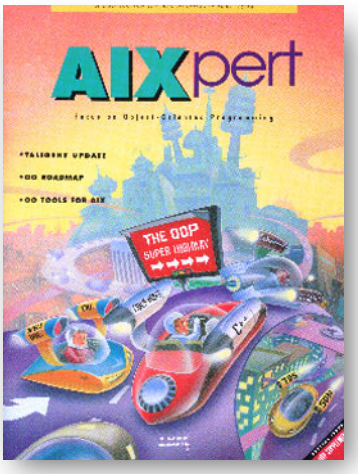
AIXpert DIGITAL VERSION IBM

TABLE OF CONTENTS



JUNE
1994

AIXpert BACK TO MAIN MENU

That's the Way It *Should* Work!

Recently, I was fortunate enough to see a demonstration of the new Taligent™ operating environment. Although it was only in beta form, it was quite impressive. Its intuitive drag-and-drop interface (called People, Places and Things™) allows you to create a "business card" to identify a person, and then use that business card icon to perform various actions related to that person. For example, to call that person, drop the business card on the telephone icon; to write a letter, drop the business card on the letter icon; and to sign the letter, drop your business card on the letter.

The demonstration consisted of a compound document that included text, a drawing, a full-motion video with audio, and a continuously updating ticker tape that could be instantly changed to display in any international character set. Also demonstrated were tools such as a highlighter, which could be used on any kind of data, whether text or graphics. The demonstration showed how to teleconference over a modem connection (simulated in the demo) using graphical displays that could be changed by either person, with the changes being immediately apparent to the other person. The resulting drawing could then be embedded in the compound document to become both—at the same time—an order for warehouse space (in the example) and a confirmation of that order. All these interactions were accomplished by program modules containing from 12 to 1,200 lines of code.

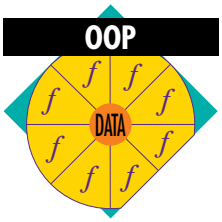
But the most amazing thing about the demo was that, in retrospect, it wasn't amazing at all. Chris Nelson, a panelist in the panel discussion in this issue, likes to say in his presentations on OpenDoc™ that, "...when the end user sees the results (of object technology), it will be perceived as a bug fix." And so it was with the demo. Every event was handled in a way that was natural and just the way that you'd expect to do it. You didn't have to worry about changing the format of data as you moved from one tool to the next; it just happened. Although there was a menu bar, you didn't need to access it for any of the actions shown in the demo—the most common tasks could be accomplished without it. Every time a particular function was shown, you could think, "That's the way it *should* work."

And so, I knew that we had to do an issue on Object-Oriented Programming (OOP). To ensure that our readers are kept abreast of this rapidly changing topic, we developed this special issue to introduce the topic. It includes a discussion with Cliff Reeves to provide the flavor of IBM's commitment to object technology. To address the current issues in object technology, we have included excerpts from a panel discussion of object technology that was held at the recent POWER Conference in Dallas, Texas. We also have articles on programming with SOM/6000, an update on Taligent's direction, a look at OpenDoc (reprinted from *BYTE* magazine), and a plan for preparing to develop object-oriented solutions as the new tools become available. We hope you find this issue helpful in preparing for the rapidly approaching world of object-oriented programming. Look to future issues of *AIXpert* for additional object-oriented articles as the technology unfolds.

George Noren

George Noren, IBM Corporation, Internal Zip 2830, 11400 Burnet Road, Austin, TX 78758. Internet: geo@austin.ibm.com. Since joining IBM in September 1979, Mr. Noren has written manuals for System/34, System/36, and AIX on both the RT® and RISC System/6000® platforms, and was a member of the InfoExplorer™ design team. He has also worked as system administrator for several AIX server machines and their clients, and is currently responsible for the Prototype Evaluation Labs in Austin. Mr. Noren studied engineering at Illinois Institute of Technology, holds a BA in English from the University of Minnesota, and an MBA from St. Edwards University in Austin.

George Noren



IBM Object Technology

A panel of senior IBM technical people discussed object technology at the POWER Conference in May 1994. Here are some of the questions asked by developers during that session.

Moderator:
Martha Harrington, manager of AIX Solution Provider Technical Support in Austin

Panel Members:
Phil Cannata, technical lead of the team developing the Taligent product on AIX
Chris Nelson, member of the AIX desktop architecture group, involved in OpenDoc
Ahmed Chibib, technical lead in the AIX Solution Provider Technical Support group
Malcolm Zung, member of the IBM Toronto PowerBench and C++ team
Jim Knutson, member of IBM's AIX Taligent development team
Roger Thornton, developer technical consultant for Taligent


Phil Cannata

Computing today is very data centric. Many customers have terabytes of data stored in databases. In the new object world, this data is encapsulated in objects floating around everywhere. How do they make the transition from the data-centric to the object-centric world? At a high level, how must they change the way they manage data?

Cannata: The object-oriented technologies do not address moving corporate data from where it currently resides. The Taligent Application Environment (TalAE) has data access frameworks that allow you to access the data. The problem is creating more data that must be made persistent. Each technology has its own way of doing that: using its own method or putting the data into the existing database.

In the future, TalAE will have properly behaved objects (semantically) that will help to reconcile the differences between databases that have similar properties. Then we can associate the data with particular types of objects and potentially make the data persistent with whatever mechanism is available.

Chibib: Today, we depend on relational databases to manage that data for us. We believe that those vendors' products will move to the object world to hide those mechanisms from users. That is, they will provide the object storage of that data as they themselves move into the object world.

Knutson: Object-oriented programming in itself is very data centric. You are actually dealing with the object, not the procedure, for modifying the data. So the translation from a data-centric environment to an object-oriented one is not as difficult as it might seem. It is not necessary to change the way you think; you are still dealing with data in its base form. Just encapsulate into objects the procedures that you normally use with data. Then it's a matter of getting the data from the databases into object form and vice versa.

Our business has a lot of information in relational databases and flat files. We use many vendor applications and some in-house programs. Although I hear a lot about object-oriented databases, it seems to me that they are sometime in the future. Would it be a reasonable strategy to do nothing and wait a year or so?

Chibib: The database vendors that we deal with (and we deal with all of them)—Oracle®, Sybase®, Ingres®, and so on—have definitely shown an interest in object programming. I can't speak for how soon they will have products, but there's definitely an interest in switching to the object-oriented world. It will not happen overnight. They have millions of lines of code to deal with, and that's not an exaggeration. But they probably will, over time, switch to that paradigm.

Thornton: As system providers, we want to ensure that you have a first-class programming environment (using our objects and frameworks)—the ability to use any one of those types of databases from any of those vendors.

We're probably not the ones to ask whether an Object Database Management System (ODBMS) is better than a relational DBMS because that depends on the application. Each product has its own strengths and weaknesses. Taligent's data access strategy provides a way to get to each of those without a massive difference in Application Programming Interfaces (APIs)—whether you're going after data in a legacy database, or you have data structures that make more sense placed in an object store, or you just want to put ad hoc data in a file. The data access framework design of Taligent 1.0 provides good availability for those data accesses.

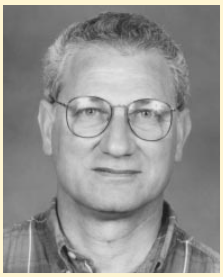
Nelson: In the database arena, one of the first areas that you will see is databases integrated into OpenDoc and Taligent. It is reasonable to consider modifying the front end of an existing database so that it integrates well within a Taligent or OpenDoc environment. Think of the front end of the database as the object. You've encapsulated that entire database in this front-end object that now has an object-oriented interface.

How do you embed database objects into a compound document, and what does that mean? What are the scenarios? How would users do that? What's the concept of linking database objects with other database objects? What's the presentation to the user? How do you take advantage of the integration of all these other facilities when working in a cooperative way in the abstract compound environment? The front end is where you will see it. And it doesn't matter whether it's an object database, a relational database, or a flat file. The front end will really take the forefront.

Thornton: Two publications I would recommend that focus on object technology and how it applies to databases are the *Journal of OO Programming* and *Object* magazine. They contain articles about real-world customers building information systems in C++, Smalltalk®, or some other Object-Oriented (OO) language that accesses legacy data in relational databases. Most of the leading database companies are working to provide tools that will make schema mapping from object to relational data structures easier.

Another organization to watch is the Object Management Group (OMG), a consortium to which our companies belong. It is defining methods so that all vendors will use the same kind of semantics for this type of data. There's a lot of information available on this topic.

Chibib: There is a learning curve as you move from a procedure-oriented to an object-oriented methodology. If you wait until everything crystallizes and then begin to learn and design, you will probably be too late. Start now by evaluating how to reengineer your application. And get your staff trained in object-oriented programming and methodology. Then you'll be ahead of everyone else.



Ahmed Chibib

Why would you choose C++ as an object-oriented language over Smalltalk?

Zung: I don't want to start any religious wars, but there are a few good reasons to choose C++. After completing a major programming effort, we chose C++ for two primary reasons. First, we wanted to use our strong background in C programming. Since C++ is a natural step from C, it was a good match. Second, at the time we started the project, we were a little tentative. We didn't want to take the full step into object-oriented programming. So we decided to go halfway and use the encapsulation features of C++, but not actually do object-oriented programming.

Once we began, we actually got into object-oriented programming very quickly. The point I'm trying to make is that C++ gives you the ability to do procedural programming if you choose (although we didn't in this case). Because Smalltalk is a pure object-oriented language, we would have had to make the full transition to object-oriented programming and design.

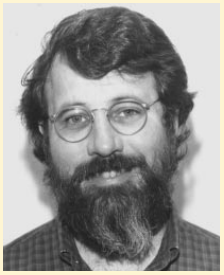
Nelson: By programming to an object model such as System Object Model (SOM), your result is really language-independent, at least within

Roger Thornton

several of the mainstream languages. You could have an object strategy in which the implementation language is C. Other objects could be implemented in C++ or Smalltalk. By placing the SOM interface on all of them, they effectively look like objects that work together.

What are the performance aspects of using SOM in an application and then moving the application to a distributed environment?

Nelson: Common Object Request Broker Architecture (CORBA) has a three-level hierarchy of distribution. SOM is a single-process model, so objects are really linked into your address space. The overhead for objects in your address space is very small—only a couple of machine instructions. Transparently, you can move into Distributed SOM (DSOM). In DSOM, objects are in separate processes and in a well-behaved NFS®. This is similar to mapping on another machine—a homogenous kind of network.


Chris Nelson

Obviously you pay a price for communications over the network, but that's generally a known quantity. With a more robust CORBA, the Object Request Broker (ORB) will communicate through a heterogeneous worldwide network using a heavy-weight Remote Procedure Call (RPC) mechanism. You pay a higher price for using a worldwide network, but the one you get depends on where the object is located. If the object is in the local address space, you do not pay for a heavyweight RPC mechanism. If the object is on a worldwide distributed network, then you must use the heavyweight RPC mechanism, which causes some additional performance considerations. You do not want a high-bandwidth transaction with an object that's located far away.


Malcolm Zung

Thornton: The difference in added overhead and scalability is small between existing RPC models and an OO RPC model. The overhead to execute a function on some other machine is high compared to executing the function in the local address space regardless of model. An OO model does not add much overhead. It is easier with OO to conceptualize what is happening when many of these modules are getting called and a lot of data is being moved. With an object model, I don't have to remember all the specific implementation details.

Nelson: The X server has provided us with a lot of experience in client/server relationships. I routinely start an application on a machine that is across the country and have it display on my workstation. The response time and usability is very good. There's no reason to expect the response time to differ just because you are using an ORB. It uses the same basic kind of transaction, which is an intuitive way of saying how good the response time will be.

Can you compare the OO RPC mechanism to the Distributed Computing Environment (DCE) RPC mechanism?

Nelson: We're getting wildly enthusiastic cooperation from other vendors in working out standards for network issues. OMG began with many proposals for an object broker description. The next step was to determine the kind of object services that should be provided. The result of the last round of proposals was one joint proposal that was agreed to by most members of OMG.

We're using standards organizations, such as OMG, to ratify these proposals and make them standard. It's going fast enough—at a reasonably good pace for the industry—so that we can get solutions to our customers in a timely fashion. We don't have to implement products before the standards are adopted.

We're ensuring that the proposal is adopted—that all the bits are set and everyone is happy. We've already announced agreements for various levels of technology sharing with other key UNIX players, including HP™ and Sun®, in order to get that interoperability. Several mechanisms exist—at the standards level and with cross-license agreements—for the companies to ensure that it works.

Everyone is working diligently to come out with an interoperable solution.

Thornton: The Taligent RPC mechanism provides a framework to build distributed applications. The implication is that if the client application has enough flexibility, we're comfortable. We're committed to the various standards, including DCE, CORBA, and others. A framework or object technology can provide a lot of flexibility with the underlying implementation without breaking your source code.

Is OMG using DCE to resolve object references? The issue is that you could partially resolve the object reference to the basic object adapter, which is the server, through the global directory. The object adapter

would know how to get to the proper cell directory. From there, you would get to the object server, which would know about the thousands of objects in that server space.

An application might have thousands of objects. You generally would not need to know all their addresses.

Nelson: OMG produces several services that must be performed on objects, including naming. The functions performed by the object naming service will dictate what the underlying System Object Model (SOM) will be.

There are other services involved with life cycles: creating and eliminating objects, keeping track of object references, notifying or passing messages to objects, and storing objects. Once OMG defines and standardizes these services, the subsystem vendors will implement them in the standard.

Does OpenDoc have security features, such as when an application accesses information from a database?

Nelson: That is currently missing in OpenDoc. It's part of the architecture that has not been filled. Look to OpenDoc Version 2 for that. Component Integration Labs will manage the OpenDoc technology and set the direction for the type of security services and authentication necessary. They are also looking for direction from OMG, which is trying to solve that same problem.

I want to start developing a new palmtop device, but I have no legacy code. There should not be an appreciable degree of overhead involved in using object technologies to implement the code for this device. Are these technologies viable for a scaled-down, low-end device where price is important, or should I continue with traditional development?

Nelson: Yes. They're very appropriate.

Zung: I'd like to comment from the C++ point of view. When we were making our product, performance was an important concern. Everyone told us that you can't get object-oriented code to run as fast as procedural code in C. But we wrote a new version of XLC using C++ that runs faster than the original version, which was written in a procedural language (not C).

Nelson: In a Personal Digital Assistant (PDA) environment, many services must be provided for applications: communications, networking, printing, graphical user interfaces, graphics interfaces, and so on. The technology within Taligent that supports those services is very appropriate for that environment.

Thornton: Another key technology that Taligent obtained from IBM is their microkernel implementation. That was very attractive to us, because to build such a device, you port the smallest kernel you can find, or write your own from scratch.

In designing a microkernel, you put only the essential functions into the kernel: process scheduling, memory management, and very low-level IPC mechanisms. Functions such as file system, networking, and I/O services traditionally found in the kernel remain in a protected process space, but run outside the kernel. You can scale other services for particular hardware applications without recoding the kernel. Traditionally, it has been difficult to change a portion of the kernel without making changes to the whole.

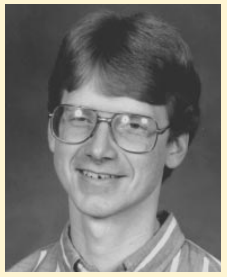
Most arguments in favor of OO are long-term cost savings for maintenance and future growth. How do you convince customers to take the first step and assume the risk involved when adopting a new technology?

Knutson: It's a tough sell. You cannot sell OO technology on near-term benefits unless you are using the application development environment from Taligent. It does most of the work for you. You only need to customize parts of it to make it fit your application.

That argument works well for a framework environment or for starting from scratch. But if you have a legacy application, you must approach it from a migration standpoint. Divide the existing application into parts so you can migrate the parts that will provide the least amount of trouble, and leave the rest of it alone. Beyond that, I don't know how you make a hard sell like that, because OO technology has a steep learning curve.

Cannata: Actually, there's a motivator in the compound document (as defined by OpenDoc). Within the next few years, users will demand that your application work in a compound document arena. The only way to work in a compound document arena is to make it object oriented. I've been talking to a company in Austin that

If you wait until everything crystallizes, you will probably be too late.



Jim Knutson

provides real estate management services. They have many applications and use standard tools including a spreadsheet, word processor, DBMS, and so on. They will not continue using those tools unless they have compound document functions.

Thornton: Many of these technologies are older than I am. It's only in the last 10 to 15 years that they have been talked about a lot—much of it has been theory. We are now beginning to produce technologies such as frameworks that provide functionality which is incrementally customizable without wholesale replacement. We could not have accomplished this without object technology. That drives the arguments that users will want it on their systems and vendors will identify ways to get it there. Hopefully, moving up that learning curve will not be as costly as it would be to not compete.

Learning how to do a good design will be key to your future.

Harrington: Try to convince the customer that there's some application that you can provide for them using OO that you couldn't otherwise provide. When you see some of the Taligent demonstrations, you get excited about new things that you can do.

Can you comment on the different types of analysis and design tools available? Do you have favorites, or have you discarded some in favor of others?

Knutson: The AIX® layers group has been using an enhanced version of the Booch notation. We had an OO design and analysis class earlier this year to bring some of our group up to speed. All the notations have strengths and weaknesses. You just have to fill in the gaps.

Thornton: It's worthwhile to invest the time to read about the principal authors in this field. It really ends up being a personal decision. They all have the same goal, but one might use a notation or style that connects better for you.

Cannata: Our team has been doing design reviews. We've developed some object diagrams to describe what should happen and what objects get instantiated to make that happen inside Taligent.

We developed scenarios for determining which objects will be created, the objects they're associated with, what methods are invoked, and so on. Everyone on the team is convinced that those object diagrams for a particular scenario are the most valuable concepts that we've produced. Not only does it help us understand the system, but we also use the same scenarios in testing; we can test all the objects in the same way. Everything else has fallen by the wayside except the object diagrams.

With these object diagrams, we could probably teach a class about Taligent internals, and maybe teach Taligent! The diagrams are a very good vehicle for teaching others what happens inside the system.

What is the Hollywood principle (don't call us, we'll call you) that I've heard mentioned regarding coding?

Thornton: When developing applications using a framework, the flow of control for that application has already been established and is controlled by the framework. When programming in this style, your code will be called by the framework. An analogy would be if a program consists of several black boxes that are pieced together, your code is inside one of those boxes. Periodically, the framework "opens the lid" and has that code execute. At first this seems strange compared to traditional approaches in which we must architect the application flow as we build it. But this is where we can achieve a great deal of programmer productivity as the programmer now only focuses on adding features to or changing the default behavior or flow.

If I'm a C++ coder and plan to write an application in Taligent, give me a scenario for what I should be doing now, and how long it will take to become really proficient.

Cannata: We have just started writing our first compound document: the calendar. Although you can use many underlying frameworks in Taligent now, I would recommend using the compound document framework. We don't have enough experience yet to say how long it will take to become proficient.

Thornton: Starting about a year and a half ago, we brought in several object-oriented programmers as consultants to write applications on the Taligent Application Environment (TalAE) and to learn about application development. In the past, we have even shut down development for one week while everyone in the company with a development system wrote applications.

From these experiences we determined what takes developers a long time to learn and what is easy to learn. Currently the API to Taligent is C++, although we hope to support other languages in the future. Therefore, the first and foremost suggestion is to learn C++ at the highest skill level possible.

Frameworks are the next thing to learn. We show demonstrations with many features that were written in a surprisingly short time period with very few people. But there is a learning curve to reach that level of productivity. Since you do not yet have the Taligent code, there are some products such as MacApp® and Object Windows Library (OWL) that you can look at or papers such as "Building Object-Oriented Frameworks" available from both IBM and Taligent. This paper also provides a bibliography to other sources of research materials.

Knutson: You can also partition your application into objects. Learn how to do a good design and OO analysis. That will be key to your future. Find out what it takes to partition an application into GUI parts, engine parts, and distributed applications. Learn the key concepts that you need to know about distributed applications and how they work. Taligent will be doing a lot of this, and the concepts and your mastery of them will be key to whether your application will perform well in a Taligent environment.

For legacy applications, is it better to start using OpenDoc since minimal changes are needed to start playing in the OpenDoc environment?

Nelson: Yes, that is a good point. We have applications, technology, and a body of thinking about migrating to the new environment. OpenDoc is intended to minimize what you need to learn and do to enter the compound document arena. That's why we believe that OpenDoc is a good first step for software vendors and end users moving toward Taligent.

You can upgrade your existing applications using your legacy of technology and programmer skills. You can also prepare to move to a richer framework with Taligent. It's not wasted because there is a large learning curve with OO technology, but you'll learn a lot with OpenDoc. Your applications can still interoperate within Taligent, so you will get a revenue stream from those applications.

You can take advantage of some Taligent frameworks that do not necessarily mean compound documents: graphics, printing, and internationalization frameworks, plus some of the other base frameworks. They are tools to help you develop applications by augmenting your existing software.

It's not going to be a piece of cake because there will be some design work about how to fit the applications into the frameworks. But you will go through that kind of thinking anyway. How do you take what you are using now and move it into a new environment? Along the way you will see benefits and be able to meld them together.

Roger, you mentioned that the concept of frameworks was a hot spot on the learning curve. I understand that the way Taligent uses the term "framework" differs from Apple's MacApp, Borland's OWL, and some other products. What technology is available now that we can review to help our people start climbing that curve?

Thornton: Both products—MacApp and OWL—would meet our definition of a framework. The primary difference is that TalAE is an entire operating environment consisting of numerous frameworks, while MacApp and OWL focus on one component of an environment—the user interface. A savvy programmer in either of those would have a very big head start on the learning curve.

If you're building an application from scratch, there's not much of a productivity story. But since you built it, you understand it. Every step was incremental. For frameworks, the theory is that someone wrote much of the application in the abstract sense. They put as much as they could into it and then handed it to you. You just modify it to do specifically what you want it to do.

You don't need to understand the entire body of code, but you do need to understand the key abstractions: the location of the interfaces to customize your application, and knowledge of the APIs. You may not need to customize some objects. And that's where the learning curve seems to be so much more than just starting up from scratch. MacApp and OWL are good example frameworks.

Zung: Working in C++ often involves using class libraries, many of which can be considered frameworks. You mentioned the difference between class libraries and frameworks. Class

OpenDoc is intended to minimize what you need to learn and do to enter the compound document arena.

libraries that actually allow you to define a new type and build on it are similar to Taligent's framework concept. The actual flow control is different, but the concept is similar. You start with a large class library and build the application-specific part of it by inheriting from the library.

Thornton: As a point of interest, many of the original MacApp designers are key architects to parts of the Taligent system, which explains why some similarity exists.

When will Taligent be available?

Knutson: General availability from IBM will be during the first quarter of 1995.

Thornton: Taligent has an early experience program whose purpose is to get the code in the hands of some developers we've been working with—to encourage them to build applications.

Those participating in the early experience program received code the first week of June 1994. This pilot program will allow us to tell our investors which class of applications was a piece of cake, which ones were difficult, the weak parts of the system that need work, and the ones that are strong. This feedback will provide valuable information for the investors (IBM, HP, and Apple) to make their early experience programs a success.

OOPS!

Here's Figure 4 that we inadvertently omitted from the article "Making Backups of Mirrored Filesystems on AIX 3.2" that appeared in the May issue. It is the shell script to reset the logical volume structure back to its original state.

```
#!/bin/ksh
# Restore system to original state.
# Root user authority required.
# Required parameter is the same lv name of previous script
umount /mnt
rmlv /dev/backuplv
#
# Find number of copies and add 1 to it. LVM supports only
# three copies (original + 2) of a logical volume. It should not
# be possible to get an error here if the first phase was done OK.
NUM=~getlvcb -c $1~
if [ NUM -eq 3 ]
then echo "Process error. Too many copies already exist. Aborting."
     exit 3
else
     NUM=~expr $NUM + 1~
fi
#
# Re-create the logical volume copy, using the allocation map file
# created by the first part of the process.
mklvcopy -m allocmap.file $1 $NUM
if [ $? -ne 0 ]
then
  echo "New mirror copy allocation failed. Process terminated."
  exit 2
fi
echo "New mirror copy allocated. Resync process begins."
syncvg -l $1
RC=$?
echo "Resync complete. Logical volume restored to original state."
exit $RC
```

Interview with Cliff Reeves

IBM DIRECTIONS

Cliff Reeves is IBM's director of object technology products. He is responsible for object-based products in IBM's Personal Software Products Division, including responsibility for the IBM-Taligent partnership. We talked to Mr. Reeves about the impact he sees from object technology being adopted by software companies and corporate developers.

What makes object technology so important at this time in our industry?

Reeves: The use of personal and desktop computers has only just begun. Even though we've seen a phenomenal growth in the past 10 years, there are many applications still waiting for computers. Increasingly, companies want to distinguish the way they offer products and services by how quickly they react to individual customer requirements and environments. For example, personal bankers need more and more information to say, "Yes, you can have that loan, and here's the interest rate and the credit limit." They must be able to do that without completing a lot of forms.

The person who handles customer complaints at an airline counter needs to recognize the frequent travelers. The last thing airlines want to do is infuriate those who travel the most. They would like to immediately say, "Yes, and by the way, we can put you in first class."

Everyone serving customers should be able to access information about the customer so they can tailor their service to meet the customer's preferences. That's the way many companies plan to distinguish themselves in the future. And this permeates every aspect of industry—from buying cars or airline tickets, to securing health insurance or visiting the doctor. Exceptional customer service is a competitive advantage. That means that everyone will have a computer on their desk for accessing customer information so they can make very quick and timely decisions.

How does object technology fit into that scenario?

Reeves: Since not everyone is computer literate, these new users will need systems that look the way they want them to look. They will need access to information, independent of concepts such as operating systems, databases, files, commands, and prompts. They will want meaningful information presented in context, using highly graphical, tactile interfaces. They will also want to work in a distributed environment, because not all data can be provided centrally or through a homogenous environment.

In addition, this future environment will change very quickly. So we need systems that are unbelievably flexible, decentralized, and good at managing change.

I have just described an incredibly complex development environment. It's very difficult to produce flexible applications. Most applications have their associated operating system and networking environments imprinted in their "DNA."

We need systems that are linked together dynamically. A request for information—satisfied one day by a centralized server in Albuquerque—must still be satisfied if that server is moved to New York. The application should not have to be redone completely to recognize that the server is moved.

Providing these kinds of systems is complicated. Assume that the Application Programming Interface (API) count is some measure of the complexity of writing an application (which is not entirely accurate). When we move to graphical interfaces to provide a friendlier and tactile, high-bandwidth communication between the user and

Cliff Reeves

the machine, the number of APIs increases by a factor of 10.

We have not even begun to discuss the issues of collaborative computing, distributive work, and the effect of the API count there. We need a set of technologies that allows us to deal with that. We need a new approach. Object technology is that new approach.

We will need to be very responsive in the future, and object technology seems to have an answer. What evidence shows that object technology will be adopted as *the* answer?

Reeves: Two or three things are happening that involve some type of standardization. First, we're beginning to see standards emerge for distributed object computing. We can see a mechanism now in the Distributed Computing Environment (DCE) and in the work within the Object Management Group (OMG). Within the OMG, we can define large separate components to handle the issues of security, location, high speed, and all that is needed to run a distributed operating environment.

Object technology allows you to build one application with reusable parts.

Secondly, we're seeing the concept of a reusable software component, which has happened very quickly. We see primitive signs of it in Visual Basic and Object Linking and Embedding (OLE), and more recently in OpenDoc, which specifically defines a reusable software component. It is not necessary for applications to be built from scratch or tied to a particular topology or platform. And more importantly, they don't have to be tied to a particular application, but can be accessible and reusable in combination.

Finally, we're seeing the emergence of a new kind of reuse—design reuse. Object technology allows you to build one application with reusable parts, and then build the same style of application from the same library of reusable components. Once you've done that two or three times, suddenly something else happens: you have a model or template showing how any particular type of application is constructed from reusable components. That produces a new method of software development called *frameworks*.

During the last four or five years, we've been hearing more about frameworks—mostly from the C++ programming world. Setting standards for frameworks takes the concept from niches into general use.

We're seeing the potential for a software value chain that really has three levels. At the lowest level, programmers (using languages like C++ and Smalltalk) will use standard frameworks for constructing core application components. At the next level, system integrators and software vendors will combine the standard components and add value by tailoring a particular program to solve a customer's problem. And lastly, users will receive the benefits from those and, in turn, manipulate the component, record the script, automate standard behaviors, and add more value.

Do you think customers and end users are ready for solutions that object-oriented technology can provide?

Reeves: I just completed a two-week road tour talking to IBM customers, consultants, and the press. IBM, Apple®, and WordPerfect® are collaborating on a software component specification that we plan to deploy across all appropriate operating systems. These companies see several benefits.

Apple has continued to innovate by offering technologies such as Quicktime®. The problem, according to Apple, is the length of time between delivering the technology as part of the operating system and its actual delivery within end user applications. It might take a couple of years to develop the technology. Then they have to evangelize developers to adopt it, and wait for a couple of years until products are available. Apple believes that a better way is to define a standard so that the software is written to a component mechanism, similar to Lego® blocks. Then every time we offer a new set, it would automatically go into applications—similar to a plug-and-play environment. We would like to use that as an operating system extension.

Another example comes from a large insurance company. This diversified financial services company just bought an investment banking firm that uses a lot of UNIX workstations and desktop PCs. The insurance company primarily uses MVS™ mainframe applications. An Information Systems (IS) person from the insurance company said, "When I need to provide an IS solution, I must build software from already existing parts, because I can no longer serve the wide variety of my own internal users by starting from scratch—I have to build from reusable software.

He added, "I had an interesting experience the other day. I walked into a department at the investment bank and said we're planning a new corporate time-and-expense accounting system. They almost threw me out. They had been running their time-and-expense reporting locally using a commonly available spreadsheet for almost two years. They print the reports and someone enters them into a transactional system for auditing, and so on. They are happy with what they have and it is familiar.

"They didn't want my new mainframe view of a time-and-expense accounting system. They wanted me to add a button to their spreadsheet to allow them to transfer the data to auditing and into our system." So from that experience, the IS person says that future applications must build from the equity that users have in their personal productivity products, yet they need to be easily extended. For that, we need a software component architecture.

How is IBM addressing the problem?

Reeves: Our answer is a set of services. I'll briefly describe a couple of them, just to give the flavor. It starts with a set of universal system services, including the following:

- Open Blueprint, providing low-level, fundamental linkage and access to common system services
- A set of integration services that totally redefine reusable software
- A set of very powerful and exciting tools that build on this model

We try to avoid building new, completely integrated monolithic applications. We want to help you build from parts—whether it's small parts like you find in class libraries, large parts like a thesaurus, or larger components, such as stock quote access.

We will deliver the functions of our operating systems in totally reusable components. That is not really new to the UNIX community, which has utilities that can be piped together to provide powerful functions. But this component strategy takes us to the next semantic level, in which the components will be more visual and have more rules about their interaction.

Could you explain what you mean by integration services?

Reeves: The integration services provide a specific definition of a reusable software component. Essentially, it's the Lego specification as well as the set of services for a pluggable software component. Integration services address four issues:

- How do you find a software component, make sure it is secure, and locate it on the network? That's based on IBM's System Object Model (SOM) technology that is, in turn, based on OMG's Common Object Request Broker Architecture (CORBA) specification. It's a very thorough, low-level programming interface to access a component.
- What is the visual part of the software component? How does it interact with other visual components? If I drop this component into an object, how do they negotiate space, and which one takes the events and maps them into a different set of actions that can be performed on the object?
- How do you link these components? To build reusable software, these components are often linked together very late in the development cycle. If we're going to have "blister pack" software that links together, then we need a more dynamic means of linking components together—from the local department, to the IS group, to end users who may just want to automate their mail box. Our mechanism for doing this is called the Open Scripting Architecture, a set of templates for linking components.
- How does the component store its data so that it can be interchanged? If applications, such as a bill of materials, are built dynamically from sets of parts, the data that defines the bill is likely to change. We need a technology that allows data to be self-defining so that as it changes, it can still be exchanged and understood. We call this technology Bento™—after the Japanese lunch box that keeps ingredients separated and in order.

That's certainly a lot of function. What can we expect to see in the near future?

Reeves: We are quickly moving toward implementation. This year we'll deliver the integration services piece to developers on OS/2®, AIX, Macintosh® System 7, and DOS/Windows™. We'll introduce it early in 1995 on our MVS systems and on OS/400®. It's probably the single, most

We will deliver the functions of our operating systems in totally reusable components.

fundamental change in the next two years that we'll see in the definition of reusable software components.

Workstation users have dramatically demonstrated that they want open systems with open software. What steps is IBM taking to ensure that other companies adopt these ideas, so that it is not an "IBM-only" solution?

Reeves: The environment I described is heterogeneous, multivendor, and multiplatform. We're deploying it on every important operating system. The technology will be licensed based on standards and partnerships. This technology (from WordPerfect, Apple, and IBM based on a set of industry standards) is being transferred into the ownership of a company called Component Integration Labs (CI Labs). CI Labs was founded on the model of the X consortium. The theory is that because of the major importance of many vendors agreeing on a component standard, the technology must be owned by a forum that is entirely accessible, open, and not controlled by a single vendor.

CI Labs was just recently incorporated, the bylaws have been signed, and anyone can join it. Just like the X consortium, there are membership levels giving increasing participation in the SIGs and in defining the standards. Although CI Labs will control the branding and certification process, anyone can obtain the source code for this technology and the specifications by anonymous `ftp`. The extreme level of membership is almost a non-membership, but with full access to the technology when it becomes available through anonymous `ftp`.

Although CI Labs will control the branding and certification process, anyone can obtain the source code for this technology and the specifications by anonymous ftp.

You can contact CI Labs now, by sending a note containing the word "help" to `majordomo@cil.org`. In fact, on the mail server you can browse all the directories that contain code, specifications, and development plans for the technology (See the sidebar on the opposite page).

What can we expect from other vendors?

Reeves: Endorsements have come from Lotus®, Novell®, Xsoft™, Borland™, and Taligent. Although each vendor has a slightly different deployment plan, you will see alpha code in the late spring and early summer of 1994, beta test in the late summer and early fall, and general availability from the different vendors in the late fall through early 1995.

You mentioned there would also be new tools.

Reeves: We are producing a new breed of compilers of "power tools" that assume the developer is building or using software components or frameworks. Most of these tools are highly visual.

We're getting an exciting set of tools from Taligent. These tools are for professional programmers who want to build the reusable software components and who are very focused on high-level function and productivity. These developers want to develop quickly and efficiently in their choice of programming languages, and to leverage their investment across several operating systems. Taligent will support the OpenDoc standard and extend it into a complete new environment for both developers and end users. An example is IBM's VisualAge™, a component-based tool that will move to the SOM and OpenDoc standards during 1994 and 1995.

That requires a powerful and not highly abstract technology. Taligent is producing all of its technology in C++ frameworks for professional programmers. They've taken the concept of what you would do with a class library of reusable software and used it to consider problems such as addressing text with very rich text functions.

How does IBM's Open Blueprint, announced in March (see *AIXpert*, May 1994), fit into the overall plan?

Reeves: The Open Blueprint is a set of low-level system services. When there is no consensus of what model to use for a particular low-level system service, Open BluePrint provides a recommended standards-based interface and hides all the complexities of the implementation. If you are working on a new emerging area for a system service, it recommends using a particular API since there is no dominant standard. Because it is based on an X/Open, OMG, or ANSI® standard, Open Blueprint is very explicit in almost every area except some emerging areas that do not have standards.

Since they are based on standards, SOM and Distributed Computing Environment (DCE) are the interfaces and system services that we recommend. We will support them in all of our systems. DCE is based on the standards delivered by

the Open Software Foundation® (OSF®) and is increasingly being adopted by system vendors. We are freely licensing IBM SOM to other system vendors. It will be part of the OpenDoc standard. It is based on, and fully compliant with, the OMG's CORBA specification for distributed objects.

At a lower level, Open Blueprint includes a technology called *AnyNet*. AnyNet is based on Berkeley Software Distribution (BSD) sockets. It's a *de facto* standard because of its simplicity and its adaptability to various underlying protocols. Open Blueprint supports all standard protocols underneath that level of interface, including TCP/IP, NetBIOS, IPX, System Network Architecture (SNA), and so on.

IBM is delivering a broad spectrum of object-oriented technology along with a migration strategy during the next year. What are the major benefits for developers and end users when they follow this strategy?

Reeves: Software and system vendors need cross-platform, multivendor integration so they can be confident that their software conforms to standards. They also want to leverage their development efforts by using software across platforms. When they write to the OpenDoc standard and use the Taligent frameworks, the result is productivity and portability across OS/2, AIX/6000®, HP/UX, and OS/400. It also allows them to focus on their added value.

System integrators and solution providers can depend on software that comes from multiple vendors and multiple platforms to have some real semantic and business value in terms of how it might be linked together.

Users can get new function quickly. When they need new function for an application, they will not be limited to the vendor that provided the original package. Other vendors' software components can extend that package very gracefully.

That summarizes what we're delivering this year. Everything I've mentioned will ship to developers in 1994. A couple new items on midrange and mainframe systems will come later.

Building Tomorrow's Solutions Today

Component Integration Laboratories (CI Labs) is the result of several leading companies banding together to openly develop and promote component software. This non-profit organization was founded by Apple Computer, IBM, Novell, SunSoft, Taligent, WordPerfect, and the XSoft Division of Xerox™. CI Labs will provide the technological specifications and foundation technology for developing and integrating component software through the OpenDoc architecture (see article on page 16).

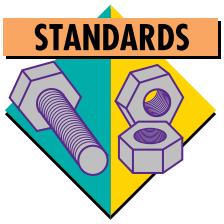
Supporting Technology Development

CI Labs will license and promote the technologies contributed by the sponsor companies, and distribute them as reference source code. Members will continue to compete on other technologies that differentiate their products, as they contribute to CI Labs their "non-differentiating" infrastructure technologies. CI Labs will manage the specifications for the technologies under its control, and will perform the following functions:

- Adopt and promote the key technologies essential for software components to integrate information and media from different applications within a networked environment and on multiple platforms
- License reference source code to developers and system vendors
- Provide open access to decision making and priority assessment in a vendor-neutral forum
- Manage and facilitate software contributions, design discussions, technology definition and evolution
- Support developers in adopting these new technologies by providing test suites, documentation, and training
- Validate the interoperability of components and platforms through a comprehensive testing program
- Assist in marketing efforts
- Collaborate with standards organizations (such as the OMG)

CI Labs Internet Lists

CI Labs provides related services and forums regarding CI Labs technologies and component software in general on the Internet. To request information, address a message to `majordomo@cil.org`. In the first line of the body (not in the subject line) enter `help` to receive E-mail describing majordomo or `lists` for a list of CI Labs mailing lists. For more information, send mail to `cil@cil.org`.



A Close-Up of OpenDoc

By Kurt Piersol

OpenDoc is a new vendor-neutral standard for compound documents that offers cross-platform support.

In recent years, there's been an industry shift toward document-based computing and away from application-based computing. Starting in the 1970s at Xerox, and continuing with architectures like OLE from Microsoft®, this movement toward more natural ways of assembling documents is now gathering significant industry momentum.

OpenDoc is a new entrant into this field—one intended to be a vendor-neutral, open standard for compound documents. *Compound documents*, the key to this type of computing, are documents composed of many different kinds of content, all of which share a single file. These documents can contain almost any type of data, such as tables, charts, and text, as well as video, sound, note cards, or 3-D graphics.

A Quick Tour of OpenDoc

Typically you can edit any or all of these types of content in place in a compound document. This means that several editors can work on a document at the same time, unlike in today's computing environment where one editor owns the entire document.

Given a set of editors working together on the same document, there must be boundaries to sort out where one kind of content ends and another kind begins. Otherwise, it would be impossible to discern which editor should work on particular sections of the document. In OpenDoc parlance, these bounded sets of content are called *parts*.

Of course, the whole point of a compound document is to be able to mix the types of content, so a mechanism is needed that can put one part inside another without the parts losing either their identity or their boundedness. This process is called *embedding* because it's rather like the typical real-world embedding process. You can embed raisins into bread, for instance, but doing so doesn't change the fact that the bread is still bread and the raisins are still raisins. Still, the resulting raisin bread is tastier than either ingredient alone.

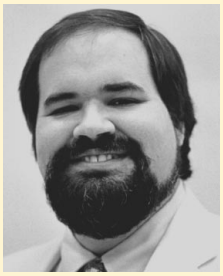
Thus, a compound document has various types of content, as well as parts embedded inside other parts. Each type of part has its own editor. All these parts share the same windows, as well as the same file on the disk or storage server. When you open an OpenDoc document, you're really looking at a collection of parts. When you edit the document, you're using a collection of editors.

These editors must work together smoothly. Editors have only a few basic tasks: storing the contents to disk (if needed), drawing the contents out to the screen or a printer, and letting the user act on the contents by accepting events such as mouse-clicks and keystrokes.

A compound document system needs to sort out the boundaries between the editors so that efficient editing can occur. OpenDoc does this by using a series of libraries that sort out the boundaries for the part editors. While this is an object-oriented system, it's important to note that OpenDoc isn't an object-oriented framework, because it wasn't designed to be extended through inheritance. Instead, it's an object-oriented interface between part editors that can be written in different programming languages and by different organizations. These libraries can be used via procedural code if required.

OpenDoc Components

OpenDoc has several important components that are used to organize a document's content and implement the sorting mechanism. The first



Kurt Piersol

of these is OpenDoc's layout system. OpenDoc helps the parts negotiate about layout so that they can avoid blasting bits over one another. The same layout system helps determine what editors are invoked and when they get mouse events. The layout system works for on-screen windows, off-screen bit maps, and printer contexts. It handles 2-D and 3-D graphics, overlapping parts, and multiple active parts that are updating asynchronously.

OpenDoc has an event-dispatching system that routes events to the correct part. It uses the layout system to let you activate parts directly, without allowing double-clicking or menus to get in the way.

OpenDoc's storage system helps parts store complex information in a shared file. It even helps with storing multiple document drafts and embedding information. This includes a data transfer system that helps the parts ship information—including embedded information—through the Clipboard, by linking, or by use of the drag-and-drop-mechanism.

Lastly, OpenDoc has a scripting system that lets users coordinate the actions of various part editors, either within a document or across a network and across platforms.

Rather than describing all these capabilities, I will instead focus on several of these areas in more detail. Some of the most interesting parts of OpenDoc revolve around how it dispatches events, handles storage, and does scripting.

OpenDoc Event Handling

OpenDoc is a cross-platform architecture, and event handling varies considerably in different GUI environments. For example, in the X Window System®, keystrokes go to the window that the cursor is over at that moment. On a Macintosh they go to the frontmost window at the insertion point. Other systems have yet other ways to determine where keystrokes should go. Similar interface discrepancies arise when you consider how to handle menus, windows, clipboards, and dialog boxes.

We members of the OpenDoc team made an early decision during the design process: Don't demand alterations of the human-interface environment. A good decision, but it left us with a problem. How could we make a mechanism that worked generically if the interface changed among platforms?

There were two options. One was to treat human-interface elements as a collection of special cases on each platform. The other, which we chose to implement, was to come up with an abstraction of the fundamentals of our problem and then fit it to each platform. Our solution was to build two major structures, the dispatcher and the arbitrator. The *dispatcher* is an object that helps the underlying platform's dispatcher to find the correct part editor. The *arbitrator* is a way for parts to tell the dispatcher which editor owns the stream of keystrokes, the menu bar, or any other shared resource. Together, they let OpenDoc work with the different human-interface models of different platforms.

The arbitrator is actually a table that shows the resource that can be owned and what part editor owns them. Each of these resources is called a *focus* of arbitration. To get resources, a part editor asks the arbitrator for a set of foci by name. The arbitrator then uses a two-phase commit mechanism to ask the present owners to give up ownership. The arbitrator asks the various owners to give up the resource in the first phase and then reassigns ownership in the second phase.

Networking and multiprocessing experts will recognize this technique as a standard way of preventing deadlocks involving resources. For example, imagine that one part editor owns the menu bar, and another has ownership of the keystroke stream. If each editor wanted the other's resource but refused to give up the one it owned, there would be a deadlock. By asking for resources as a set and assigning ownership in two phases, OpenDoc prevents this sort of "deadly embrace" between part editors.

OpenDoc's arbitrator is extensible, by both platform implementers and application developers. At any point, a new arbitrator focus can be added by creating an object called a *focus module* and adding it to the arbitrator. This extensibility means that new hardware resources can be managed through the arbitrator, as well as new software resources, such as server connections.

Most systems deliver events to windows, but because compound documents have parts that divide up those windows, a second stage is needed to get the events from the windows to the parts inside them. The OpenDoc dispatcher does this by taking human-interface events from the operating system and handing them to the correct part editor. This dispatcher is similarly open-ended.

OpenDOC has several important components to organize a document's content and implement the sorting mechanism.

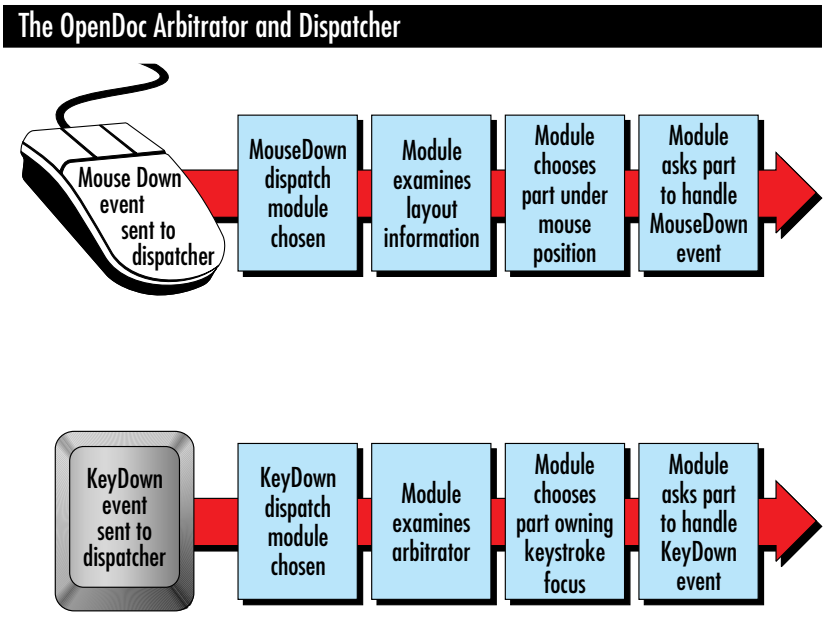


Figure 1. The OpenDoc arbitrator and dispatcher

Figure 1 shows how the OpenDoc arbitrator and dispatcher work. A stream of mouse and keyboard events are directed to the dispatcher, which directs them to the appropriate editor handling that portion of the document. Where necessary, as in the case of the keyboard event shown in Figure 1, an arbitrator module may be required to decide which editor receives the keystroke.

Although some systems have nested windows that could theoretically be used for embedding, they usually don't have enough information about layout to make a very effective embedding tool. So OpenDoc provides a more sophisticated mechanism called a *frame*. Frames are part of OpenDoc's layout system, and they help the parts divide up the drawing area of a window into separate regions for each part. The use of frames allows OpenDoc to support overlapping parts that are simultaneously active, a requirement for many multimedia documents. The OpenDoc dispatcher reads the arbitrator and frame information to decide which part editor should get an event (see Figure 1).

Some systems, such as Windows, have a tremendous array of events that can be passed to applications. Others, like the Macintosh, have a much smaller set of such events. The OpenDoc dispatcher must be able to handle both extremes. It can have a new behavior added at runtime to handle new kinds of events. In addition, any part editor can monitor the dispatcher and watch the stream of events that passes through the dispatcher. Both monitoring and extension are accomplished through objects called *dispatch modules*, which are added at runtime to the dispatcher.

Storage in OpenDoc

Probably the single most interesting part of OpenDoc is its approach to storage. There are many different data formats on many different systems. Superficially, it appears that programmers have a perverse desire to define new storage formats that are different and incompatible. Even the same applications often have different formats on different machines.

But there are some really good reasons for the differences, however annoying they may be. There are some trade-offs that make choosing a file format difficult. The two most common trade-offs come when you must choose between standardization and innovation, and between publication and efficient editing.

The first trade-off problem arises when you want to add new features, but you have an existing file format of your own or there is a market need to support a particular format. Unfortunately, file formats are seldom extensible in a convenient way.

Then there's the second problem, which tends to arise when a standard interchange format has been proposed. The design of such formats is almost always centered around making them easy to read and extend. Unfortunately, this often means that they aren't designed for efficient editing or high performance on a particular machine. The compromise is often to support both formats and allow one to be rewritten as the other.

The worst problem from a compound-document point of view is that these formats don't mix very well. Efficient editing formats are often based on replaceable pieces (that is, strings of bytes) that can be randomly accessed through some sort of table of contents. This makes them very easy to edit efficiently, but they are very specific to a particular editor.

Efficient publication formats are often stream oriented, making them easy to understand for a program, without regard to its internals. Problems arise when you try to mix piece-based formats and sequential formats. If you insert piece-based information into sequential information, there are often unacceptable constraints about when and how new pieces can be added. When you put sequential information into piece-based formats, there are often problems with the size of the sequential information, and there's no way to

extend the sequential information without understanding the piece-table structure of the container.

OpenDoc addresses these problems by creating an open meta-format, a way to store both sequential and piece-based information in a file without conflict. Although OpenDoc can support many different storage systems, it specifies one implementation on each platform based on Bento, a compound-document storage format developed at Apple that was designed to address many of these problems, support multimedia, and work on many platforms (see Figure 2). OpenDoc storage makes no assumptions about whether formats are sequential or piece-based. It allows random or sequential access.

As shown in Figure 2, document parts can be stored anywhere within the file. These parts can then be reconstructed into a single stream of data by referencing the file's table of contents.

To explain OpenDoc's storage model, I'll work up from the simplest elements of storage to the full document structure. At the base of OpenDoc documents are stream-like entities called *values*. Every one of these values looks to an editor like a complete file. Each value has read, write, and seek operations, just like any typical file.

In addition, OpenDoc supports two extra operations, insert and delete, that insert and delete data into the middle of a stream without copying massive amounts of information. It does this by maintaining a table of contents that assembles random chunks of the file into the appearance of a stream. Inserts and deletes generally alter the table of contents instead of moving the contents around. Sequential formats can use stream operations, and piece-based formats can use insert and delete operations to alter data in place.

Values are collected inside objects called *storage units*. Every OpenDoc part has its own storage unit, which consists of a list of named properties, each of which has a list of typed values. Thus, a storage unit is a lot like a directory in a typical file system. Properties are a lot like named files, and values form the contents of each file.

Unlike a directory in a typical file system, however, a storage unit has the added advantage of being able to store multiple formats for every property. A file system has only one set of contents per name, whereas a storage unit can have several, each with its own type. This makes OpenDoc uniquely suited to storing multiple

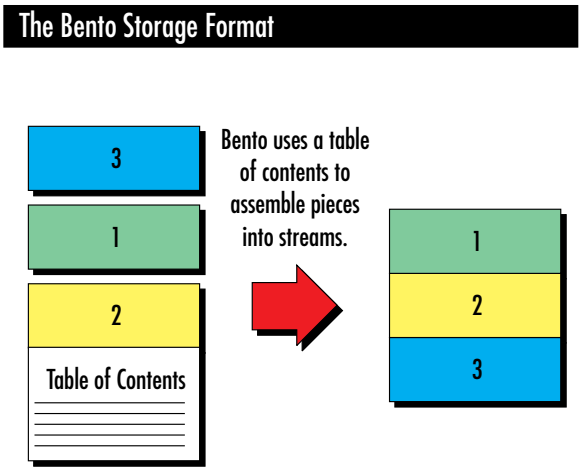


Figure 2. The Bento storage format

representations of a part. An OpenDoc part editor could, for instance, store both a standardized format and an efficiently editable format with a single name.

Storage units are collected into structures called *drafts*. A draft is really a list of storage units—a snapshot of the state of a document. When you make a draft in OpenDoc, you're really saving the state of the document for later retrieval, even if changes are made. OpenDoc is very efficient about how it stores drafts, using the information from read/write/insert/delete operations to store only the changes from the last draft.

One exciting thing about drafts is that values can use them to refer to storage units in a very robust and flexible fashion. Any value can include a reference to another storage unit in a draft. This allows pieces of data in various formats to have a standard way of referring to one another. As a result, a draft can support many different organizational structures, from simple hierarchies such as a file system to full hypertext webs.

Every draft has a pointer to the topmost part, or *root part*, of a document, which is used to open the draft into windows or to print the document. The rest of the document embedding is done using OpenDoc's reference mechanism. Thus, documents can have many different structures. The root part is in charge of it all, and it sets up the document structure based on its own content rules. A spreadsheet, for example, might allow embedding only on the main sheet, while a Rolodex editor might allow it on any card. The same part can be embedded in many locations.

The OpenDoc Storage Model

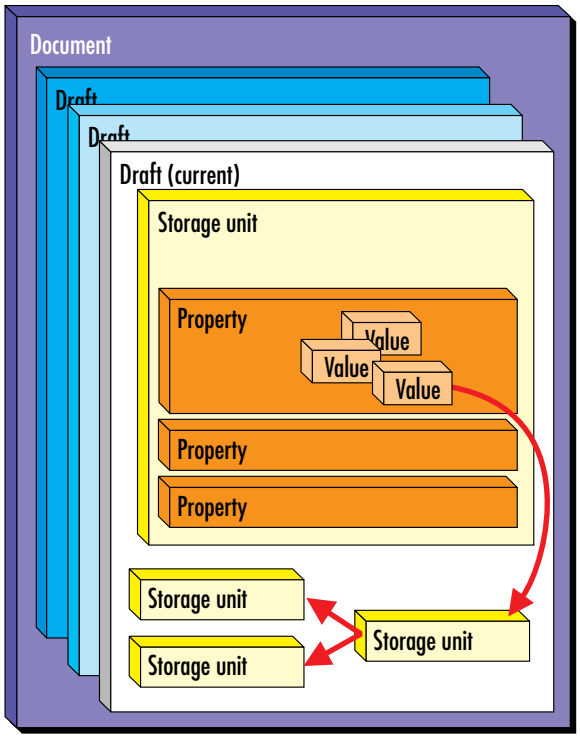


Figure 3. The OpenDoc storage model

Finally, there are documents that are just lists of drafts. One special draft, the *current draft*, represents the latest state of the document (see Figure 3). Users see the current draft upon opening a document. This structure is one of the key design points in OpenDoc. It supports more kinds of documents than any previous compound-document system. Even better, for part editors that don't need all its power, they can simply do stream I/O operations and ignore the rest.

Referring to Figure 3, values contain the basic atoms of data and look like files to OpenDoc editors. Values are collected inside storage units, which contain a list of typed values. A storage unit functions as a directory in a file system and can refer to multiple copies of the data in different formats. Storage units are collected into drafts. Each draft is a snapshot of the information. A document is composed of different drafts, the latest snapshot of which is contained in the current draft.

OpenDoc Scripting

OpenDoc scripting is designed with two major points in mind. First, scripting is a human interface, intended for users—not C programmers. Second, scripting is most valuable in the context of work-flow applications, where several kinds of data are used in some sequence to accomplish a task.

It's tempting for a designer to imagine that scripting, like automating user tasks, is just another programmatic interface: Just expose the innards of your editor and let users program it any way they like.

This doesn't really work, though. Imagine that you're creating a scripting system for a word processor. Internally, run-length encoding is an efficient way to store character-formatting information. However, at the scripting interface, you'd hardly want your users to understand the arena of run-length arrays so they can set the font of a word. Instead, you'd create an abstraction of the data that lets them select words, lines, or paragraphs, and let them set the font of any of those things.

The second major point of OpenDoc, workflow, is just as important as the first point. A user usually has a task to complete that involves many distinct actions on many distinct kinds of data. To have a useful automation system for OpenDoc, you need to be able to script the actions of many parts in a single script. These parts could be in many documents, on many machines, across networks on many different kinds of hardware and operating systems. Even worse, some of the instructions need to be delivered through store-and-forward systems, such as E-mail. Clearly, this is a tough—but important—problem to solve.

The OpenDoc team chose to extend an existing solution to this problem. Over the past few years, Apple has created a scripting architecture that it calls OSA (Open Scripting Architecture), which solves much of this problem for the Macintosh.

OSA is a series of libraries that each address parts of the problem. There's a standard calling convention, called Apple Events, that allows applications to call one another over a network or on a single machine. OSA scripting systems can coordinate activities among many machines and many applications from a single script. Another library, the Apple Events Manager, simplifies the process of making and receiving Apple Event calls.

Another OSA library allows different scripting languages to call one another and can even be used to plug other scripting architectures into OSA. A good example of such an architecture is the OLE 2.0 automation interface. OSA provides a standard way to adapt other scripting architectures so that they can be called from OSA-compatible languages.

There's also a standard record of calls, called the Registry, that defines a common set of operations that most editors can support. Operations such as copy, paste, create, delete, and move are all defined in a standard way in the Registry.

Central to the design of the Registry commands is a standard way of naming selections in individual applications. This is one of the keys to verbal intuition and consistency in OSA. OSA's naming scheme, called *object specifiers*, lets users name individual objects or groups in a standard way. It lets users treat their documents as if they were a giant object database that they can query in a natural way. For instance, it's quite easy in OSA to set the color of "every cell whose value is less than zero" in a spreadsheet to simply "red" or set the color of "cell R1C2" to "green."

OSA also includes a standard way for applications to publish the kinds of objects they make available for scripting, as well as what can be done to those objects. This user terminology is described in a resource and allows OSA scripting languages to support many different language syntax forms. Thus, they can be internationalized to resemble natural spoken languages as well as existing computer languages. Since OSA was intended to be a cross-platform architecture, it fits in nicely with OpenDoc's goals.

However, the OpenDoc team needed to extend this naming architecture to handle embedding. OpenDoc includes a service, called the *name resolver*, that allows part editors to easily handle object specifiers. This service parses the object specifier and then calls back into the part editor to resolve object specifiers into pointers to the part editor's data structure. If the specification crosses a part boundary, the name resolver can switch contexts and begin resolving in the new part. It handles query-type specifiers and even does query optimization in some cases.

The OpenDoc team also needed to extend the OpenDoc dispatcher to deliver the Apple Event (which is called a *semantic event* in OpenDoc) to the correct part. This involved using the name resolver to determine which part should handle the semantic event and then dispatching the event to that part.

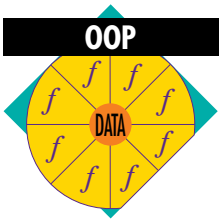
The result is quite striking. Users can refer to objects using the visible embedding structure of the document, saying things like "delete seconds 1 through 30 of the first movie of paragraph 1 of my document." These instructions can be delivered either in real time or through store-and-forward systems, thanks to OpenDoc's sophisticated naming scheme.

Getting Your Hands on OpenDoc

OpenDoc is being implemented right now. By the time you read this, it will probably be in its alpha-testing phase. Once that is finished, we plan to release the source code generally. A vendor-neutral organization, CI Labs (Component Integration Labs), will be the owner of OpenDoc (including a cross-platform version of OSA) and will be the point of contact for obtaining the source code. You can contact CI Labs at 688 Fourth Avenue, San Francisco, CA 94118, (408) 974-6549, or on the Internet at `cil@cil.org`.

There are white papers, specifications, and other documentation available from CI Labs right now, with more on the way. In the meantime, look for the various CI Labs companies to begin releasing seedings of the technology on their respective platforms.

Kurt Piersol is a software architect at Apple Computer (Cupertino, California), where he leads design teams and coordinates design elements on a number of projects. Currently working on OpenDoc, he previously led the Apple Events project, coordinated the design of a Mac® Interapplication Communication system, and was the early technical leader for AppleScript™. You can contact him on BIM c/o "editors."



Taligent Update

By Richard Hoffman

This article updates information in the August and November 1993 issues of AIXpert.

In March 1992, Apple and IBM created Taligent, a company dedicated to developing a comprehensive object-oriented operating system. Since then, much has changed. Priorities have been revised, a new partner invested, and IBM began work on its first offerings based on the new technology.

Operating Systems, Layers, and Components

Taligent's original goal was to deliver a new operating system in the mid-1990s. This system was to be object-oriented at all levels, "right down to the interrupts." The architects originally believed that the dramatic improvements they were seeking—reducing development cycles by a factor of 10 or more and creating a plug-and-play interface at all levels—could be obtained only in a system designed from the start with objects in mind.

But, at IBM, programmers were convinced that it would be possible to create a "mapping layer" between existing operating systems and new components (as shown in Figure 1). This layer would allow Taligent technology to appear on AIX, OS/2, and other platforms. Developers could take advantage of most of Taligent's frameworks without committing to a new operating system. The layered approach also enables users to run Taligent applications along with applications they run on today's operating systems.

Thus, Taligent acquired a new goal: to produce an application environment that would offer the same benefits as the originally envisioned system but which would run on existing systems.

Since existing operating systems have their own procedures, the layered approach does not meet Taligent's goal of a system that is extensible at all levels. Therefore, Taligent intends to provide a native operating system based on its technology and the IBM microkernel, shown in Figure 2. For now, the priority is to develop layers that allow Taligent investors to quickly integrate the technology into their own product lines. Taligent may also decide to release some components as separate products.

Taligent has already delivered the initial toolkit of its application frameworks to investors and to a small group of independent software vendors and corporate developers. The Taligent product suite is still on track for commercial release in the mid-1990s.

Meanwhile, IBM has already released early code for its version of the Taligent frameworks on OS/2 to selected customers and vendors. Full-scale beta programs for Taligent frameworks are planned later this year on both OS/2 and AIX.

A New Investor

In January 1994, Hewlett-Packard® (HP) agreed to purchase a 15% stake in Taligent, with Apple and IBM each owning 42.5%.

HP will license and use Taligent's object-oriented products and technologies on its HP/UX operating system. Taligent will license key open systems technologies from HP. HP also received a seat on Taligent's board of directors and gained the same early access to Taligent's products and technology already enjoyed by IBM and Apple.

"We're making this investment because integrating Taligent's comprehensive object-oriented frameworks with HP's products and technology will strengthen our leadership in distributed-object computing," said Gary Eichhorn, general manager of HP's Workstation Systems Group.

From the beginning, Taligent's charter was to enlist additional investors and licensees of its products and technology. "IBM and Apple formed Taligent in 1992 with the goal of making it the *de*


Richard Hoffman

facto industry standard in object-oriented technology," said James A. Cannavino, IBM senior vice president, Strategy and Development. "This announcement adds support from another very important industry computing platform and clearly positions Taligent as a leader in object technology."

Commitment to Standards

Along with the HP announcement, Taligent announced plans to use industry-standard processes to make its programming interface broadly available and to develop products that comply with key industry standards for interoperability and distributed computing.

To ensure the broad acceptance of the Taligent application frameworks, Taligent will submit its interface specification to X/Open for adoption through X/Open's Fast Track process. After the Application Programming Interfaces (APIs) are adopted, X/Open will establish a testing-based certification program to make the APIs available to third parties.

Once adopted, X/Open will publish the approved Taligent APIs, distribute conformance tests, and provide certification and branding of conforming implementations. Through its Open Systems Requirements Process, X/Open will provide a forum for customer requirements and manage the consistency and compatibility of extensions to the interface specification.

Taligent also announced its support of the Object Management Group's Common Object Request Broker Architecture (CORBA) distributed computing standard. Taligent is licensing HP's CORBA implementation, called HP Distributed Object Management Facility (DOMF), and its underlying Distributed Computing Environment (DCE) technology. DOMF and DCE/9000 are HP's technologies designed to allow users to share information and resources across a network. Incorporating these technologies into Taligent's products will allow developers to write distributed computing applications and take advantage of CORBA services on any compliant system.

Taligent intends to take advantage of the agreement between HP and IBM to integrate the HP DOMF with IBM's System Object Model (SOM) and its extensions for distribution. These technologies will allow a common messaging mechanism for sending and receiving objects. Taligent intends to support the merged HP/IBM technology and the CORBA 2.0 initiative to ensure that Taligent-based applications can interoperate easily across multiplatform networks.

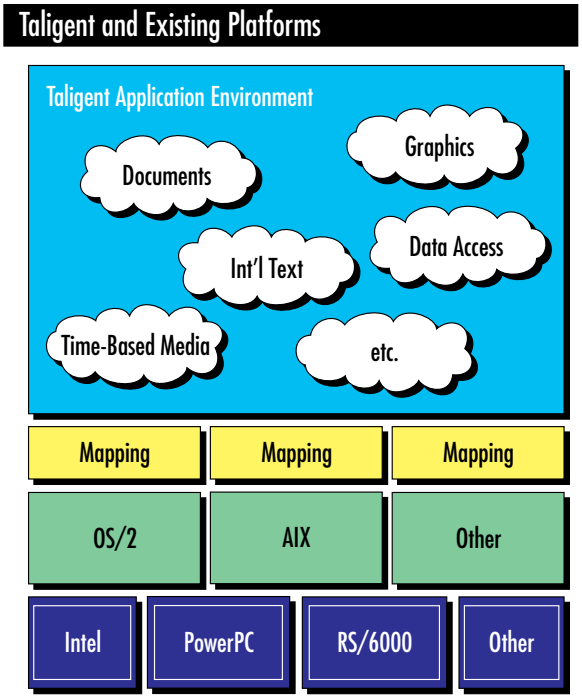


Figure 1. Taligent and existing platforms

The commitment to industry standards can also be seen in Taligent's determination to use only ANSI-standard C++, and its pervasive use of Unicode as a basis for text.

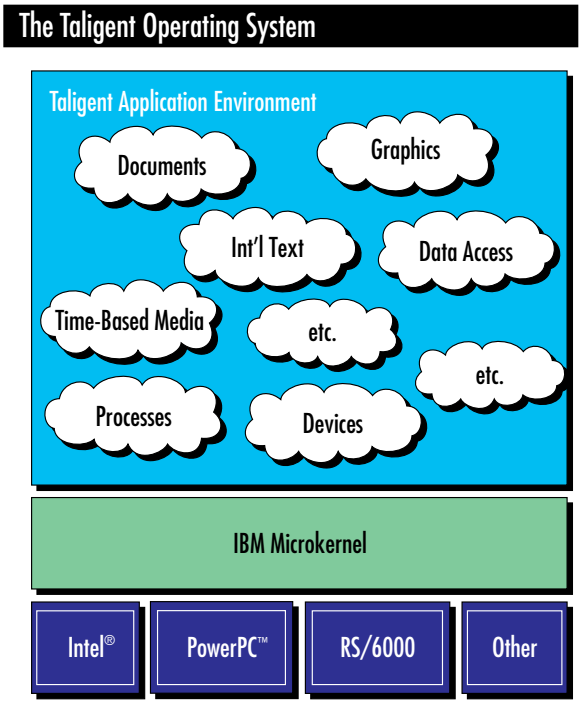


Figure 2. The Taligent operating system

Taligent Demonstration

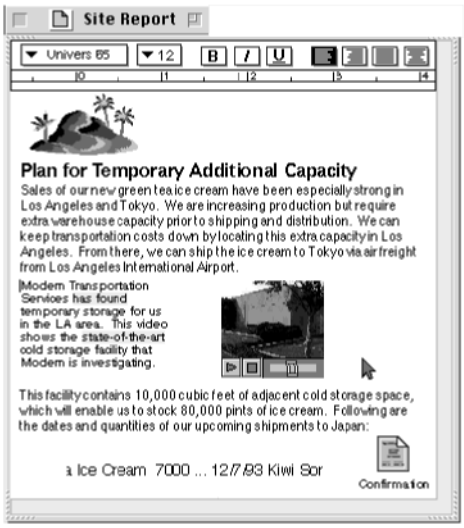


Figure 3. Ice cream memo

Taligent Demonstration

Figure 4. Tool palettes

An Early Peek

Taligent has displayed parts of its system in a series of recent demonstrations. The system continues to evolve, but this description of the PC Expo version should provide a good idea of what the demonstration conveys.

The demonstrations show features of the Taligent system, such as its task-centered interface, its pervasive use of international text, state-of-the-art graphics, and multimedia. The demonstrations also show how seamlessly Taligent applications work with each other, even when they have not been designed to do so.

The demonstration consists of 32 separate applications, written by four Taligent engineers during a four-week period. The scenario of an ice cream company renting storage space gives Taligent the opportunity to explore areas such as inter- and intra-office communications, telephony, collaboration, and multimedia.

The 32 applications do not form a suite, but were designed and written independently without specific knowledge of the other applications. The smallest required a dozen lines of code; the largest (a program with complex graphics and networking requirements) required 1,200 lines.

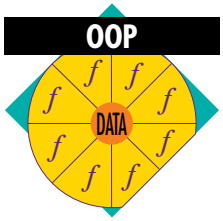
In the demonstration, the sales representative receives electronic mail from the ice cream vendor. The mail contains a "business card"—a Taligent object that represents data about an individual. The sales rep drops the business card on a sheet of stationery to perform a database lookup. He then drops the card on a phone icon to call the vendor. The rep and the vendor can collaboratively use a space management application to schedule space in the rep's warehouse.

Figure 3 shows a memo written by the vendor to describe the final arrangement to his manager. It includes several live objects: a video describing the site; a ticker tape giving constant updates on ice cream sales; and a copy of the sales confirmation obtained from the space scheduling program. One of the more impressive moments in the demo occurs when we see the video and the ticker tape in independent motion. Such seamless integration between applications is rare, even when the applications have been designed to work together.

Another interesting feature of the Taligent interface is the use of generic tools. Highlighting in Figure 3 was done using the highlighting tool shown in Figure 4. But the highlighting tool was not written specifically for the document application; it works with any text in any document. Taligent's vice president of technology, Mike Potel, compares this to using pens (tools) and paper (documents) in the real world. "You wouldn't buy a pen that would work only with one kind of paper. You expect any of your pens to work with any piece of paper," says Potel.

Richard Hoffman, IBM Corporation, 11400 Burnet Road, Mail Stop 9370, Austin, TX 78758. Mr. Hoffman is IBM's technical liaison to Taligent. He is responsible for the exchange of technical information and the resolution of technical issues between IBM and Taligent. Mr. Hoffman holds a BSc from the University of Sheffield in the U.K. and a PhD in Mathematics from the University of Texas.

Object-Oriented Programming with SOM/6000



By Debora Blakely-Fogel

This article introduces the basic concepts of programming with SOM/6000 to help you begin developing reusable software.

The Object Management Group (OMG) has set forth its Common Object Request Broker Architecture (CORBA) standards to provide interchangeability of objects. IBM's System Object Model/6000 (SOM/6000), now available with AIX 3.2, fully conforms to these standards.

There are two primary ways to provide the services required by computer applications:

- **The functional or procedural approach to programming** provides one or more libraries of functions and procedures with which programmers can operate on device drivers, files, windows, and so on.
- **The object-oriented approach to programming** provides one or more libraries of objects that are computer abstractions of the real objects. These objects can interact with each other. For example, the object movie is a software unit that can be played, stopped, and edited. It can also interact with an audio track object to provide synchronized audio and video. The current trend in software engineering is to move toward object-oriented programming.

The object-oriented approach has several advantages. *Encapsulation*, the key to object-oriented programming, ensures data operations are performed only on appropriate data. Figure 1 illustrates a travel object whose definition is kept separate from its many possible implementations (such as bicycle, airplane, covered wagon, and so on). Class descriptions can be modified with last minute design changes without having to recode major sections of the application. Consistent user interfaces across many applications are possible by sharing common objects. Reusing the same objects reduces coding time and increases quality.

Hardware engineers do not redesign a circuit board from scratch. They select from a library of prebuilt components with well-defined interfaces and link these together. Object-Oriented (OO) programming employs lessons learned from the more mature field of hardware engineering.

Figure 2 shows one of the key concepts of OO programming—*inheritance*—by sharing common features of the wheel object among bicycle wheels and horse-cart wheels.

Object-oriented programming, however, has some disadvantages. Most OO programming languages have a tight binding between the object and the programs that use the object. If an object changes, it is often necessary (as in the case of C++) to recompile the application. The binary-level interface to the object is not portable to other language execution environments. Programs written in one language cannot use objects from another language.

IBM's SOM was developed to address the problems with object-oriented programming while preserving its benefits. The Interface Definition Language (IDL) allows any supported programming language to implement object methods, while using a different language for the client application. Figure 3 shows that objects within a SOM object library can be implemented in any supported language.

Objects can be changed or used in different environments without recompiling the code that implemented the objects. The Distributed SOM


Debora Blakely-Fogel

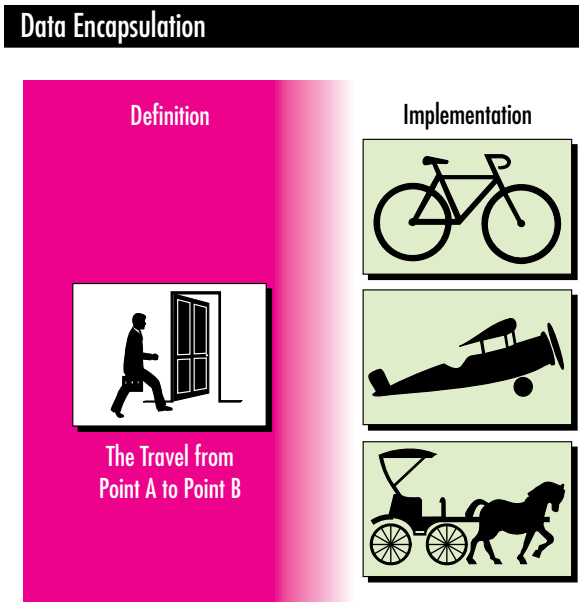


Figure 1. Data encapsulation—the key to object-oriented programming

(DSOM) framework provides the foundation for distributed objects. Currently, DSOM is based on sockets, but will be integrated with the Distributed Computing Environment (DCE) in a future release.

Frameworks

In addition to SOM itself (the SOM compiler and the SOM runtime library), the SOMObjects Developer Toolkit provides a set of frameworks (class libraries) that can be used in developing object-oriented programs. These frameworks include DSOM, Interface Repository, Persistence, Replication, Emitter, and Event Management.

DSOM Framework

DSOM enables application programs to access SOM objects across address spaces. Application programs can access objects in other processes on the same or different machines. DSOM provides this transparent access to remote objects through its Object Request Broker (ORB). The location and implementation of the object are hidden from the client—the client accesses the object as if it were local. Currently, DSOM supports the distribution of objects among processes within a workstation (Workstation DSOM) and across a local area network consisting of OS/2 and AIX systems (Workgroup DSOM).

Figure 4 shows a client application accessing an object in a different application through the DSOM ORB.

Interface Repository Framework

The Interface Repository is a database that holds all the information contained in the IDL description of a class of objects. It consists of the 11 classes defined in the CORBA standard for accessing the Interface Repository. Thus, the Interface Repository framework provides runtime access to all information contained in the IDL description of a class of objects.

Persistence Framework

The Persistence framework is a collection of SOM classes that provides methods for saving objects and later restoring them. Since objects can be stored in either a file or a more specialized repository, the state of an object can be preserved beyond the termination of the process that creates it. This facility is useful for constructing object-oriented databases, spreadsheets, and other applications that store and later modify data. The Persistence framework includes the following abilities:

- Objects can be stored individually or in groups.
- Objects can be stored in default formats or in specially designed formats.
- Objects of arbitrary complexity can be saved and restored.

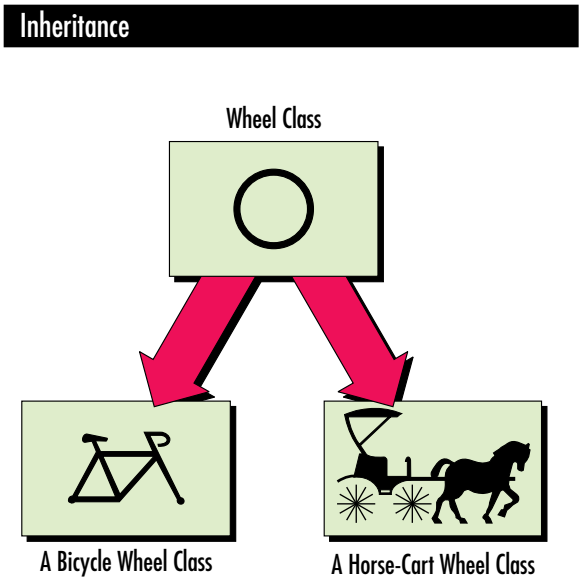


Figure 2. Class inheritance enables programmers to reuse existing functionality

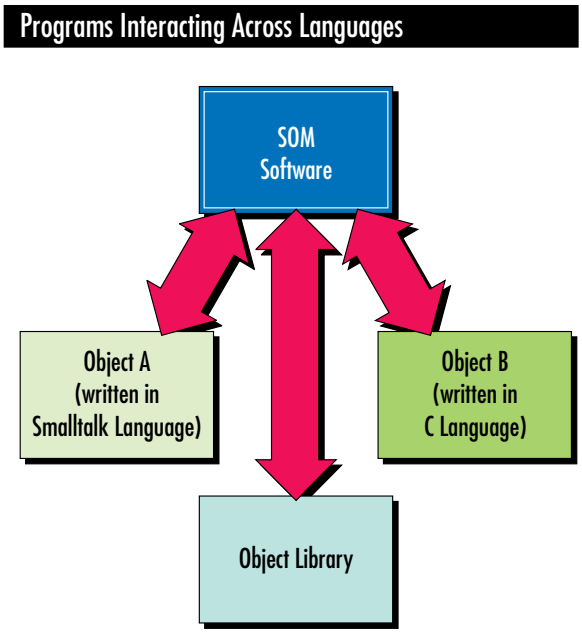


Figure 3. SOM enables programs to interact across language barriers

Replication Framework

The Replication framework is a collection of SOM classes that allows a replica or copy of an object to exist in multiple address spaces while maintaining a single-copy image. In other words, an object can be replicated in several different processes while logically it behaves as a single copy. Updates to any copy are propagated immediately to all other copies. The Replication framework handles locking, synchronization, and update propagation, and guarantees consistency among the replicas.

The Replication framework can be exploited only if the applications are structured appropriately. The recommended structure is similar to the Model-View-Controller paradigm used by Smalltalk programmers. The Replication framework proposes a View-Data paradigm. The data object has whatever "state" information the application desires to store in it. The view object has no state, but has methods to show a rendition of the state contained in the data object. In addition, it may have some data that pertains to the image being displayed to the user. For example, in a visual presentation, the colors used for different regions may be in the view object while the content information comes from the data object.

The view and data must have a protocol between them so that when the data object changes, a signal is sent to the view object to note the change and refresh the display. This protocol can be extended to multiple views on the same data object, whereby an update to the data object is automatically seen in all visual presentations. Effectively, the views "observe" the data.

The Replication framework is concerned with data objects only. Application developers must implement the "observation" protocol between the views and the data. The Replication framework requires that data objects be derived from a distinguished framework class `SOMRReplicbl`. Figure 5 shows an object—Lassie—that is replicated across *n* processes. Notice that the object Lassie is an instance of the class `Replicated Dog`, which is a subclass of the classes `Dog` and `SOMRReplicbl`.

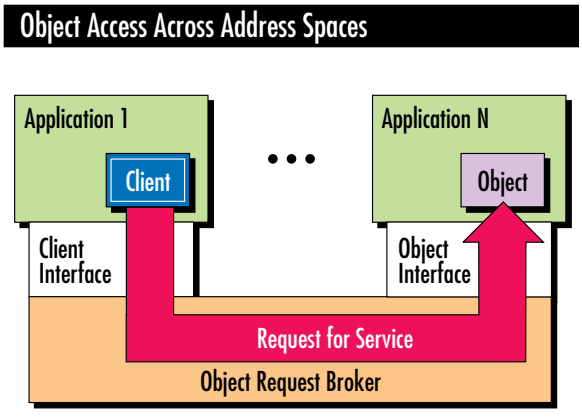


Figure 4. DSOM enables object access across address spaces

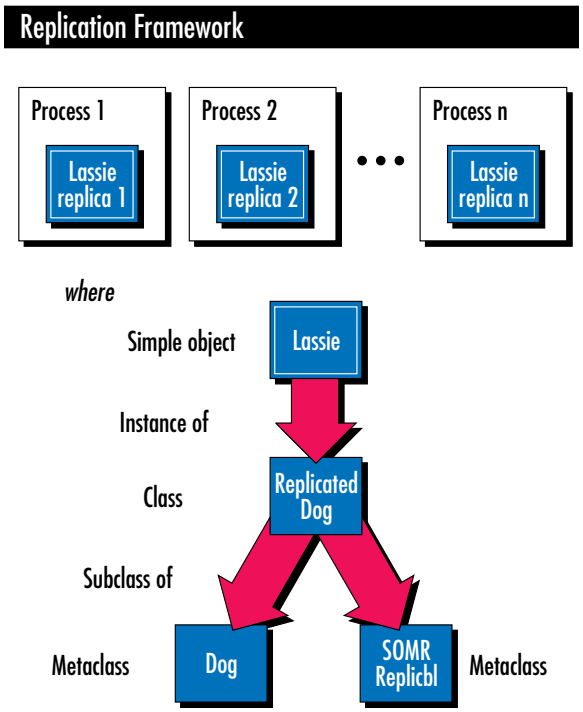


Figure 5. Replication framework

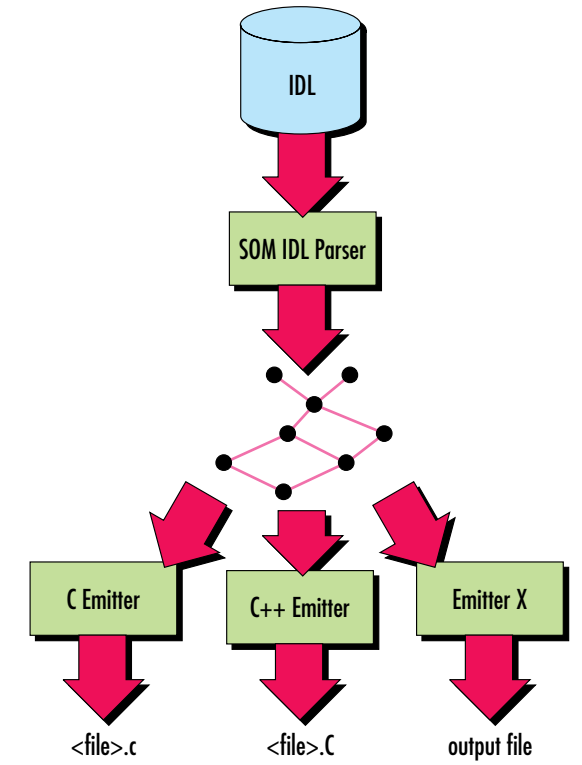


Figure 6. SOM compiler translates IDL files to required output

Emitter Framework

The Emitter framework is a collection of SOM classes that allows programmers to write their own emitters. *Emitter* describes a back-end output component of the SOM compiler. Each emitter input is information about an interface, generated by the SOM compiler as it processes an IDL specification and produces output organized in a different format.

SOM provides a set of emitters that generate the binding files for C and C++ programming (header files and implementation templates). Figure 6 shows the ability of the SOM compiler to generate source files (such as `<file>.c` and `<file>.C`) based on the emitter definition by the user. In addition, developers can write their own special-purpose emitters. For example, an implementor could write an emitter to produce documentation files or binding files for programming languages other than C and C++.

Event Management Framework

The Event Management framework is a central facility for registering all events of an application. This registration facilitates grouping various application events and waiting on multiple events in a single-event processing loop. Replication framework and DSOM use this facility to wait on their respective events. Any interactive application that uses DSOM or replicated objects must also use Event Management framework.

These frameworks (class libraries) can provide OO programmers with the following capabilities:

- The ability to access objects across address spaces using the DSOM framework
- CORBA-compliant runtime access to all information contained in the IDL description of a class of objects using the Interface Repository framework
- Save and later restore objects using the Persistence framework
- A replica (copy) of an object can exist in multiple address spaces using the Replication framework
- A central facility for registering all events of an application using the Event Management framework
- Tools to write emitters using the Emitter framework

Implementing SOM Classes

The IDL specification for a class defines only the interface to the instances of the class. The implementation of those objects—the procedures that perform their methods—is defined in an *implementation* file. To assist users in implementing classes, the SOM compiler produces a template implementation file, a type-correct guide for how the implementation of a class should look. The class implementor then modifies this template to implement the class' methods.

The SOM compiler can also update the implementation file to reflect later changes made to a class' interface definition file (`.idl`). These incremental updates include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. These updates to the implementation file do not disturb existing code in the method procedures.

The Implementation Description and Template

A programmer may implement a simple class `Hello` with one method, `sayHello`, that prints the string "Hello World!" when invoked from a client

program. Figure 7 illustrates an implementation description file, `hello.idl`, that defines the necessary IDL specifications for the class `Hello` and its method `sayHello`. Figure 8 illustrates the SOM compiler-generated template file, `hello.c`, produced from `hello.idl`.

The first line of the implementation template (Figure 8) defines the `Hello_Class_Source` symbol. It is used in the SOM-generated implementation header files for C to determine when to define various functions, such as `HelloNewClass`. For interfaces defined within a module, the directive `#define <className>_Class_Source` is replaced by the directive `#define SOM_Module_<moduleName>_Source`. The second line includes the SOM-generated implementation header file. This file defines a struct holding the instance variables for the class, macros for accessing instance variables, macros for invoking parent methods, and so on.

For each method introduced or overridden by the class, the implementation template includes a stub procedure—a procedure that is empty except for an initialization statement, a debugging statement, and possibly a return statement. The stub procedure for a method is preceded by any comments that follow the method's declaration in the IDL specification.

The `SOM_Scope` symbol is defined in the implementation header file as either `extern` or `static`, as appropriate. The term `void` signifies the return type of method `sayHello`. The `SOMLINK` symbol defined by SOM represents the keyword needed to link to the C or C++ compiler. Its value is system-specific. Using the `SOMLINK` symbol allows the code to work with a variety of compilers without modification.

Following the `SOMLINK` symbol is the name of the procedure that implements the method. After the procedure name is the formal parameter list for the method procedure. Because each SOM method always receives at least one argument (a pointer to the SOM object that responds to the method), the first parameter name in the prototype of each stub procedure is called `somSelf`. The `somSelf` parameter is a pointer to an object that is an instance of the class being implemented (`class Hello` in this example) or an instance of a class derived from it.

Unless the IDL specification of the class includes the `callstyle=oidl` modifier, the formal parameter list includes one or two additional parameters (prescribed by the CORBA standard)

```
#include <somobj.idl>
interface Hello : SOMObject
{
     void sayHello();
     // This method outputs the string "Hello, World!".
};
```

Figure 7. Interface definition file

```
#define Hello_Class_Source
#include <hello.ih>
/* This method outputs the string "Hello, World!" */
SOM_Scope void SOMLINK sayHello(Hello somSelf, Environment *ev)
{
     /* HelloData *somThis = HelloGetData(somSelf); */
     HelloMethodDebug("Hello", "sayHello");
}
```

Figure 8. Implementation file

before the parameters declared in the IDL specification:

- An (`Environment *ev`) input/output parameter that permits the return of exception information
- A (`Context *ctx`) input parameter if the IDL specification of the method includes a context specification

The first statement in the stub procedure is enclosed in comments only when the class does not introduce any instance variables. The purpose of this statement is to initialize a local variable (`somThis`) that points to a structure representing the instance variables introduced by the class. The macros defined in the Hello implementation header file use the `somThis` pointer to access those instance variables.

Next in the stub procedure is a statement to facilitate debugging. The `HelloMethodDebug` macro is defined in the implementation header file. It takes two arguments: a class name and a method name. If debugging is turned on, the macro produces a message each time the method procedure is entered.

The way in which the stub procedure ends is determined by whether the method is a new or overriding method. For new methods, the stub procedure ends with a `return` statement unless the return type of method is `void`. For overriding methods, the stub procedure ends by making a "parent method call" for each of the class' parent classes. If the method has a return type that is not

void, the last of these parent method calls is returned as the result of the method procedure. The class implementor can customize this return statement if needed.

Extending the Implementation Template

To implement a method, add code to the body of the stub procedure. In addition to standard C or C++ code, class implementors can also use any of the functions, methods, and macros provided by SOM for manipulating classes and objects. SOM provides two facilities especially for class implementors: accessing instance variables of the object responding to the method, and making parent method calls.

SOM provides a macro that simplifies the process of converting C++ classes to SOM classes.

To access internal instance variables, class implementors can use either the short form, _variableName, or the long form, somThis->variableName. To access internal instance variable a, for example, the class implementor could use either _a or somThis->a. The somThis pointer must be properly initialized in advance using the <className>GetData procedure, as illustrated in the previous example by the first statement in the stub procedure.

Instance variables can be accessed only within the implementation file of the class that introduces the instance variable, not within the implementation of subclasses or within client programs. To allow access to instance data from a subclass or from client programs, use an attribute rather than an instance variable to represent the instance data.

In addition to macros for accessing instance variables, the implementation header file generated by the SOM compiler contains definitions of macros for making parent method calls. When a class overrides a method defined by one or more of its parent classes, the new implementation often just needs to augment the functionality of the existing implementations. Rather than completely reimplementing the method, the overriding method procedure can invoke the procedure that one or more of the parent classes uses to implement that method, then perform additional computation as needed. The parent method call can occur anywhere within the overriding method. For example, for class Hello with parents File and Printer and overriding method somInit, the SOM compiler defines these macros:

- Hello_parent_Printer_somInit
- Hello_parent_File_somInit
- Hello_parents_somInit

Each macro takes the same number and type of arguments as Hello. Using the macro Hello_parent_File_somInit invokes File's implementation of somInit. The parent macro invokes the parent method for each parent of the child class that supports the method Name. Therefore, Hello_parents_somInit would invoke both File's and Printer's implementation of somInit.

For C++ programmers implementing SOM classes, SOM provides a macro that simplifies the process of converting C++ classes to SOM classes. This macro allows the implementation of one method of a class to invoke another new or overriding method of the same class on the same receiving object. This is done by using _methodName(arg1, arg2, ...), the shorthand syntax, rather than the long form somSelf->methodName(arg1, arg2, ...). To use the shorthand syntax, the macro METHOD_MACROS must be defined prior to including the .xih file for the class.

Refining the .idl file for a class is typically an iterative process. As mentioned earlier, the SOM compiler assists in this development by reprocessing the .idl file and making incremental updates to the current implementation file. The incremental update includes these changes:

- Stub procedures are inserted into the implementation file for any new methods added to the .idl file.
- New comments in the .idl file are reformatted appropriately and transferred to the implementation file.
- If the interface to a method has changed, a new method procedure prototype is placed in the implementation file. As a precaution, the old prototype is also preserved within comments.

To ensure that the SOM compiler can properly update method procedure prototypes in the implementation file, class implementors should adhere to editing the following types of changes:

- A method procedure name should not be enclosed in parentheses in the prototype.
- A method procedure name must appear in the first line of the prototype, excluding comments and white space. A new line must not be inserted before the procedure name.

Using SOM Classes

Figure 9 shows how an object variable is declared and two different methods for creating instances of a class.

Declaring a SOM Variable

When declaring an object variable, an object interface name defined in IDL is used as the type of variable. In this example, the declaration `Bye obj;` declares `obj` to be a pointer to an object that has type `Bye`. Because the sizes of SOM objects are not known at compile time, instances of SOM classes must always be dynamically allocated. Therefore, a variable declaration must always define a pointer to an object.

In SOM, objects of this type are instances of the SOM class named `Bye`, or of any SOM class derived from this class. All SOM objects are type `SOMObject`, even though they may not be instances of the `SOMObject` class. If the type of object to which the variable will point is not known at compile time, the object can be declared to be type `SOMObject`.

Creating Instances of a SOM Variable

SOM provides the `<className>New` and the `<className>Renew` macros for creating instances of a class. These macros are illustrated in the above example. The `ByeNew` macro allocates enough space for a new instance of `Bye`; creates a new, initialized class instance; and returns a pointer to it. The `ByeNew` macro automatically creates the class object for `Bye`, as well as its ancestor classes and metaclass, if needed. After a client program has finished using an object that was created with the `<className>New` macro, the object should be freed by invoking the method `somFree` on it.

The `<className>Renew` macro is used only when the space for the object has been allocated previously. This macro converts the given space into a new, initialized instance of `<className>` and returns a pointer to it. The argument of `<className>Renew` must point to a block of storage large enough to hold an instance of class `<className>`. The SOM method `somGetInstanceSize` can be invoked on the class to determine the amount of memory required.

In addition, using the `<className>Renew` macro requires that the class object be already created. The C and C++ usage bindings for a SOM class provide static linkage to a `<className>NewClass` procedure that can create the class object.

```
#include <hello.h>
main()
{
   Bye obj; /* A pointer to a "Bye" Object */
   SOMClass helloCls; /* A pointer for the Hello class object
   */
   Hello objA[10]; /* an array of Hello instances */
   unsigned char *buffer;
   int i;
   int size;

   /* Create the Hello class object: */
   helloCls = HelloNewClass(Hello_MajorVersion,
   Hello_MinorVersion);

   /* Get the amount of space needed for a Hello instance */
   size = _somGetInstanceSize(helloCls);
   size = ((size + 3)/4)*4; /* round up to doubleword multiple
   */

   /* Allocate the total space needed for ten instances */
   buffer = SOMMalloc(10*size);

   /* Convert the space into ten separate Hello instances */
   for (i=0; i<10; i++)
    objA[i] = HelloRenew(buffer + i*size);
   ...

   ...
   /* Instantiate the Bye class object */
   obj = ByeNew();
   ...

   ...
   /* Free the space used for the Bye and Hello class objects.
   */
   somFree(obj);
   for (i=0; i<10; i++)
    _somUninit(objA[i]);
   SOMFree(buffer);
```

Figure 9. Object variable declaration and class instantiation

In Figure 9, many instances of the `Hello` class will be created. Therefore, the function `HelloNewClass` is used to create the `Hello` class object. Once the class object has been created, the example invokes the method `somGetInstanceSize` on this class to determine the size of a `Hello` object, uses `SOMMalloc` to allocate storage, and then invokes the `HelloRenew` once for each object to be created rather than performing separate memory allocations using the `HelloNew` call.

When an object created with the `<className>Renew` macro is no longer needed, its storage must be freed. Our example uses the `SOMMalloc` function to allocate memory and must

```
void setMany(in short start, in short numArgs, in va_list ap);
_setMany(aVector, somGetGlobalEnvironment(), 2, 4, 20, 12,
 32, 41);
```

Figure 10. Method invocation with a variable number of arguments (short form)

```
va_list start_ap, ap;
Vector aVector = VectorNew();
...
start_ap = ap = (char *) SOMMalloc(4 * sizeof(long));
va_arg(ap, long) = 20;
va_arg(ap, long) = 12;
va_arg(ap, long) = 32;
va_arg(ap, long) = 41;
Vector_setMany(aVector, somGetGlobalEnvironment(), 2, 4,
 start_ap);
```

Figure 11. Method invocation with a variable number of arguments (long form)

use the corresponding SOMFree function to free the objects' storage. Before doing this, the objects in the region to be freed should be deinitialized by invoking the somUninit method on them. This allows each object to free any memory that it may have allocated without the program's knowledge. The somFree method also calls the somUninit method.

Invoking Methods on Objects

To invoke a method in C, a client programmer can use the macro _<methodName>. To avoid possible ambiguity, the programmer can also use the long form of this macro,<className><methodName>. In C, calls to methods defined using IDL require at least two arguments: a pointer to the receiving object and a value of type (Environment *). The Environment data structure specified by CORBA passes environmental information between a caller and a called method.

If the IDL specification of the method includes a context specification, then the method has an additional implicit context parameter. When invoking the method, this argument must follow immediately after the Environment pointer argument. None of the SOM-supplied methods require context arguments. The Environment and context method parameters are recommended by the CORBA standard.

If the IDL specification of the class that introduces the method includes the callstyle=oidl modifier, the (Environment *) and context arguments should not be supplied when invoking the method; that is, the receiver of the method call is followed immediately by the arguments to the method. Some classes supplied in the SOMObjects Developer Toolkit are defined in this way to ensure compatibility with the previous release of SOM.

If a C expression is used to compute the first argument to a method call, an expression without side effects must be used, because the first argument is evaluated twice by the _<methodName> macro expansion. In particular, a somNew method call or a macro call of <className>New cannot be used as the first argument to a C method call, because that would create two new class instances rather than one.

As a convenience, methods whose final argument is type va_list can be invoked by specifying a variable number of arguments, as shown in Figure 10.

Since this is the short form of the invocation macro, the first variable-arguments form illustrated is available only in the absence of ambiguity. Alternatively, the long-form macro, which is always available, requires a va_list, as shown in Figure 11.

Obtaining a Method's Procedure Pointer

Method resolution is the process of obtaining a pointer to the procedure that implements a particular method for a particular object at runtime. The method is then invoked subsequently by calling that procedure and passing the method's intended receiver—the Environment pointer, the context argument, and the method's other arguments, if any. C and C++ programmers may wish to obtain a pointer to a method's procedure for efficient repeated invocations.

Obtaining a pointer to a method's procedure can be done in one of two ways, depending on whether the method is to be resolved using offset resolution or name-lookup resolution. Obtaining a method's procedure pointer via offset resolution is faster, but it requires that the name of the class that introduces the method and the name of the method be known at compile time. It also requires that the method be defined as part of that class's interface in the IDL specification of the class.

Using offset resolution to obtain a pointer to a procedure, the C/C++ usage bindings provide the SOM_Resolve and SOM_ResolveNoCheck macros. The usage bindings themselves use the first of these, SOM_Resolve, for offset-resolution method calls. The difference between the two macros is that the SOM_Resolve macro performs consistency

checking on its arguments, but the macro SOM_ResolveNoCheck does not. Both macros require the same arguments:

receiver: Object to which the method will apply (It should be specified as an expression without side effects.)

className: Name of the class that introduces the method

methodName: Name of the desired method

The last two names (className and methodName) must be given as tokens, rather than strings or expressions.

To obtain a pointer to a method's procedure using name-lookup resolution, use the somResolveByName procedure or any of the somLookupMethod, somFindMethod, or somFindMethodOK methods. These methods are invoked on a class object that supports the desired method. They take an argument specifying the somId for the desired method, which can be obtained from the method's name using the somIdFromString function.

In addition to methods, SOM objects can also have attributes. An *attribute*, an IDL shorthand for declaring methods, does not necessarily indicate the presence of any particular instance data in an object of that type. Attribute methods are called get and set methods. For example, if a class Hello declares an attribute called msg, then object variables of type Hello will support the methods _get_msg and _set_msg to access or set the value of the msg attribute. Read-only attributes have no set method. The get and set methods are invoked in the same way as other methods.

Using Class Objects

Using a class object can involve getting the class of an object, creating a new class object, or simply referring to a class object through a pointer.

Getting the Class of an Object

To get the class for which an object is an instance, SOM provides a method called somGetClass. The somGetClass method takes an object as its only argument and returns a pointer to the class object of which it is an instance. Getting the class of an object is useful for obtaining information about the object. In some cases, such information cannot be obtained directly from the object, but only from its class.

A class can override the somGetClass method to provide enhanced or alternative semantics for its objects. Because it is usually important to respect the intended semantics of a class of objects, the somGetClass method should normally be used to access the class of an object.

In a few special cases, it is not possible to make a method call on an object to determine its class. For such situations, SOM provides the SOM_GetClass macro. In general, the somGetClass method and the SOM_GetClass macro may have different behavior. Although this difference may be limited to side effects, it is also possible for their results to differ. The macro SOM_GetClass should be used only when absolutely necessary.

Creating a Class Object

A class object is created automatically the first time the <className>New macro is invoked to create an instance of that class. In other situations, it may be necessary to create a class object explicitly.

As seen earlier, it is sometimes necessary to create a class object before creating any instances of the class. For example, creating instances using the <className>Renew macro or the somRenew method requires knowing how large the created instance will be, so that memory can be allocated for it. Obtaining this information requires creating

Vendors That Have Announced Plans to Support SOM

Independent Software Vendors continue to sign up to support IBM's distributed object computing environment.

American Management Systems
Borland International®, Inc.
Chip Chat Cawthon Software®
Cirrus Technology, Inc.
Continuum Company, Inc.
Digitalk, Inc.
Easel Corporation
Footprint Software, Inc.
Hewlett-Packard
Inference® Corporation
Information Advantage®, Inc.
Intermedia Development Company, Inc.
KASEWORKS™, Inc.
MetaWare®, Inc.
Microformatic
Object Design®, Inc.
ParcPlace® Systems, Inc.
Raleigh Systems, Inc.
Sundial Corporation
SunSoft, Inc.
Watcom™ International Corporation

Implementing SOM Classes:

1. Create an IDL file to define the interface to the objects of the new class
2. Use the SOM compiler (`sc`) to generate the following files:
   - implementation template (`.c`)
   - definition (`.h`)
   - implementation header (`.ih`)
3. Customize the implementation template

Using SOM Classes:

1. Create a client program that uses the class
2. Compile and link the client code with the class implementation
3. Execute the client program

Figure 12. Steps for programming with SOM

the class object. As another example, a class object must be explicitly created when a program does not use the SOM bindings for a class. Without SOM bindings for a class, its instances must be created using `somNew` or `somRenew`. These methods require that the class object be created in advance.

Referring to Class Objects

The `<className>NewClass` procedure initializes the SOM runtime environment, creates the class object (if necessary), creates class objects for the ancestor classes and metaclass of the class (if necessary), and returns a pointer to the newly created class object. After it is created, the class object can be referenced in client code using the macro `_<className>`.

The procedure takes two arguments: the major and the minor version number of the class. These numbers are checked against the version numbers built into the class library to determine if the class is compatible with the client's expectations. The class is compatible if it has the same major version number and the same or higher minor version number. Major version numbers usually change only when a significant enhancement or incompatible change is made to a class. Minor version numbers change when minor enhancements or fixes are made. Downward compatibility is usually maintained across changes in the minor version number.

The `somFindClass` or `somFindClsInFile` methods can be used to create a class object when not using the C or C++ language bindings for the class, or when the class name is not known at compile time. Details about using these methods can be found in the *SOMObjects Developer Toolkit User's Guide*.

Summary

SOM classes, designed to be language neutral, can be implemented in one programming language and used by programs written in another language. To achieve language neutrality, the interface for a class of objects must be defined separately from its implementation. Each of these processes, implementing SOM classes and using SOM classes, involves three basic steps shown in Figure 12.

Debora Blakely-Fogel, IBM Corporation, RISC System/6000 Division, 11400 Burnet Road, Austin, TX 78758. Ms. Blakely-Fogel is an advisory programmer responsible for providing AIX technical assistance to software vendors. She has a BS in Mathematics from the University of Massachusetts at Lowell and an MS in Computer Science from New Mexico State University.

IBM World Wide Web Servers Online

As mentioned in the editorial to the May 1994 *AIXpert*, the IBM World Wide Web servers are now online and available for access. The amount of information online is growing each week, so check frequently to keep up to date. Included in the online information is a version of the May issue of *AIXpert* for browsing. The June 1994 and the February 1994 issues will be available shortly, as well as printable PostScript® versions of the articles in those editions. Older issues will be posted as time permits.

To access the main IBM home page, use the URL of: `http://www.ibm.com/`

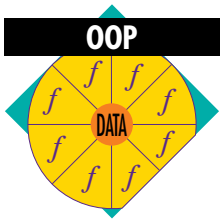
To access the IBM Austin home page directly, use the URL of:

`http://www.austin.ibm.com/`

From the IBM Austin home page, choose the following buttons to get to the *AIXpert* information:

Services and Support
POWER Developer
Technical Library
AIXpert

Creating Object-Oriented Solutions



By Peg MacPhail and George Noren

This article defines a road map that can guide you into the world of object-oriented application programming on AIX, OS/2, and OS/400. The road map uses non-proprietary, open technologies that exist today.

IBM and other companies, such as Hewlett-Packard, Sun, Apple, and Microsoft, are investing heavily in object technology. With such extensive development effort focused on object technology, we can expect new and innovative programs to solve previously complex problems, as well as new programs to replace existing procedure-oriented applications.

This transition to an object-oriented environment will take years, during which time a mixture of object-oriented and procedural programs will coexist. Object-oriented solutions that appear early in this transition period will have an advantage over later solutions by establishing the standards and shaping the object-oriented market of the future.

Background

Figure 1 shows the evolution from objects to object-oriented frameworks. Object programming uses components called *classes* as building blocks that can be used in programs—similar to integrated circuits in circuit boards. An object-oriented class provides canned code (called *methods* or *behaviors*) and attributes for a specific application component such as a device driver. An object class designer could create a general device driver class and then further refine it using *subclasses* to define any particular methods or attributes of a specific device driver. These components or classes can be used either directly by an application or customized by creating further subclasses.

The developer of a large object-oriented application is likely to deal with many different object classes. Since reusing code is a major objective, it is vital to keep track of these class definitions. Class libraries were developed to organize collections of class definitions for easy access. Usually, object-oriented class libraries provide methods and attributes for related sets of application components. For example, a set of device driver classes could be packaged as a class library.

Application frameworks provide sets of pre-integrated objects that perform specific tasks. They multiply the productivity gain of object classes by providing a higher level of prefabricated code. Application frameworks can be used in developing user interfaces or performing functions such as printing and communication. Typically, an application framework provides a basic set of functions for a task that the developer can customize and extend.

A Road Map for Cross-Platform Development

Figure 2 shows a common, cross-platform suite of high-level Application Program Interfaces (APIs) and services. These complementary APIs and services are open, non-proprietary, and intended for a wide range of industry platforms. For example, C++ and System Object Model (SOM) are rapidly becoming the object-oriented technology of choice for programming language, object distribution, and object packaging. We expect OpenDoc and the Taligent frameworks to set the standard for compound- document and framework technology.

OpenDoc technology comes from Component Integration Laboratories (CI Labs), a consortium being formed by Apple, Novell, WordPerfect, IBM, and others. OpenDoc provides the state-of-the-art compound-document support with visualization, Open Scripting Architecture (OSA), document storage (Bento), and SOM technologies. CI Labs will provide source code much like the X consortium provides code for X-Windows

George Noren

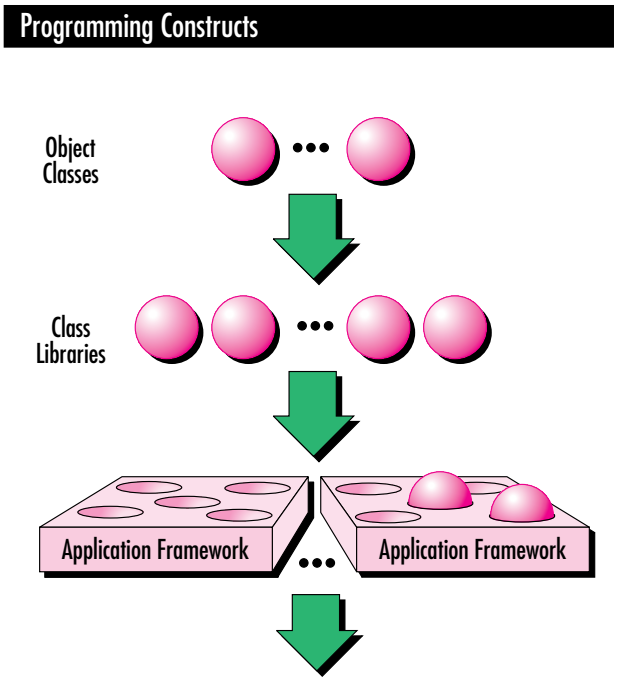


Figure 1. Object-oriented programming constructs

today. Developers can use the OpenDoc SOM object classes to create OpenDoc versions of their applications. The pervasiveness of OpenDoc and SOM means that OpenDoc applications are easy to port and can interoperate extensively, including two-way interoperation with Microsoft's OLE compound documents.

The OpenDoc environment includes both new object-oriented application programs as they are developed, and existing procedure-based application programs. A procedural application must be changed to be enabled on SOM so it can be easily moved to other platforms. It must also be changed to be enabled on OpenDoc, which allows it to interchange data with other programs that support OpenDoc.

The PowerOpen™ Environment (POE), defined by the PowerOpen Association, has been adopted by IBM and the other PowerOpen Association members (including Motorola™, Apple, Bull®, and Thomson-CSF™) as a strategic, cross-platform support environment for procedural applications. The POE consists of the base PowerOpen Application Binary Interface (ABI) and two graphical presentation engines: Macintosh Application Services (MAS) and OSF/Motif®. The PowerOpen ABI is a proper superset of the Spec 1170 work, a common UNIX specification that has been submitted to X/Open for approval. It provides a consistent binary interface across multivendor platforms for procedural applications. The PowerOpen ABI also provides a cross-industry operating system foundation for the emerging object-oriented application framework.

The SOM and OpenDoc components, shown in Figure 2, help to integrate applications across the different platforms. A SOM component provides a distributed object computing technology that is compliant with the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA). This component allows objects from different types of class libraries—such as Smalltalk, C++, and C—to interoperate.

SOMObjects Developer Toolkit has been shipping on AIX and OS/2 since mid-1993. Because SOM is available on so many operating environments, interoperability among objects is a reality. Even if SOM is not on a platform, messaging can occur between SOM and the other Object Request Broker (ORB) if the platform has an OMG CORBA-compliant ORB.

A program can support compound documents by generating OpenDoc parts. Each part supports a specific set of methods, such as those used to open or edit its contents. Since each kind of part knows how to handle the data associated with it, an OpenDoc document can contain many different kinds of document parts without actually knowing how to handle the data from any of them. The OSA technology provides a higher-level messaging capability with predefined verbs, providing support for a defined interface to common services such as mail and print. OpenDoc parts are SOM-enabled.

Application Frameworks

Figure 2 shows several application frameworks that provide high-level APIs and services for programs running on many platforms. The following examples describe these SOM-enabled application frameworks:

- **Taligent Application Environment (TalAE)** includes object-oriented 2-D and 3-D graphics, international language support, a standard "look and feel," Taligent OS services, and multimedia support.
- **Taligent Development System (TDS)**, a set of object-oriented application development tools, allows incremental link and compile, automated build, multiple views, hyperlink navigation, integrated source-level debugging,

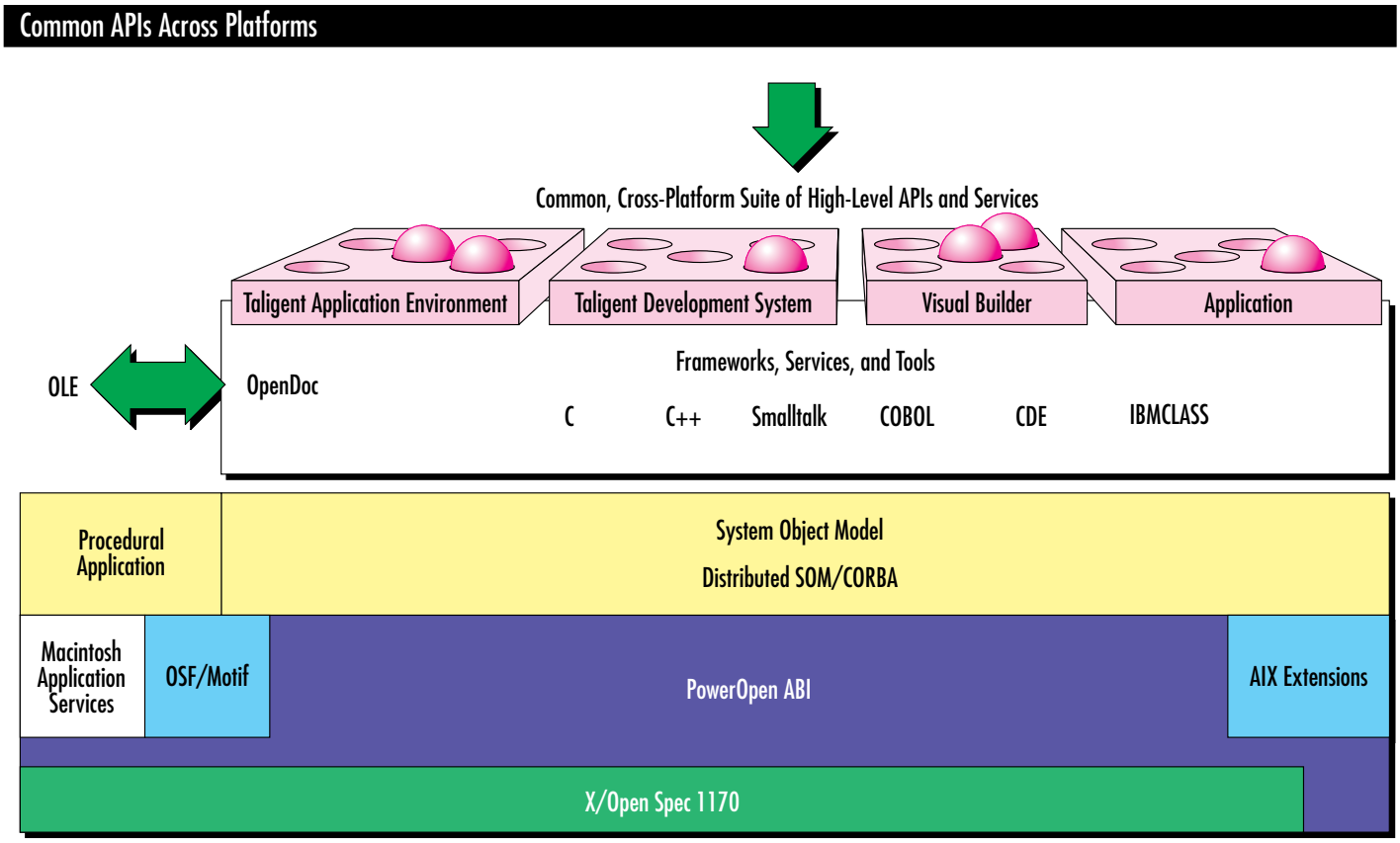


Figure 2. Object technology provides common APIs across diverse platformss

dynamic browsers, and a Graphical User Interface (GUI) builder.

- **A Visual User Interface (UI) builder** allows programmers to use graphical techniques to quickly build an application. These applications become OpenDoc parts for use within UI application frameworks.
- **VisualAge** is an integrated application development environment that provides visual programming and construction-from-components technologies. In addition, it provides support for team programming, library services, advanced GUI, communications and transaction processing, database components, access to other applications, visual SQL query builder capability, multimedia capability, and a complete application development environment including IBM Smalltalk.

C Set++ for AIX provides a robust compiler for C and C++ applications, as well as a C++ Class Browser, C++ Class Libraries, HeapView Debugger, and a test coverage tool.

Taking Advantage of OO Technology

Object-Oriented (OO) technology is a developing discipline. Although it is not yet fully available on all operating environments, you can still prepare today for its arrival. The first step is to develop object-oriented expertise. If you do not have experience with OO projects, take training classes, attend seminars, and read articles and books on the subject.

Once you have a good foundation, analyze your applications for potential improvements from OO technology. Consider not only the function your application offers to an external user, but also what your application might offer to another application. Consider subfunctions that might be useful to a user or another application. Then, model your application into one or more objects with the appropriate subfunctions that each would expose to another application or to a user.

The idea is to separate the interface from the implementation. Think of it as a contract in which the application agrees to perform a task, and is empowered to do it any (legal) way that it wants.

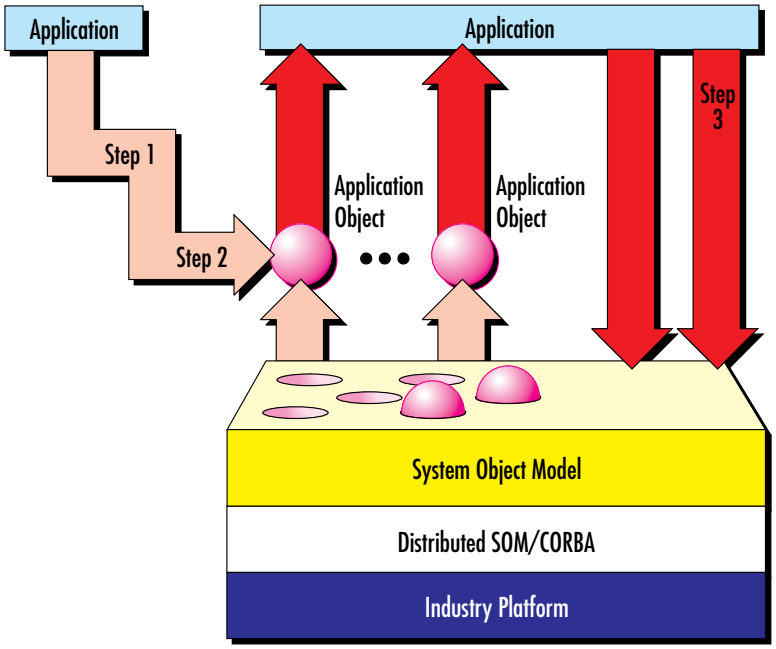


Figure 3. Migrating an existing application

It is important to set realistic expectations and goals, and make your first attempt with a non-critical application that can be easily separated and defined. The first step is to create a small, skilled OO team and develop a plan to move the application into the OO world. Once you have completed a successful project, you can then leverage that experience to other programming teams to create new applications using application frameworks. Consider using OpenDoc for applications that will run on OS/2 or Windows.

Migrating an Existing Application

Figure 3 shows a schematic of the process of moving an existing application to an OO environment. The numbers in the figure refer to the following steps involved in migrating the application:

1. Model your application design into theoretical objects.
2. Create SOM or C++ objects to make an object interface to the application.
3. Add SOM client calls to the application to use other object services.

Step 1 is a paper-and-pencil exercise. Divide your application into objects. If you have a highly modularized design, break your application into modules or groups of modules. You can then take advantage of the modular design when migrating your application. For example, if the function in `module_x` changes, you could rewrite `module_x` as an object. Calls to `module_x` in the procedural application would then be replaced by SOM client calls to the object `module_x`, bypassing the `module_x` procedural code in the application.

If you never want to change your procedural application, the objects you create to export the application's interface can be like a firewall between the object world and the application. New function can be written as objects. However, access to your program through a procedural call will not produce the new object function unless you change the procedural application to call the objects.

Steps 1 and 2 make your application available as a server to object clients. These steps do not require any changes to your procedural program. Although you may change your procedural program to make this process easier, it is not required. Steps 1 and 2 can be done independently of step 3. Step 3 changes your application to be an object client to an object server. This step can be done independently of steps 1 and 2.

Example: Migrating an Existing Application

The following example illustrates the concepts involved in migrating an application to object technology. All example code is in C; C++ would look slightly different because it supports object pointers, constructors, and destructors.

The example program presents exercises for a student on a computer. The student can stop during an exercise and finish it later. The set of exercises can be tailored to the student's performance. For example, if a student needs more emphasis on a certain topic, the processing state can be set to show that the student must do an extra exercise. The original exercise can be stopped until the additional exercise is completed, then the student can return to finish the original. Figure 4 shows pseudo-code for the structure of the program.

Step 1: Model your application design into theoretical objects. This application could be modeled into the following objects: Teacher, exercise_x, exercise_y, exercise_z, Student, and printer.

Step 2: Create SOM or C++ objects to make an object interface to the application. To do that, write the Interface Definition Language (IDL)

interface to support the teaching exercises application program. The code for the Teacher object might appear as follows:

```
#include <somobj.idl>
interface Teacher : SOMObject
{
   void teach_exercises;
   attribute string name;
};
```

The SOM compiler generates program stubs that will be filled in to call the procedural application. You would then write other IDL interfaces for the remaining objects. To make the objects externally available, register their IDL with the SOM Interface Repository and add the object methods to a dynamic link library.

Step 3: Add SOM client calls to the application to use other object services. Replace the calls to the database and print routine with SOM calls to the student and printer objects. For example, you could add code similar to Figure 5 to the application in place of the existing calls.

Example: Migrating an Existing Desktop Application

OpenDoc can be used to migrate a desktop application to object technology. The following three steps (illustrated in Figure 6) are similar to those in the previous example.

Step 1: Model your application design into compound document parts. In the previous teaching exercises example, make each exercise a document. The test questions in each section can be a compound-document part. Figure 7 shows the kind of functions that can be achieved using compound documents.

Step 2: Create OpenDoc parts for the application. The term *part* refers to an object class used by developers to derive any document part class. Developers must implement about 50 object calls for each part. The following are examples of these calls

```
void Draw(
   XMPFrame* frame,
   XMPShape* invalidShape)
XMPBoolean HandleEvent(
   XMPFrame* frame,
   XMPEventData event)
void AddDisplayFrame(
   XMPFrame* frame,
   XMPName* viewtype)
```

Step 3: Add OpenDoc calls to the application. To do this, call other OpenDoc script-enabled services such as mail, calendar, or print.

```
Teaching exercises application pseudo-code
     Global variables: processing_state structure, section_x,
     section_y structure,
          section_z structure, student_data structure, name

main routine (name)
     initialize
     get processing_state from database
     get student_data from database
     while (processing_state, exercise_status = not_done) do
          Case switch on processing_state
               Case processing_state = need_section_x: call give
               section_x()
               Case processing_state = need_section_y: call give
               section_y()
               Case processing_state = need_section_z: call give
               section_z()
          end_Case
     end_while
     if processing_state, print_status = print_section then print
     selected sections
     cleanup and return

give_section_x routine()
     get section_x from database
     fill in section x.structure with data already in student_data
     structure
     give exercise_x_to_student
     save section_x data and set new processing_state
     return;
```

Figure 4. Teaching exercises program pseudo-code

```
#include <printer_object.h>
#include <student_object.h>
printer_object objp
student_object objs
.
.
.
objp = printer_objectNew();
objs = student_objectNew();
.
.
.
get_student_data(objs,somGetGlobalEnvironment(),name);
.
.
.
_print_exercises(objp,somGetGlobalEnvironment());
.
.
.
_somFree(objp);
_somFree(objd);
```

Figure 5. Adding SOM calls to a procedural application

Using OpenDoc for Migrating Application

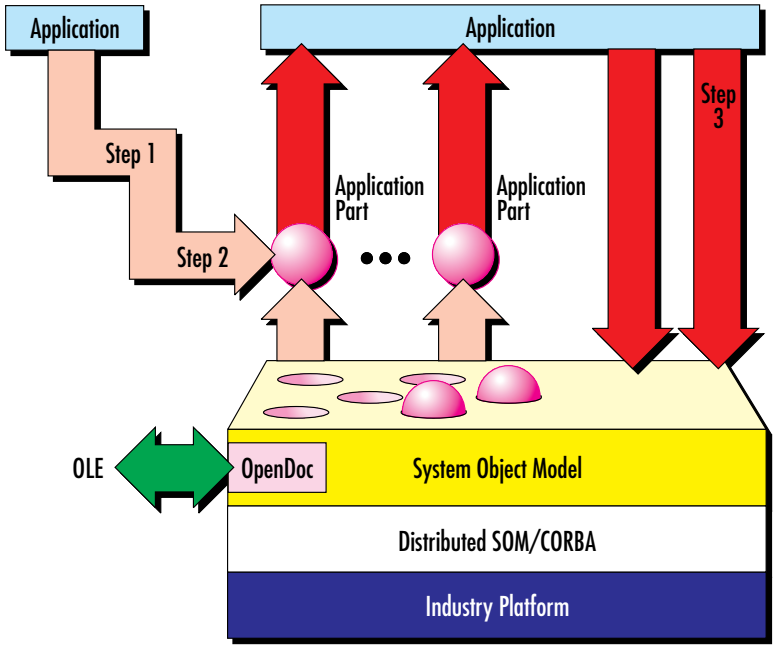


Figure 6. Migrating an application using OpenDoc

Compound-Document Screen

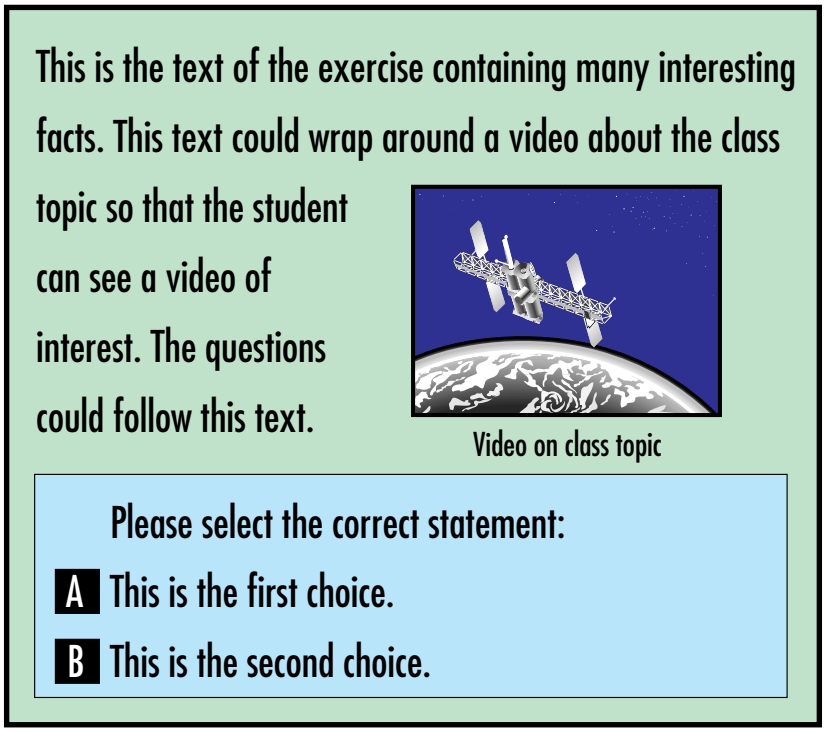


Figure 7. Example compound-document screen

Summary

IBM has already started to roll out an object-oriented, cross-platform development and run-time environment. This environment will provide interoperability between platforms and application portability across platforms (not limited to IBM platforms) that support the new environment. In the new environment, older procedure-based applications can still interoperate with objects and frameworks using the common base of SOM. Procedural applications will not, however, be as integrated into a framework as an object that is written to that framework.

Because various operating systems will support the new environment, porting an application from one operating system to another becomes very easy, usually involving only a recompile of the source. Using frameworks as building blocks in the code further reduces development and code-maintenance time and costs. Applications can be available more quickly on a larger number of platforms.

Peg MacPhail, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: macphail@austin.ibm.com. Ms. MacPhail is a technical consultant for RS/6000 solution providers. She has spent nearly 20 years with IBM in a variety of technical positions. She holds a BA from the University of Connecticut at Storrs and a MCS from Texas A&M University in College Station.

George Noren, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: geo@austin.ibm.com. Since joining IBM in 1979, Mr. Noren has written manuals for System/34, System/36, and AIX on both the RT and RISC System/6000 platforms, and was a member of the InfoExplorer design team. He has also worked as system administrator for several AIX server machines and their clients, and is currently responsible for the Prototype Evaluation Labs in Austin. Mr. Noren studied engineering at Illinois Institute of Technology. He has a BA in English from the University of Minnesota and an MBA from St. Edwards University in Austin.