shaw, "Document Formatting Systems: Survey, Concepts and Issues", ACM Computing Surveys, vol. 14, no. 3, pp. 417-472, Sept 1982.

[Gehani 1982] N. Gehani, "The Potential of Forms in Office Automation", IEEE Transactions on Communications, vol. Com-30, no. 1, pp. 120-125, Jan 1982.

[Goldberg and Robson 1983] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, May 1983.

[Halbert 1984] D.C. Halbert, "Programming by Example", Ph.D. Thesis, Dept. of EE and CS, University of California, Berkeley CA, 1984.

[Hogg, et al. 1985] J. Hogg, O.M. Nierstrasz and D.C. Tsichritzis, "Office Procedures", in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, pp. 137-166, Springer Verlag, Heidelberg, 1985.

[Horak 1985] W. Horak, "Office Document Architecture and Office Document Interchange Formats: Current Status of International Standardization", IEEE Computer, vol. 18, no. 10, pp. 50-60, October 1985.

[Meyer 1986] B. Meyer, "Genericity versus Inheritance", ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 391-405, Nov 1986.

[Meyrowitz 1986] N. Meyrowitz, "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework", ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 186-201, Nov 1986.

[Moon 1986] D.A. Moon, "Object-Oriented Programming with Flavors", ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, pp. 1-8, Nov 1986.

[Myers 1987] B.A. Myers, "Creating User Interfaces by Demonstration", Ph.D. thesis, CSRI Technical Report #196, Department of Computer Science, University of Toronto, May 1987.

[Mylopoulos and Levesque 1983] J. Mylopoulos and H. Levesque, "An Overview of Knowledge Representation", in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, ed. M. Brodie, J. Mylopoulos, pp. 3-17, Springer-Verlag, New York, 1983.

[Nierstrasz 1985] O.M. Nierstrasz, "Message Flow Analysis", in *Office Automation: Concepts and Tools*, ed. D.C. Tsichritzis, pp. 283-314, Springer Verlag, Heidelberg, 1985.

[Nierstrasz 1988] O.M. Nierstrasz, "A Survey of Object-Oriented Concepts", in *Object-Oriented Concepts, Applications and Databases*, ed. W. Kim and F. Lochovsky, Addison-Wesley, 1988.

[Russinoff 1988] D. Russinoff, "Proteus: a Frame-based Non-monotonic Inference System", in *Object-Oriented Concepts, Applications and Databases*, ed. W. Kim and F. Lochovsky, Addison-Wesley, 1988.

[Stroustrup 1986] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[Tsichritzis, et al. 1982] D.C. Tsichritzis, F. Rabitti, S.J. Gibbs, O.M. Nierstrasz and J. Hogg, "A System for Managing Structured Messages", IEEE Transactions on Communications, vol. Com-30, no. 1, pp. 66-73, Jan 1982.

[Tsichritzis 1982] D.C. Tsichritzis, "Form Management", CACM, vol. 25, no. 7, pp. 453-478, July 1982.

[Tsichritzis, et al. 1987] D.C. Tsichritzis, E. Fiume, S. Gibbs and O.M. Nierstrasz, "KNOs: KNowledge Acquisition, Dissemination and Manipulation Objects", ACM TOOIS, vol. 5, no. 1, pp. 96-112, Jan 1987.

[Weyer and Borning 1985] S.A. Weyer and A.H. Borning, "A Prototype Electronic Encyclopedia", ACM TOOIS, vol. 3, no. 1, pp. 63-88, Jan 1985.

[Zisman 1978] M. Zisman, "Use of Production Systems for Modelling Asynchronous Concurrent Processes", in *Pattern-Directed Inference Systems*, pp. 53-68, Academic Press, 1978.

[Zloof 1982] M.M. Zloof, "Office-by-Example: A Business Language that Unifies Data and Word Processing and Electronic Mail", IBM System Journal, vol. 21, no. 3, pp. 272-304, 1982.