Application Development Using Objects[1]

D.C. Tsichritzis
O.M. Nierstrasz

Abstract

Much of the cost of developing and maintaining applications can be attributed to our disposition to build systems largely from scratch. An application development support system would shift the emphasis from programming of arbitrary systems to *configuration* of certain classes of applications from pre-packaged software. In order for this style of application development to be successful, we argue that it should be carried out in an object-oriented software environment. Such an environment would consist of an object-oriented language and system that integrates various object-oriented approaches, user interface tools for monitoring and interacting with active objects, object design tools, and support for evolving application-oriented objects.

1 Introduction

An *Application Development Support System* (ADSS) is an environment of software tools for designing, configuring and maintaining application systems. For example, an application designer might use an ADSS to describe the components of a business application and the relationships between them. The ADSS should guide the designer in locating and configuring the pre-packaged components that will comprise the application, and, in the case of extraordinary design requirements, aid in the specification and development of the custom-made components. Once an application has been configured, the ADSS will continue to support the designer, who will have to deal with evolutionary changes to the application and with integration with other applications.

Application programming consumes a great deal of effort but it receives little attention in terms of scientific research. A number of tools have been developed which give assistance to application developers, such as high-level languages, program editors, database management systems, etc. Similarly, a number of methodologies have been proposed and applied that supposedly make the task better organized and more efficient, for example, structured design, structured programming, requirements specification models, etc. The problem, however, still remains: application programming takes a great deal of effort and involves delays. In addition, it seems that we are running out of well-defined and nicely structured applications. Many applications that we need to develop are ill-defined, evolving, complex and distributed. There are two important reasons, therefore, to re-evaluate our methods. First, to increase the productivity of our current effort. Second, to enable us to attack new, important and exciting applications.

Object-oriented languages and systems have been proposed as a promising new paradigm for software development [Cox '86]. Before we apply such methods, however, we should look more carefully at the production cycle of application development.

[1]In *Information Technology for Organisational Systems, Proceedings EURINFO '88*, ed. H.−J. Bullinger et al., pp. 15−23, Elsevier Science Publishers B.V. (North−Holland), 1988. Also invited talk in *Conférence Européene, Entreprise et Communication*, and CRAI Workshop on Object-Oriented Programming.

Most existing tools and methods for application development make the implicit assumption that effort proceeds through some well-known steps (i.e., the waterfall model of software development). Starting with some much abused "reality", there is an analysis of requirements. Then we proceed to some specifications and system design. Programming, documentation and integration follow. Then we have testing and validation. Finally, the manuals and the end-user support are developed. Each tool at our disposal is supposed to help us in at least one of these steps. We do not question the usefulness of the tools. However, there are still problems with application development. The following are a series of arguments that demonstrate that this well-known series of steps presents problems:

1. The so-called "reality" is often a moving target. We cannot get it right because we do not understand it or are unable to specify it. We cannot properly describe it because by the time we are through the development cycle it has changed.

2. For some applications the entities, the structure, and sometimes the goals of the system cannot be defined a priori.

3. Apart from some general tools and program libraries, there is a tendency to develop most of the programs from scratch. No wonder it takes so much effort and entails delays.

4. There is complete freedom in the design and development effort. As a result it may be very exciting, but the absence of design constraints leads to overdesign and costly overruns.

5. There is a feeling that the system will have a long and relatively stable life. There is a tendency, therefore, to make it rigid and efficient. In many cases it would have been better to make it flexible even at the cost of efficiency.

In general we approach application development the way Michaelangelo approached the Sistine Chapel. We want to do everything great, perfect and from scratch, and leave it for the admiration of future generations. Unfortunately most application systems do not have the longevity of great works of art. We seem, therefore to waste creative talent which is in short supply throughout the ages.

Fortunately many so-called "naive" users do application programming in a different way. For them the process is straightforward. Once they understand the requirements, they search the magazines for appropriate packages. They put them together with whatever "glue" is provided by the operating system. They prototype the system quickly and they evaluate whether it is useful. If not, they tinker and change it until they get it right. We suggest that application development could proceed in an analogous way.

We assume that a successful ADSS must make heavy use of pre-packaged, reusable software. That is, we believe that the task of developing new applications with an ADSS should consist primarily of *configuration*, and only marginally of programming. We do not assume that the software base need be small and fixed. It is clear, however, that from a given software base, one can only generate certain classes of applications without additional programming effort. As a consequence, we believe it imperative that an ADSS should be initially targeted towards the production of a well-defined *subset* of some interesting class of applications. Additionally, the applications to be generated by an ADSS are assumed to be themselves evolving, *open systems*. Applications must be able to evolve quickly to keep up with the changing needs of an organization, and to adapt to iterative modification to the hardware and software environment.

We distinguish between *application-specific* aspects of an ADSS, such as the application modeling and configuration tools and the application software base, and the *environmental* aspects. If we consider the most pressing environmental issues that an ADSS must cope with, we see that they are largely addressed by object-oriented techniques. These issues include reusability, concurrency, distribution and open systems. We therefore propose that an ADSS should be based on an object-oriented software environment.

2 Object-Oriented Software Environment

The fundamental idea of the object-oriented paradigm is *encapsulation*, that is, the partitioning of systems into *objects* with an visible abstract interface and a hidden realization. Object-oriented systems exploit encapsulation primarily to achieve high reusability of software. Encapsulation has also been demonstrated to be useful for reliability (through strong-typing), software maintainability, concurrency and distribution (active objects, etc.), and security (via capabilities), to mention a few areas [Nierstrasz '88]. An object-oriented software environment makes these techniques available to the application developer by providing programming language and run-time support for objects, a software base, and software management tools. For such an environment to be a successful aid to application development, there must also be tools for developing user interfaces, for designing and configuring new objects, and for coordinating and combining objects to solve more complex tasks.

An object-oriented software environment for an ADSS would consist of programming language and run-time support, a software base, and software management tools. These components are concerned with basic programming issues and may be shared from one ADSS to another. There are also several application-oriented environmental issues that can be addressed generically. These include user interface support, object design tools and support for object evolution.

The proposed application development cycle is summarized in steps in Table I. Notice that the effort does not have a clear beginning and an end. It is a continuous cycle where the last step feeds back to the first. For comparison, we also outline the steps that "hackers" usually take to develop prototypes. This analogy points out that we are not proposing something very radical. We are simply trying to raise hacking to a level of respectability.

ADSS	**Hacking**
• select objects from software base	• get packages from network
• modify them	• hack them
• configure via scripts	• link with shell commands
• monitor behaviour	• prototype & test
• evolve software base	• distribute on the network

Table I: Application development cycles

In the following sections we shall discuss:

1. exploitation and integration of established and emerging object-oriented techniques in an object-oriented programming language and system
2. user interface tools for visualisation, monitoring and interaction with active object-oriented applications

3. tools for the design and configuration of evolving object-oriented software bases

4. advanced active objects for cooperative, distributed and evolving applications.

3 Object-oriented language and support

Hybrid is a prototype language and system that integrates a number of object-oriented paradigms and language constructs [Nierstrasz '87]. In particular, established mechanisms for achieving reusable object code have been integrated with an object-oriented model of concurrency and an extendible type system for objects. An initial language design has been completed and a prototype implementation is under way [Konstantas et al. '88].

There are, additionally, several important directions in which Hybrid can be further developed, notably that of support for open, evolving systems. The key idea is that an object-oriented language is not just a language for programming, but fundamentally a language for *packaging*. When properly applied, this principle can put a great deal of functionality in the hands of an application developer, without imposing inconvenient constraints or overhead. We shall discuss, in the context of Hybrid, the research issues concerned with the integration of object-oriented approaches.

3.1 Reusability

Object-oriented languages have traditionally emphasized mechanisms for achieving software reusability. In this way objects refine the idea of a library or package. Reusability can be enhanced in many ways:

1. Instantiation: multiple objects can be statically or dynamically created from either an *object class* description, or from a "prototypical object"

2. Structuring: *instance variables*, that is, the hidden data of an object instance, may themselves be object instances

3. Class inheritance: internal structure (instance variables) and the implementation of operations (*methods*) may be shared between object classes

4. Overloading: the realizations of outwardly similar object classes may be transparently altered, thus permitting greater software independence

5. Parameterization: generic "container" objects and tool objects can be applied uniformly to different targets

Other forms of inheritance exist to implement different kinds of reusability: multiple inheritance, part inheritance, scope inheritance, "delegation", etc. [Wegner '87]. Different approaches provide the programmer with more or less control over the forms of reusability, depending on whether the emphasis is on efficiency and reliability, or on flexibility and dynamic change.

Furthermore, some of these mechanisms require software management overhead, for example, some schemes for multiple inheritance permit subclasses to define new operations that may access inherited instance variables, thus "violating encapsulation" of the parent class: the implementation

of the parent class may then no longer be altered without affecting its children. In other cases, it may be necessary for objects to have different encapsulations depending on the context in which they are used. As an extreme example, a debugger may need to see all of an object's "hidden" variables.

If we believe that a programmer should have the best tools available to solve a problem as quickly and efficiently as possible, then we must carefully evaluate how various object-oriented mechanisms for software reusability can be made available in a consistent and manageable fashion. For example, we may need to develop language mechanisms (or software tools) that permit experimentation and flexible re-configuration during the development phase of an application, yet provide the option of customizing when the configuration stabilizes. This is analogous to "compiling" a prototype into a production system when it is no longer necessary to be able to interactively modify it.

3.2 Concurrency

The potential that object-oriented approaches offer for the programming of concurrent systems has not been emphasized until now by most object-oriented languages. As in traditional programming languages, there are basically two approaches to programming concurrent applications:

1. active entities synchronize and communicate
   through shared passive objects
2. active entities communicate with one another directly by message-passing.

Both solutions have their advantages and disadvantages in an object-oriented framework, though there is some room for exploiting the best of what both have to offer. The Hybrid language supports a concurrent object model that integrates these two approaches. The basis of the Hybrid approach is to model all computation as interaction between (potentially) active agents that communicate by remote procedure calls. Whether actual message-passing occurs depends partly on decisions that can be made by the compiler, and partly by the run-time system.

In addition, scheduling and interleaving of threads of control are enabled by *delay queues* and *delegation* [Nierstrasz '87]. These mechanisms are powerful and general enough to solve interesting problems, such as resource management (e.g., for bounded buffers etc.), administration of parallel subtasks, triggering and constraints. On the other hand, delegation and delay queues can be used in a unstructured fashion, resulting in code that may be hard to understand or modify. Higher-level mechanisms either packaged as objects (e.g., triggers) or as language constructs (e.g., transactions) are therefore desirable in situations where the demand is high enough. Language constructs for managing nested transactions (as in Argus [Liskov '83]) and for managing bundles of related threads of control or "subactivities" are consistent with the Hybrid activity model, and will be included in subsequent versions of the language.

3.3 Object types

Object-oriented languages can be either typed or untyped. In the untyped case (Smalltalk [Goldberg & Robson '83], or Lisp with flavors [Moon '86]), one may send any message to an object: it is the object itself that decides at run-time whether it can handle the message, and, if so, how. In a sense, every object performs its own type-checking at run-time.

In the typed case, instance variables are typed, and the language guarantees that a variable can never be bound to an instance whose type does not conform to that of the variable. In a statically-typed language, the compiler will also be able to determine the methods to be invoked at compile-time. When dynamic-binding is permitted, the compiler can only verify that an operation exists, but will not be able to bind it to an implementation. Some form of method lookup table must then be maintained for each object class, and instances must be responsible for remembering their class. It is possible to mix these two approaches, as in C++ [Stroustrup '86].

Type systems for objects must cope with object-oriented reusability mechanisms. For example, in the presence of class inheritance or polymorphism, two objects may be of equivalent types, or one may be of a subtype of the other's type, even though their representations may differ. Viewed in this way, an object type is effectively a predicate describing certain static properties of object classes: A subtype *conforms* to a parent type if its properties necessarily include those of the parent. Two types are equivalent if they are subtypes of one another. For example, in a type system that considers a type to be a set of operation names, a subtype will have at least the same set of operations as a parent and possibly more.

There is little agreement on what belongs in a type system for objects. Certainly operation names are important, but we must also capture the argument and return types of each operation if a compiler is to be able to type-check expressions. Side effects of operations and mutability of expressions and values may also be important. If exception-handling is present in a language, then the type system may need to formalize possible exceptions to be raised as a property of an object class, so that one may statically determine which exceptions are or are not being handled. Other interesting properties of objects may need to be formalized when different object models are used: concurrent behaviour, triggers, part hierarchies, etc. are all properties that are hard to capture if types are just sets of operations.

Furthermore, it has been demonstrated that encapsulation may need to be violated in cases where true software independence cannot be cleanly achieved. For example, in WYSIWYG user interfaces, it is necessary to be able to get "close" to an object in order to see its "hidden" representation. A realistic type system for objects should cope with the fact that an object's "type" may be different according to the context in which it is being used.

Once we have a suitable type system, there are several dynamic issues that are best handled by type-management tools. In an environment where object types are continually being defined, re-defined and re-implemented, it important to manage versions of types and the mappings between object instances and their realizations. If object classes are themselves objects (as in the Smalltalk system), and it is possible to change object classes within the running system, then there is the possibility of object instances being "orphaned" when their realization is altered or destroyed. The management of type dependencies, method lookup tables, shared source and executable, distribution of object classes in a network, type versions, and the mapping of instances to their realizations is the job of an object and type management system. Clearly, it is important to fulfill these managerial functions with a minimum of run-time overhead. Many of these issues are addressed in the implementation of the first Hybrid prototype [Konstantas et al. '88].

3.4 Open systems

We can identify several ways in which a system can be "open":

- portable: the less that is "hard-wired" in a system, the more easily it can be ported to a new environment
- extendible: it should be possible to add new functionality in a painless (i.e., modular) fashion
- modifiable: obsolete parts of a system should be removable or modifiable
- reusable: it should be possible to remove parts of a system for reuse elsewhere
- integratable: the entire system should be open to integration with other applications

If we accept object-oriented programming as a good basis for approaching the implementation of more open systems, then we are immediately faced with the problem of coping with heterogeneous systems, or multi-paradigm programming. Encapsulation is presently the only technique we have for integrating heterogeneous software and hardware: machines connected to other machines are typically made to think they are talking to a slightly strange I/O device; code written in one language using code written in another often does so through a procedure call interface. Since object-oriented programming languages provide more advanced encapsulation models, it is worth investigating how object-oriented techniques may be exploited to achieve the integration of heterogeneous systems.

The first step is to develop better object-oriented languages that integrate several existing object models (e.g., for object classes, typed objects and concurrent objects) and provide the programmer with more flexibility in applying object-oriented mechanisms for reusability, etc. Next, we must evaluate how we can apply these ideas to the encapsulation of software written in non-object-oriented languages, and how these languages may be extended to exploit these new possibilities (as has been done with Lisp [Moon '86], C [Cox '86, Stroustrup '86] and other languages). Finally, we must see whether "standard" object-oriented interfaces can be exploited across language barriers to simplify integration efforts.

3.5 Run-time support

Many object-oriented techniques cannot be fully exploited without some run-time support. The run-time support for Hybrid objects entails concurrency, distribution, object-naming and persistence.

In each case it is necessary to find some way of mapping these concepts from the object model to the host operating system. In effect, the run-time support provides an abstract machine layer on top of the operating system. Depending on what functionality is available at a lower level, this task may be straightforward or quite painful. For example, active objects can be messy to implement if there is no direct support for "light-weight" processes. Similarly, if there is no support for "memory-mapped" files or for persistent virtual memory, then one must implement persistent objects using the file system or a database management system.

Object-naming, especially in a distributed environment, may be partially supported by an operating system that allows communication between named processes over a network. With the proviso that objects be persistent, however, one may not establish a straightforward equivalence between long-lived object names and short-lived process names.

4 User interface support

Application designers are faced not only with the general problem of developing user interfaces for their applications, but also the more specific problem of *object visualisation and monitoring*. The latter problem is especially important in active, multi-user systems, where users may need to monitor the progress of their co-workers and of active (i.e., software) agents.

Since the cost of developing user interfaces can be quite high, short cuts are traditionally taken. First of all, interaction and presentation are usually abstracted into terms such as streams, windows, menus, etc., and packaged as part of the operating system kernel (i.e., device drivers) or as standard libraries. This kind of packaging of user interface primitives is entirely consistent with the object-oriented approach: these basic object classes would be part of a common software base.

Second, the development of user interfaces themselves can be partially automated by the use of user interface management systems [Buxton et al. '83, Pfaff '85]. Typically, however, these systems depend on the separability of the user interface from the application, and achieve rapid UI development by a "late binding" of the application to a fixed set of pre-packaged interaction components. In practice, however, it is often necessary to be able to get "close" to an application, especially real-time and direct manipulation applications [Hill '86]. Here too, object-oriented approaches may help, provided they acknowledge that encapsulation must be violated in certain cases. To achieve reusable applications, it is not so important to separate the user interface from the application, but to be able to separate objects *with* their interfaces from the rest of the application.

Object-oriented applications typically rely heavily on *direct manipulation* as the principle interaction paradigm. This is partly because well-designed direct manipulation interfaces can be easy to learn, and partly because it is very natural to represent the objects of the application visually to the user. We propose that visualisation aids be directly supported by the user presentation system. We believe that the ability to depict internal system operations using images, sound, and animation should be an integral part of the development environment.

The notion of depicting program behaviour by images and animation is not new (see [Barth '86] for a bibliography, as well as [Foley & McMath '86]). The novelty of our proposal lies in the specification language for expressing animated (and multimedia) activities. These activities can be triggered by internal system operations, which cause an appropriate visualisation of those operations to be displayed. We have already begun work on dynamic object visualisation [Fiume et al. '87, Fiume & Tsichritzis '87], and have found that the object-oriented approach is a useful way of encapsulating the visualisations corresponding to particular system operations. This encapsulation can hide issues such as the output device used, whether or not the animation is precomputed, and the nature of the graphical primitives used.

Our goal is to produce dynamic visualisations of objects, visualisations which cannot necessarily be computed beforehand due to the many possible paths that object execution may take.

To support the visualisation facilities as well as multimedia documents and animation, we have devised a language for specifying the temporal co-ordination of multimedia activities [Fiume et al. '87]. We have implemented several versions of this language, and have progressed from a prototype that simulated object-oriented primitives to an implementation in C++, an object-oriented language. We plan a migration of the scripting language to Hybrid.

The scripting language allows us to specify temporal co-ordination at two levels:

1. Co-ordination of the internal motions of animated objects (e.g., to specify the "walking" behaviour of an articulated animated object, a number of individual motions must be co-ordinated).

2. Co-ordination of the overall behaviour of a system composed of many active objects (e.g., the execution of two objects must coincide, or one must follow the other, or both must terminate simultaneously).

The second use of the scripting language can be applied to objects that are not animated. This could lead to a useful way of modeling temporal activities in a system. This aspect of the language can be used to specify the temporal co-ordination of several different media to be synthesized into a multimedia activity.

Furthermore, we wish to specify dynamic activities, that is, activities that respond to system dynamics. We have therefore proposed an event-driven extension to the language. This will provide us with the functionality to produce dynamic object visualisation, object monitoring, and interactive animation.

5 Object design tools

Object-oriented languages and software environments promise much in the way of software reusability and rapid development of applications and application "prototypes". Realizing these promises, however, is not simply a matter of providing "better" object-oriented languages. The "object design problem" can be understood in terms of four subtasks:

1. decomposition: how can an application be decomposed into a system of objects cooperating to solve a problem?

2. selection: which objects from the reusable software base can help to build the application?

3. configuration: how do we build the application from the pre-packaged objects?

4. evolution: how do objects evolve into objects that belong in the software base?

The object design problem, as stated, assumes that the task of building an object-oriented application can and should be more one of configuration than of "programming".

The decomposition problem is similar to the traditional problem of decomposing and specifying a large application in terms of a number of manageable "modules". The difference is that the responsibility of each of the pieces is more formally delineated by object boundaries. We expect that an object design tool to aid in the decomposition task would borrow techniques from CAD technology. Such a tool would be used for specifying software objects and the relationships between them, and it would guide the preparation of the information needed for the selection and configuration tasks.

The selection problem is that of deciding what existing software may be useful for meeting the design specifications. We take it as given that the software base may be extremely large, containing, say, tens of thousands of useful object classes, and that it will be constantly evolving. We assume that most objects will be highly reusable, and can be plugged into an application with

little or no modification. In a closed or slowly evolving software environment, it may be possible for experienced programmers to fulfill the selection task with the help of on-line documentation and a "browsing" facility. For large, evolving software bases, however, it will not be humanly possible to keep track of all the software available. It is even doubtful that today's browsers can help solve this problem. Instead, a kind of "intelligent browser" is needed to help rank reusable objects according to their relevance to the application. An intelligent browser should make use of the same kind of information humans use to solve the selection problem, namely: What is a piece of software "about"? What kinds of applications has it been used in? What software "belongs together"? An intelligent browser could be implemented using techniques as simple as keyword searches combined with synonym lists, or using more advanced expert system technology.

The configuration problem has traditionally been solved by programming. That is, one realizes the application by programming the specified objects. However, if an application can be realized almost entirely by pre-existing objects, one can configure a prototype by high-level programming, or *scripting*. A scripting language "glues" objects together to solve a problem. We need object-oriented equivalents to the concepts of data streams and "pipes" that can be used to configure software objects.

Finally, the evolution problem is basically a methodological one, but one that must be carefully considered in an object-oriented setting. Object design tools will never help programmers to build applications if the software bases do not contain well-designed, reusable objects. This suggests that the entire software production cycle should be integrated with the object-oriented system, and organized in such a way as to encourage the development of better software bases. This is a radical departure from the way software is presently developed. Instead of worrying only about meeting the deadlines for building an application, we should be equally concerned with improving our software environment. If we cannot build an application entirely from pre-fabricated pieces, then there must be something missing from our software base: either a new class of objects should be added, or some existing classes need to be refined so that they will solve our problem.

6 Evolving active objects

Our discussion of object-oriented software has thus far concentrated on well-defined pre-packaged object classes. In evolving systems, however, a more flexible approach to objects may be appropriate. We wish to consider, therefore, not only evolving applications and software bases, but objects that may themselves evolve. We identify at least three areas where this capability may be important: evolving configurations, evolving tasks and evolving knowledge.

Objects evolve through *dynamic inheritance*, that is, they may either inherit new *parts* or new *acquaintances*. In the first case, objects evolve through internal modifications; in the second case, they may evolve indirectly through environmental changes. Part inheritance is perhaps more interesting because we can consider the inheritance of not only instance variables and methods, but more generally the inheritance of anything that can be encapsulated as an object. Depending on the problem domain, we may wish to inherit facts, rules, histories, tasks, scripts, etc.

For example, during the development of a prototype of an application, one may employ a scripting language to configure pre-packaged objects. If we encapsulate these scripts, then the scripting objects may evolve together with the prototype. Since the configuration is not frozen into object classes, one may more freely experiment and re-configure the application.

As another example, consider the case of *messenger objects* that are used to deliver and handle messages in a distributed environment. In an active mail system, messenger objects may search for recipients, ask a series a questions and bring back the results to the sender [Hogg et al. '83]. The ability to travel in a network and to query name servers for information about possible routes are generally useful properties that should obviously be defined in a standard class. On the other hand, it is doubtful that all the types of work that a messenger may have to perform at each of its destinations can be determined in advance and fixed in a particular sub-class. Instead, we may allow messenger objects to dynamically inherit the tasks they must carry out. The same messenger may then adapt itself to changing knowledge about the network and the state of a given node.

We have previously implemented a system for evolving active objects using the Lisp flavors package [Tsichritzis et al '87, Casais '88]. Any given system of evolving objects, however, must adopt a particular model of dynamic inheritance depending on the problem domain being addressed. It is not clear which models are more generally useful than others, or even how these systems can best be employed to handle applications characterized by evolutionary change.

7 Concluding Remarks

We have proposed that an ADSS should be built using an object-oriented software environment, and that that environment evolve as the ADSS is developed. In particular, we are continuing our work to:

1. implement a compiler and run-time support for a concurrent, distributed object-oriented language descended from Hybrid
2. develop and refine a user interface support system for object visualisation, monitoring and interaction, based on a temporal scripting language for animated objects
3. develop object design tools to help in the configuration of applications from existing software bases
4. develop evolving active objects that can exploit dynamic inheritance to adapt to changing knowledge and requirements

Finally, we would like to comment on the irony of the situation. Some twenty years ago a remark was made (by R. Graham) that we were building computer systems like the Wright brothers. We put them together and we push them off a cliff. If they fly, great; if not, we start all over again. The remark was intended to show the need for more rigorous and thorough development. Prototyping and development environments such as we are proposing here gets us back to the Wright brothers. Are we going backwards? It is worth pointing out that the first successful man-powered airplane was built on the basis of a simple engineering principle. After every crash, it should be possible to pick up the pieces and try again with some modifications. They succeeded.

References

[1] P.S. Barth, "An object-oriented approach to graphical interfaces", ACM Transactions on Graphics, vol. 5, no. 2, pp. 142-172, April 1986.

[2] 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.

[3] E. Casais, "An Object-Oriented System Implementing KNOs", Proceedings COIS, ACM SIGOIS, Palo Alto, March 1988, (to appear).

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

[5] E. Fiume, D.C. Tsichritzis and L. Dami, "A Temporal Scripting Language for Object-Oriented Animation", Proceedings of Eurographics 1987 (North-Holland), Elsevier Science Publishers, Amsterdam, 1987.

[6] E. Fiume and D. Tsichritzis, "Multimedia objects", IEEE Office Knowledge Engineering Newsletter, vol. 1, no. 1, Feb. 1987.

[7] J.D. Foley and C.F. McMath, "Dynamic Process Visualization", IEEE Computer Graphics and Applications, vol. 6, no. 2, pp. 16-25, March 1986.

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

[9] R.D. Hill, "Supporting concurrency, communication, and synchronization in human-computer interaction–the Sassafras UIMS", ACM Transactions on Graphics, vol. 5, no. 3, pp. 179-210, July 1986.

[10] J. Hogg, M. Mazer, S. Gamvroulas and D.C. Tsichritzis, "Imail - An Intelligent Mail System", IEEE Database Engineering, vol. 6, no. 3, Sept 1983.

[11] D. Konstantas, M. Papathomas and O.M. Nierstrasz, "Integrating Object-Oriented Models: The Hybrid Approach to Evolving Systems", (submitted for publication).

[12] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM TOPLAS, vol. 5, no. 3, pp. 381-404, July 1983.

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

[14] O.M. Nierstrasz, "Active Objects in Hybrid", ACM SIGPLAN Notices Proceedings OOPSLA '87, vol. 22, no. 12, pp. 243-253, Dec 1987.

[15] O.M. Nierstrasz, "Encapsulation and Object-Oriented Concepts", in *Object-Oriented Concepts, Applications and Databases*, ed. W. Kim and F. Lochovsky, Addison-Wesley, (to appear).

[16] G. Pfaff, ed., *User Interface Management Systems*, Springer-Verlag, New York, 1985, (Proceedings of the IFIP/EG Workshop on User Interface Management Systems, Seeheim, FRG, October 1983).

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

[18] 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.

[19] P. Wegner, "Dimensions of Object-Based Language Design", ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 168-182, Dec 1987.