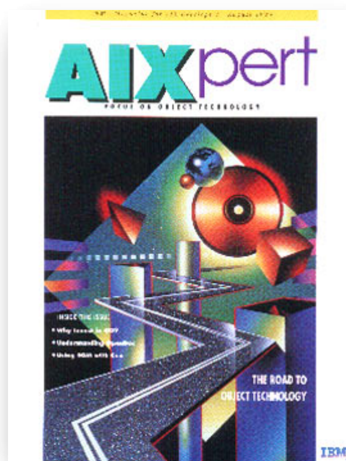
AIXpert
DIGITAL VERSION IBM

TABLE OF CONTENTS



Commentary

Wake-up Call
By George Noren

Object-Oriented Programming

Cooperation Among Metaclasses in SOM
By Ira R. Forman and Scott H. Danforth

Using SOM with C++
By Jennifer Hamilton, Robert Klarer, Mark Mendell, and Brian Thomson

Major Features Adopted by the C++ Standard Committee
By Josee Lajoie

Why Invest in Object-Oriented Programming?
By Dan Hattenberger

Standards

OpenDoc and Its Architecture
By Chris Nelson

Communications

SNA Server/6000 Performance on SMP
By Dov Bulka, Michelle Hermanson, Chris Selvaggi, and Julia Sime

AIX

Re-engineering the Time-To-Market Process
By Eddie Ho, Eric Dunn, and Peter Stoll

Signals in Multithreaded Programs
By Chary G. Tamirisa

Threads Programming in AIX Version 4
By Marc Miller

AUGUST
1995

Wake-up Call

From time to time on any journey, travelers should take time to rest, and to reassess their progress and direction. But tarry too long and you might arrive at your destination too late to be effective, or you may become distracted and not reach your destination at all. So if you have been resting too long in the "Wait & C" motel along the road to objects, this issue provides a gentle nudge to remind you that plenty is happening. It's time to get on the road again.

If you haven't started down the road yet and are concerned about startup costs of object technology, be sure to read Dan Hattenberger's enlightening article "Why Invest in OOP?". You may decide that the costs of staying home are far greater. When you decide to get started, consult our selection of OO classes that are offered in the near future to help plan a training path to get you going quickly. However, if you've already started the journey, you should be interested in the other developments that have occurred since our last issue on Object Technology (June 1994).

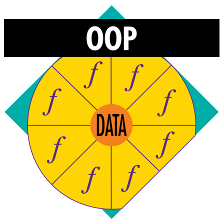
With the release of Taligent's CommonPoint™ development environment (see *AIXpert*, May 1995) and the availability of some Direct-to-SOM compilers ("Using SOM with C++" in this issue), building OO programs has received a lot of attention. Standards, too, have been improved ("Major Features Adopted by the C++ Standard Committee"). OpenDoc® is emerging as the technology that brings an open standards solution to the compound document arena. Read all about it in "OpenDoc and Its Architecture". For a deeper treatment of one of SOM's more robust features, see "Cooperation Among Metaclasses in SOM".

Dealing with other issues, we have a double-barreled coverage of threads programming. "Threads Programming in AIX Version 4" provides an excellent foundation on which to build your threads programming skills, and "Dealing with Signals in Multithreaded Programs" provides a more detailed description of one aspect of threads programming on AIX 4.x. In addition, "SNA Server/6000 Performance on SMP" looks at porting a uniprocessor application to an SMP environment with valuable insights for others that are planning such an effort. "Re-engineering the Time-To-Market Process" shows how the AIX® environment can help streamline communications between product design and manufacturing operations to get new products to market faster.

George Noren

George Noren, IBM Corporation, Internal Zip 4103, 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



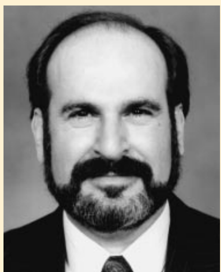
Cooperation Among Metaclasses in SOM

By Ira R. Forman and Scott H. Danforth

In SOM, metaclasses are independently programmed to impart properties to classes. As such, method table entries become resources over which metaclasses can conflict. SOM's metaclass framework implements a technique called metaclass cooperation for addressing this problem. This article describes metaclass cooperation. It also provides an example of the use of metaclass cooperation between Distributed SOM (DSOM) and Before/After Metaclasses in the SOMObjects Toolkit.

In SOM, metaclasses determine the behavior of classes, which in turn determine the behavior of ordinary objects (objects that are not classes). Both SOM metaclasses and classes are defined by subclassing. As a result, metaclasses can be referred to by name in SOM, and can be explicitly used by programmers when defining new classes and metaclasses.

The SOMObjects Toolkit (Version 2.1) contains 15 metaclasses. When new metaclasses are defined by subclassing from these, the result correctly reflects the semantics implemented by the individual metaclasses that are combined. In other words, the implementations of the different abstractions provided by the Toolkit metaclasses do not interfere with each other when they are combined by subclassing. For this reason, we describe the Toolkit metaclasses as *cooperative*.


Ira R. Forman

Scott H. Danforth

A previous paper[3] described SOM's notion of cooperation, which is summarized in the next section. This article contains an example of cooperation that combines two metaclasses from the Toolkit. The example shows that both individual Toolkit metaclasses and their combinations are useful. Cooperation between metaclasses enables this. In general, programmers of a cooperative metaclass do not need to know the other cooperative metaclasses with which it can be combined. Cooperative metaclasses are useful components for *open* object-oriented systems

The next section describes SOM's notion of cooperation (if you are not familiar with SOM, see the sidebar on page 6 before reading this article). The subsequent two sections describe useful metaclasses, followed by a section that presents a useful combination of these metaclasses based on cooperation.

Overview of Cooperation

Metaclass cooperation is the basis for creating metaclasses that can be composed. This implies that programs can be factored in new ways that lead to better modularity [5]. In addition, metaclass cooperation is essential to the proper operation of SOM, because the SOM runtime automatically derives new metaclasses as necessary to support the requirements of a new subclass. Figure 1 illustrates this based on the class declarations (in Figure 2) expressed using the SOM Interface Definition Language (IDL).

In this example, the programmer has declared class `C` as a subclass of both `A` and `B`. When the SOM runtime constructs the runtime class object `C`, it is necessary to determine the metaclass of which it will be an instance. As explained in "Reflections on Metaclass Programming in SOM[3]," the need for `C` to respond* to both the `foo` and `bar` class methods results in using a SOM-derived metaclass, denoted `DMC` in Figure 1.

When different metaclasses are automatically combined into a SOM-derived metaclass, as illustrated by Figure 1, methods or class variables

*An object responds to the methods supported by its class. The methods supported by a class are either the methods introduced by the definition of the class or the methods inherited from the class' parents.

SOM-derived Metaclasses

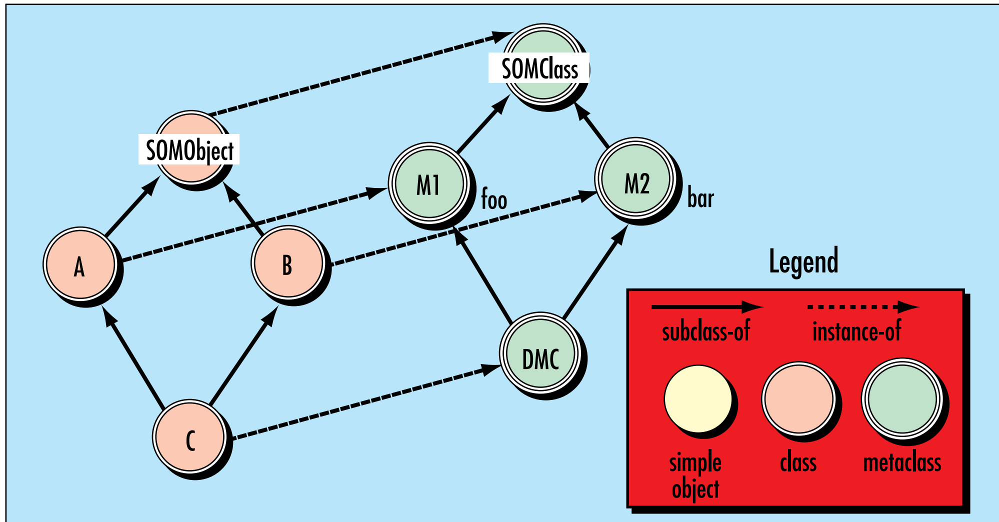


Figure 1. SOM-derived metaclasses

introduced by the different metaclasses cannot conflict with each other. This is because both the methods and instance data introduced by classes (in this case, metaclasses) are segregated into groups corresponding to their introduction of class when they are combined by using multiple inheritance. Thus, there can be no interference between `M1` and `M2` in instances of `DMC`.

In addition to introducing new class variables and methods, a metaclass programmer can also override inherited methods. Combined metaclasses can interfere with one another. For example, `somInitMIClass` is an important class method introduced by `SOMClass` (the base class for all metaclasses). This method determines the procedure pointers that are stored in the instance method table during initialization of class objects. Metaclass programmers can override `somInitMIClass` to store whatever is desired in the instance method table. But the power to explicitly load a class' instance method is a double-edged sword. How can metaclass programmers know that what they put in a class' instance method table (using a `somInitMIClass` method override) does not conflict with what another metaclass that overrides `somInitMIClass` puts there?

To ensure that a class is initialized to have the appropriate properties, SOM-derived metaclasses chain two special methods upwards to all parents to allow initialization of the class variables introduced by all ancestor metaclasses. These methods are `somInitMIClass` and `somClassReady`. This enables the composition of properties embodied

```
#include <somcls.idl>
interface M1 : SOMClass {
    // declare a metaclass that
    // introduces the foo class method
     void foo();
};
interface M2 : SOMClass {
    // declare a metaclass that
    // introduces the bar class method
     void bar();
};
interface A : SOMObject {
    // declare a class that responds to
    // the foo method
     implementation { metaclass = M1; };
};
interface B : SOMObject {
    // declare a class that responds to
    // the bar method
     implementation { metaclass = M2; };
};
interface C : A, B {
    // declare a class that inherits
    // from A and B
};
```

Figure 2. Class declaration

by metaclasses and also causes the problem of interference over the definition of methods. If different, unrelated metaclasses are used as parents of an automatically SOM-derived metaclass, then different, generally unrelated `somInitMIClass` method procedures are executed in sequence.

Overview of SOM

In SOM, classes are objects whose classes are called *metaclasses*. A class differs from an ordinary object, because a class has (in its instance data) an instance method table defining the methods to which instances of the class respond. During the initialization of a class object, a method is invoked on it, informing the class of its parents. This allows the class to build an initial instance method table. After this is done, other methods are invoked on the class to override inherited methods or to add new instance methods.

When diagramming class hierarchies, this article shows metaclasses drawn with three concentric circles, ordinary classes (classes that are not metaclasses) drawn with two concentric circles, and ordinary objects (objects that are not classes) drawn with a single circle. The initial state of an example SOM program is depicted in Figure A. There are four objects: `SOMObject` (a class), `SOMClass` (a metaclass), `Dog` (an ordinary class), and `Rover` (an ordinary object).

There are two relationships among objects that must be understood. First, there is the instance of relation between objects and classes depicted by the dashed arrow from an object to its class. When convenient, the inverse relation—class of—is also used. `SOMObject` is an instance of `SOMClass`, and `SOMClass` is the class of itself. An object's class is important because an object responds only to the methods that are supported by its class—the methods that the class introduces or inherits.

Second, there is a relationship between classes called the subclass of relation, which is depicted by the solid arrow from a class to each of its parents. `SOMClass` is a subclass of `SOMObject`. `SOMObject` has no parents.

`SOMObject` introduces the methods to which all SOM objects respond. In particular, `SOMObject` introduces the `somDispatch` method, which provides a single, general dynamic dispatch mechanism for executing method calls on objects. Furthermore, a class can arrange its instance method table so that all method calls are routed through `somDispatch`. As a result, it is simple for SOM metaclass programmers to arrange for completely arbitrary processing in connection with method invocations on SOM objects.

As a subclass of `SOMObject`, `SOMClass` is an object, but it also introduces the methods to which all classes respond. For example, `SOMClass` introduces the `somNew` method, which creates instances of a class. Also, the methods responsible for creating and modifying instance method tables are introduced. All metaclasses in SOM are ultimately derived from `SOMClass`. (Similar arrangements of classes are also used in CLOS[7], ObjVlisp[2], Dylan[1], and Proteus[12].) The SOM API allows new abstractions to be created by programming metaclasses. In more general terms this has been called a *metaobject protocol*[7,8] or *computational reflection*[9]. The strength of this general approach is that new abstractions can be created after the object model is implemented. That is, the Before/After Metaclasses are not part of the SOM kernel, but they are part of a framework for programming metaclasses (see Reference 3 for more information) that is built with the SOM API. By providing a metaobject protocol, we

These unrelated `somInitMIClass` method procedures might then interfere with each other.

To solve this problem, SOM metaclass cooperation provides a framework for metaclass programmers to create method procedures that "cooperate" in executing inherited methods (instead of simply overriding these methods). This allows a cooperative metaclass to implement its desired semantics without interfering with the semantics implemented by other cooperative metaclasses with which it might be combined.

The following sections describe two important class frameworks enabled by cooperative metaclasses in the SOMObjects Toolkit (Version 2.1).

Before/After Metaclasses

A *before method* is a behavior that precedes the action of some program construct. An *after method* is a behavior that succeeds the action of some program construct. Before and after methods are familiar to users of CLOS[7, 11] where the granularity of application is the individual method. In the class-based object model SOM, the more natural granularity for before/after

were able to add a new abstraction to SOM.

Interfaces to SOM objects are described using IDL, an object interface definition language defined by the Common Object Request Broker Architecture (CORBA[10]) standard of the Object Management Group (OMG). SOM IDL is a CORBA-compliant version of IDL that allows SOM class descriptions to be supplied in addition to object interface definitions. (The interface to a class is described by the IDL alone. SOM IDL allows additional information about the implementation to be added.) The SOMObjects Toolkit has tools called *emitters* that translate SOM IDL into language-specific bindings for the corresponding classes of SOM objects. For example, to C and C++ programmers this means that emitters produce header files for both the users and the implementer of the class.

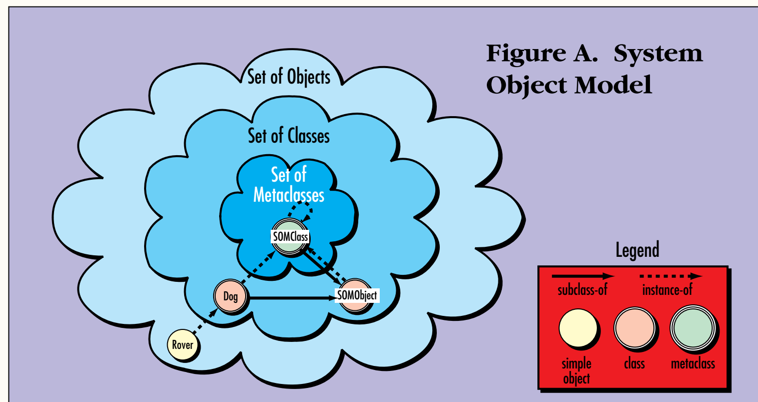


Figure A. System Object Model

The following example shows the basic structure of an IDL definition for an object interface named Dog.

```
interface Dog : SOMObject {
   // method and attribute declarations
   //here
   #ifdef  __SOMIDL__
   implementation
   {
       metaclass = SOMClass;
       // instance variable declarations
       // here
   };
   #endif
};
```

At the same time, it is a SOM IDL description of a class Dog that supports this interface. The #ifdef and #endif —part of the CORBA IDL language—are used to hide the SOM class implementation section from non-SOM IDL compilers.

In this example, the interface Dog inherits from the SOMObject interface, and at the same time, the class Dog is declared to be a subclass of SOMObject. CORBA and SOM support multiple inheritance; additional parents of Dog can be listed alongside SOMObject in a comma-separated list. The actual methods and instance variables of Dog are not relevant to the current discussion.

As illustrated here, the implementation section can explicitly indicate a metaclass to be associated with the class of objects that supports the interface being defined. This association is not necessarily direct. For reasons described in this article, the actual class of the class described by any given SOM IDL is, in general, a subclass of the indicated metaclass.

methods is the class, because there are many applications that fit this granularity[5]. Therefore, the SOMMBeforeAfter metaclass introduces two methods—sommBeforeMethod and sommAfterMethod—that its instances (classes) arrange to run respectively before and after each instance method. By default, these two methods do nothing. To define a specialized before/after behavior, divide SOMMBeforeAfter into subclasses and override the sommBeforeMethod and the sommAfterMethod with the desired behavior.

For example, consider the classes in Figure 3. The SOMMTraced metaclass overrides sommBeforeMethod with code that prints the method name and the calling parameters, and overrides sommAfterMethod with code that prints a message indicating that the method has finished execution and also the returned value. As a result, all methods supported by the class TracedDog (an instance of SOMMTraced and a subclass of Dog) have this before/after behavior. That is, for all methods invoked on the object Lassie, trace information is printed before and after each method invocation because Lassie is an instance of TracedDog. The IDL for the TracedDog class, shown at the left of the figure, is the only source

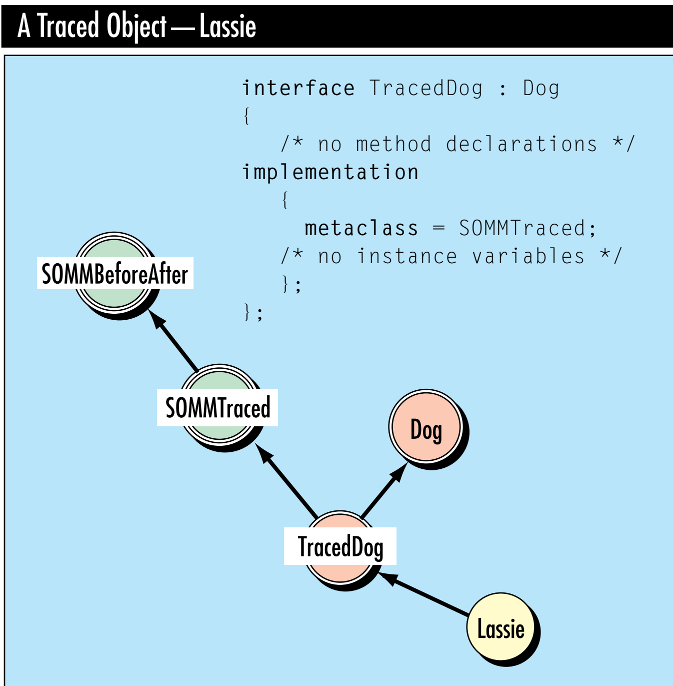


Figure 3. Example of a traced object—Lassie

```
somDispatch ( self, primaryMethod, ... )
   1. BeforeMethod( class(self), self,
                    primaryMethod, ... )
   2. retval := primaryMethod( self, ... )
   3. AfterMethod( class(self), self,
                   primaryMethod, retval, ... )
   4. return retval
```

Figure 4. Before/After dispatcher

code that needs to be written to implement `TracedDog`. The compiler of the SOMObjects Toolkit can generate from IDL all the necessary code to implement `TracedDog` (in either C or C++).

The `SOMMBeforeAfter` method overrides the method `somDispatch`, which is introduced by `SOMObject`. The new dispatcher looks like Figure 4.

In this case, `primaryMethod` is the method being invoked on a target object such as `Lassie`, for which before/after behavior is desired. In the pseudo-code used in this article, the first parameter to a method invocation is always the target object. The ellipses represent all the other actual parameters to the method. As noted earlier, the metaclass of the object `Lassie` supports `sommBeforeMethod`; that is, the class `TracedDog` responds to `sommBeforeMethod`. This is why, in the pseudo-code in Figure 4, `class(self)` is the target object for the `sommBeforeMethod` and `sommAfterMethod` method invocations.

For efficiency in the actual implementation, methods in SOM are usually invoked directly, and the `somDispatch` method is not generally called. To route all method invocations through `somDispatch`, the `SOMMBeforeAfter` metaclass places redispatch stubs in the method table to call `somDispatch`.*

A redispatch stub is a small piece of code that routes a method invocation through `somDispatch`. The `SOMMBeforeAfter` metaclass also arranges for the contents of the original method table to be saved so that `somDispatch` can invoke the primary method. In addition, the `SOMMBeforeAfter` metaclass ensures that `somDispatch` does not dispatch itself (which would cause a dispatch loop). The details of how this is done are very specific to the SOM API and beyond the scope of this article. More information on the SOM API can be found in Reference 6. In other words, the `SOMMBeforeAfter` metaclass needs some control over the method dispatch and overrides all method table entries with a piece of code (the redispatch stub) that ensures that control passes through `somDispatch`.

Distributed SOM

A second important framework that requires control of the method dispatch definition is DSOM, which implements remote method invocation as depicted in Figure 5. The specification is simple: given an object reference, you must be able to invoke methods on the remote object (identified by the object reference). In the usual convention, the invoking process is called the *Client*, while the process that holds the receiving object is called the *Server*.

*This is done with a method called `somOverrideMtab`, which is part of the SOM Application Programming Interface (API) that has been deprecated. *Deprecation* of a method means that the documentation of the method has been withdrawn from the Toolkit, and use of the method is being discouraged. Because of SOM's promise of release-to-release binary compatibility, a deprecated method's implementation is still available. Most of the deprecated methods (such as `somOverrideMtab`) are methods of `SOMClass`. The intent of these methods was to allow Toolkit users to build classes dynamically and to do metaclass programming. Classes can be dynamically built with `somBuildClass`. Metaclass programming is done by subclassing from the Toolkit's metaclass framework.

DSOM implements this according to the Object Management Group's Common Object Request Broker Architecture (CORBA) specification[10], which states that this is done through a level called the Object Request Broker (ORB), shown in Figure 6. The ORB must provide the following services: marshaling the actual parameters of the invocation, forwarding the invocation to the appropriate process, invoking the method at the server process, and finally returning any computed values.

The DSOM implementation of an ORB is based on implementing an object reference as a proxy object for the remotely located object. The way to invoke a method on a remote object is by invoking the method on the proxy, as shown in Figure 7. (The relationship between the CORBA specification and DSOM is more complex than we have shown. Object references can exist as an object or in a linear form; there is automatic conversion in the DSOM implementation. The *SOM User's Guide* section 6.8 contains a more thorough description of this relationship.)

Proxies are constructed as shown in Figure 8. As an object, a proxy has a class `ProxyFor_A` (in SOM 2.1, the name is actually `A__Proxy`, but we use `ProxyFor_A` for readability) that is created by joining the class of the remote object (`A`) with the DSOM framework class `SOMDClientProxy`. The metaclass of `SOMDClientProxy` is `SOMDMetaProxy`; because of inheritance of metaclass constraints in SOM, `SOMDMetaProxy` is also the metaclass of `ProxyFor_A`.

The key to implementing the DSOM proxy is redefining `somDispatch` in the class `SOMDClientProxy`. This redefinition interfaces with the ORB to forward a method invocation to the remote target object. Statically declared methods are invoked directly through the method table, but the method invocation is routed through `somDispatch` when its method table entry is replaced with a redispatch stub (a small piece of code that redirects the direct invocation through `somDispatch`). A dynamically constructed DSOM proxy class has all method table entries for methods inherited from the target class (`A` in Figure 8) replaced with redispatch stubs. In addition, some (but not all) methods that are inherited from `SOMDClientProxy` are also forwarded to the target object (by an explicit invocation of `somDispatch` within the method's implementation).

The `SOMDClientProxy` ensures that the `somDispatch` of all proxies properly interfaces with the client side of an ORB, marshals the

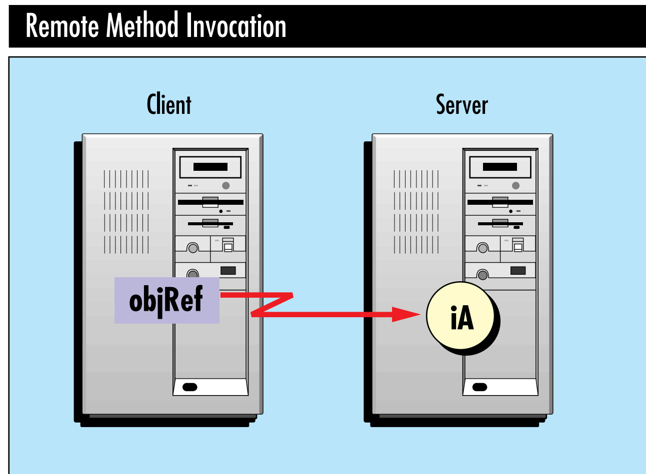


Figure 5. Remote method invocation

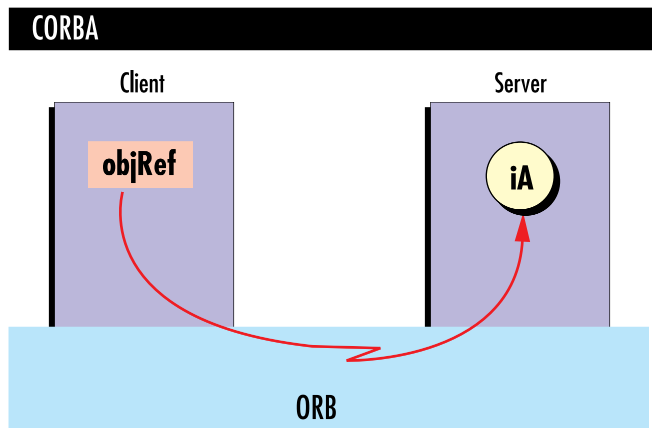


Figure 6. The Common Object Request Broker Architecture

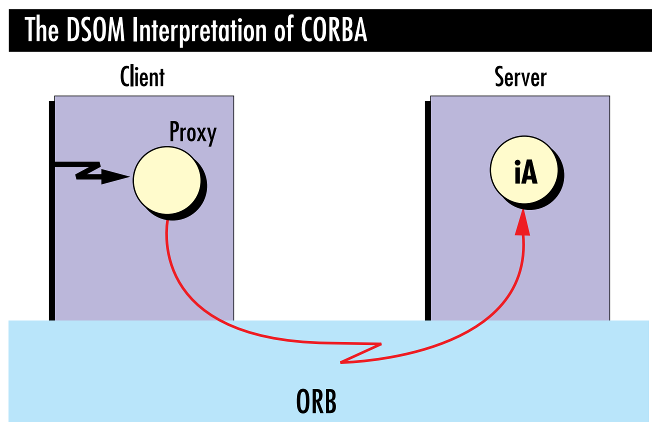


Figure 7. DSOM interpretation of CORBA

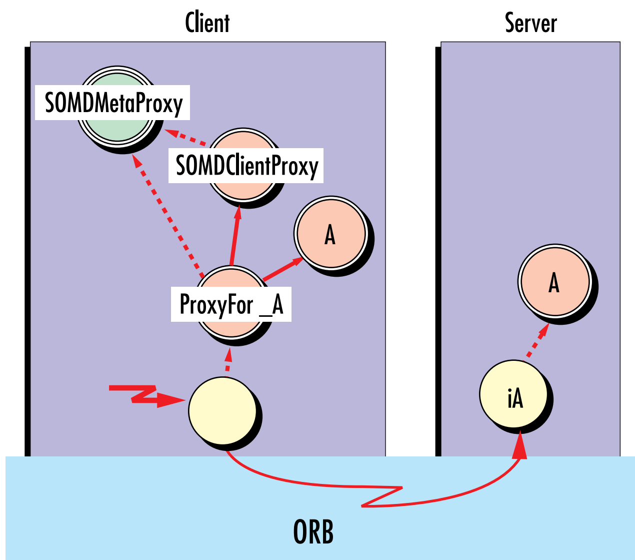


Figure 8. The DSOM class structure

```
somDispatch ( somSelf, primaryMethod, ... )
     1. Marshal the actual parameters.
     2. By using ORB services, send the
        invocation to the server.
     3. Bring back the values returned by the
        remote invocation.
     4. If the returned value is a CORBA
        object reference, convert it to a
        proxy.
```

Figure 9. SOMDClientProxy Dispatcher

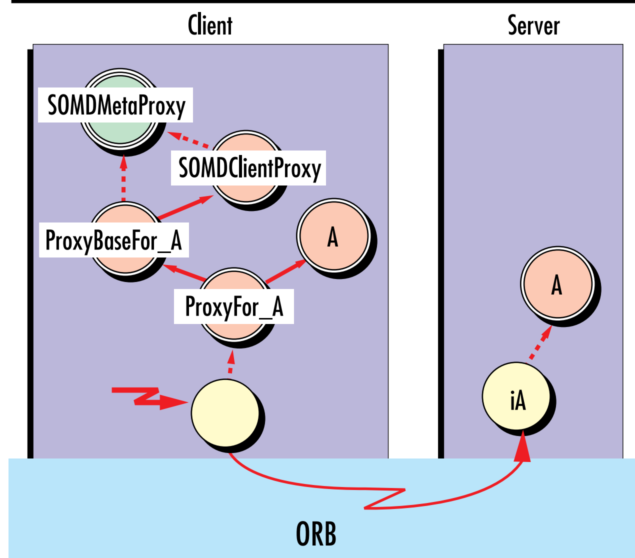


Figure 10. User-created proxy base class

actual parameters of the invocation, forwards the invocation to the appropriate process, and returns computed answers (see Figure 9).

`SOMDClientProxy` is the default base class for dynamically created proxies. In addition, `SOMDClientProxy` can be subclassed to create specialized base classes for proxies, as in the following example:

```
interface ProxyBaseFor_A :
SOMDClientProxy {
     ...
};
```

To use this proxy base class, it must be named in the IDL of the target class with the `baseproxyclass` SOM IDL modifier. Figure 10 depicts a proxy class with a specialized base proxy class. The advent of the Before/After Metaclasses in SOMObjects Toolkit 2.1 has generated good reasons to create specialized base proxies, as shown in the next section.

The Need for Cooperation

Each of the two frameworks—Before/After Metaclasses and DSOM—has a metaclass that requires the ability to define `somDispatch` to impart its property to ordinary objects. Without the ability to cooperate, composition of these two frameworks is not possible. The technology for cooperation[3] is based on reinterpreting the content of a method table entry as the sequence of methods, all of which might be invoked. The rationale for this is simple: from the metaclass programming viewpoint, metaclasses compete over resources such as a method table entry. By reinterpreting the method table entries as a sequence, you change the granularity of the resource conflict. Metaclasses can request the first, last, or arbitrary position in the sequence corresponding to an individual method table entry.

There are two areas of cooperation in which to compose DSOM proxies with Before/After Metaclasses. First, each framework must cooperate on the definition of the `somDispatch` method. In this case, the Before/After Metaclass requests the first method on the cooperation chain. `SOMDClientProxy` provides the last method on the cooperation chain.

Secondly, each framework must agree as to where to place redispatch stubs in the method table. The Before/After Metaclasses require that redispatch stubs be placed in all entries, while the DSOM proxy places redispatch stubs in most

(but not all) entries. In this case, DSOM's requirements take precedence. The proxy metaclass, `SOMDMetaProxy`, arranges this by requesting the first position in the cooperation chain for the method `somOverrideMtab` and ending the execution of the method without overriding all of the method table entries.

The result of this cooperation is that when Before/After Metaclasses compose with DSOM proxies, the before/after behavior occurs only for the method invocations that are forwarded from the proxy to the target object. This might not be what is desired by application programmers in all cases. That is, there are cases where before/after behavior is needed on a method, regardless of whether it is forwarded. However, Before/After Metaclasses cooperate on `somDispatch`, and DSOM uses `somDispatch` only to forward methods from the proxy to the target. A future version of the DSOM proxy will allow either to be easily programmed.

There is an additional question to address: Even if composition is possible, will it be used? That is, are there real examples that require the kind of cooperation presented here? This question is affirmatively answered by the following familiar example: transaction processing with logging—keeping an audit trail or journal for transactions.

A DSOM remote method invocation can capture the notion of a transaction (assuming the DSOM server is acting in single-thread mode). Transactions should be logged at the requesting sites.[4] Each proxy could be required to be individually crafted to log its activity. This necessitates much duplication of effort, especially considering the fact that logging is very much like tracing. Assume that we have a `Logging` metaclass that is a subclass of `SOMMBeforeAfter` and is much like the `SOMMTraced` metaclass.

Figure 11 shows how to arrive at the desired structure. First, create a general (and reusable) `LoggingProxy` with the following IDL:

```
interface LoggingProxy : SOMDClientProxy
{
      implementation {
         metaclass = Logging;
      };
};
```

This class is used as the base class for deriving a `LoggingProxy` class for any class. For class `A` in Figure 11, the IDL for the target class looks like the following.

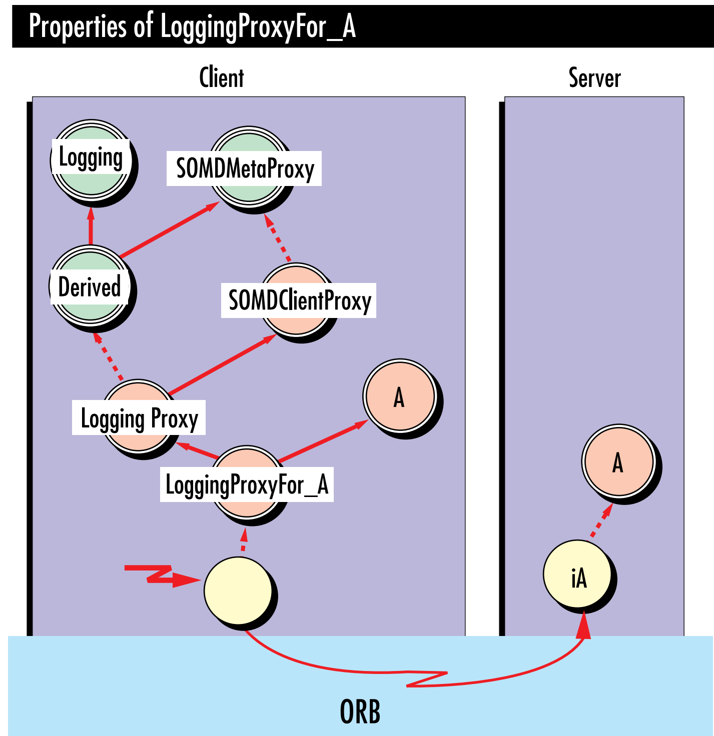


Figure 11. LoggingProxyFor_A has properties imparted by both Logging and SOMDMetaProxy

```
interface A {
   implementation {
      baseproxyclass = LoggingProxy
      ...
   };
};
```

The advantage of this arrangement (over the usual object-oriented abstractions) is better separation of concerns. There are three basic concerns that have been separated into independently programmed modules:

- Functionality of the transaction (located in the method being invoked)
- Remote invocation (inherited from `SOMDClientProxy`)
- Audit trail (located in the `Logging` metaclass)

Each of these concerns can be independently programmed only because `Logging` cooperates on `somDispatch` rather than overriding it. Without the ability to cooperate, DSOM would have to be modified to provide logging. Such a provision (probably implemented by subclassing `SOMDClientProxy` and overriding `somDispatch`) would not be reusable.

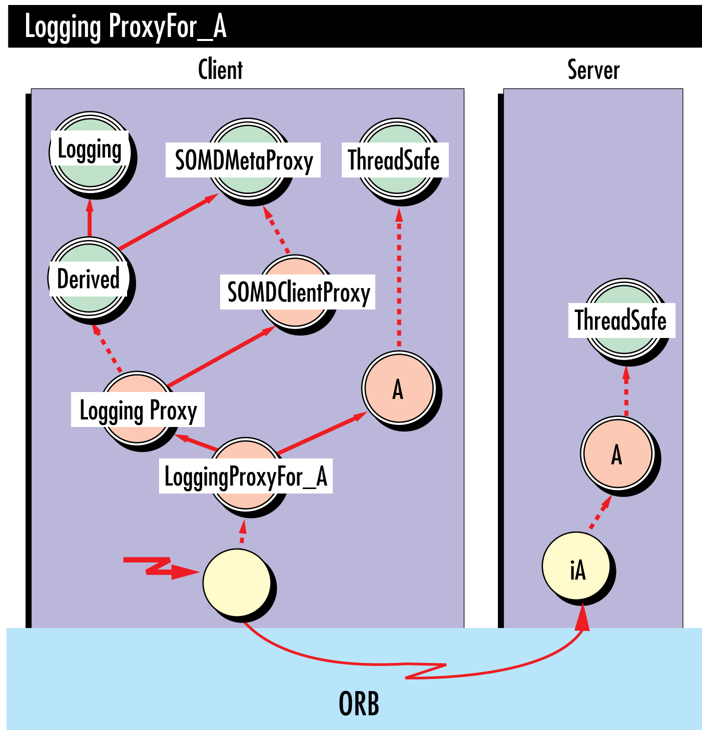


Figure 12. LoggingProxyFor_A is not thread safe

There is another aspect of cooperation. By design, DSOM does not provide concurrency control when the server operates in multithreaded mode. The reason is that there are many schemes for concurrency control[4], and it is not feasible for DSOM to support them all. Instead, DSOM customers can use the Before/After Metaclasses to provide concurrency control as shown in Figure 12. The metaclass `ThreadSafe` on the server can provide concurrency control, because `ThreadSafe` can acquire locks before a method is invoked and release locks afterward. Note that although `ThreadSafe` also appears on the client side (as a metaclass of `A`), `ThreadSafe` has no effect on the proxies to `A` instances. This is because DSOM does not allow inheritance of the metaclass constraint from the class being proxied.

Conclusion

A metaclass can embody a property (such as thread safety) independently of the objects that possess the property. The metaclass can impart such properties to the class at runtime without modification of the class, which leads to improved modularity. But the key to reuse is the ability to compose the metaclasses so that the properties are naturally composed. The standard object-oriented notion of method override leads to metaclasses that compete over the definition of methods and, thus, interfere with each other's operation (that of imparting properties). SOM uses cooperative metaclasses that add to the function of a method and avoid interference. This article shows that such cooperation is essential to a highly desirable solution to a very practical problem (that is, independently programming transaction logging and remote method invocation).*

Acknowledgments

The authors wish to thank Liane Acker for her comments on this article.

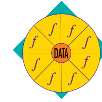
Presented at the Eighth IBM International Conference on Object Technology in San Francisco, June 13-16, 1995

References

1. Apple Computer. *Dylan: An Object-Oriented Dynamic Language* (1992).
2. Cointe, P. "Metaclasses Are First Class: the ObjVlisp Model." *OOPSLA '87 Conference Proceedings* (October 4-8, 1987) p. 156-165.
3. Danforth, S.H. and Forman, I.R. "Reflections on Metaclass Programming in SOM." *Proceedings of OOPSLA '94*. Portland, Oregon. (October 23-26, 1994).
4. Date, C.J. *Database: A Primer*. Addison-Wesley (1983).
5. Forman, I.R., Danforth, S.H., and Madduri, H.H. "Composition of Before/After Metaclasses in SOM." *Proceedings of OOPSLA '94*, Portland, Oregon. (October 23-26, 1994).
6. *IBM SOMobjects Developer Toolkit User's Guide*. Version 2.1 (October, 1994).
7. Kiczales, G., des Rivieres, J., and Bobrow, D. G. *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: The MIT Press. 1991.
8. Kiczales, G. and Paepcke, A. *Open Implementations and Metaobject Protocols*. Cambridge, Massachusetts: The MIT Press. 1994.

*Currently, in the SOMObjects Toolkit (Version 2.1), the interface to metaclass cooperation (as described in Reference 1) is not public. It is experimental and available on request. In the future, we (the authors) expect it to be available in the form of an operation on a class, much the same way as the Before/After Metaclass is an operation on a class.

9. Maes, P. “Concepts and Experiments in Computational Reflection.” *OOPSLA ’87 Conference Proceedings* (October 4-8, 1987) p. 147-155.
10. Object Management Group. *The Common Object Request Broker: Architecture and Specification*. OMG Document Number 91.12.1. 1991.
11. Paepcke, A. (ed.) *Object-Oriented Programming: The CLOS Perspective*. Cambridge, Massachusetts: The MIT Press. 1993.
12. Russinoff, D.M. “Proteus: A Frame-Based Nonmontonic Inference System.” *Object-Oriented Concepts, Databases, and Applications*. Kim, W. and Lochovsky, F.H. (ed.) New York: ACM Press. 1989. p. 127-150.



Ira R. Forman, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Forman is a member of the Object Technology Products group, where he specializes in object-oriented distributed systems and object composition. He has a PhD in Computer Science from the University of Maryland.

Scott H. Danforth, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Danforth is a member of the Object Technology Products group. He is currently developing language-neutral object technology for binary class libraries and system-level frameworks for OOP. He has a PhD in Computer Science from the University of North Carolina at Chapel Hill.

OpenDoc and Its Architecture

By Chris Nelson

OpenDoc and its related technologies represent an important standard for compound documents and component integration. IBM is both a member and a contributor of technology to the CI Labs consortium, holder of the OpenDoc technology. IBM is also producing the UNIX® reference implementation of OpenDoc. This article provides an overview of OpenDoc architecture and its related technologies: Open Scripting Architecture (OSA), Bento, ComponentGlue Technology, and System Object Model (SOM). It also discusses the programming model for developing software based on OpenDoc.

Compound document creation has undergone major changes during the last 25 years. With the advent of new technologies, a short history of compound documents can help put OpenDoc technology into perspective.

The 1970s—Documents by Hand

In the late 1970s and early 1980s, text, charting, and accounting tools were available to handle a wide variety of user needs. But each tool could work only with its own type of data and output that data in a limited fashion. If a document needed to include output from all of these tools, it was literally a cut-and-paste operation. If the document required any changes, it often meant starting from scratch, running each tool in succession, and hand-pasting the results together again (see Figure 1).

The 1980s—Tool-Level Integration

By the mid-1980s, platforms that enabled output from tools to be captured and placed in documents were readily available. One application or editor owned the entire document. Document layout was set aside for "pictures" of the embedded data, as shown in Figure 2.

Although this method was much easier than the manual cut-and-paste operations, this approach had several problems:

- The output data was static or "dead." If changes were required, the user had to return to the original tool, make the changes, then perform the electronic cut-and-paste operation.
- Generally, only pictures of the document could be traded across computers, since the

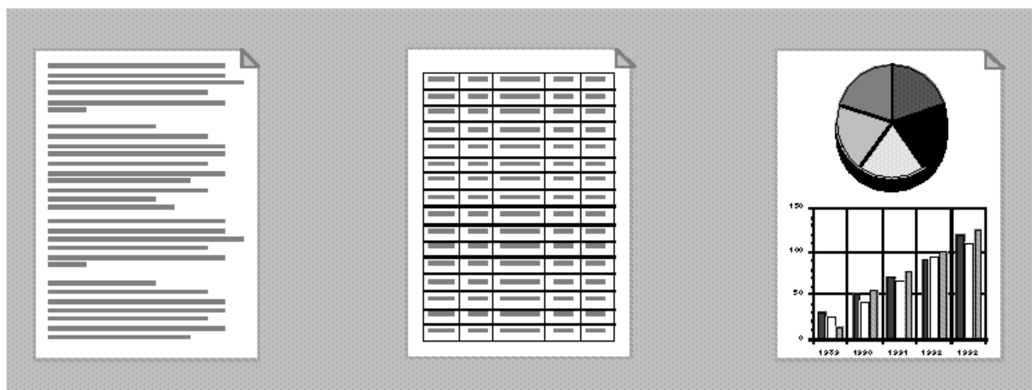


Figure 1. The 1970's—Documents were integrated by hand

Chris Nelson

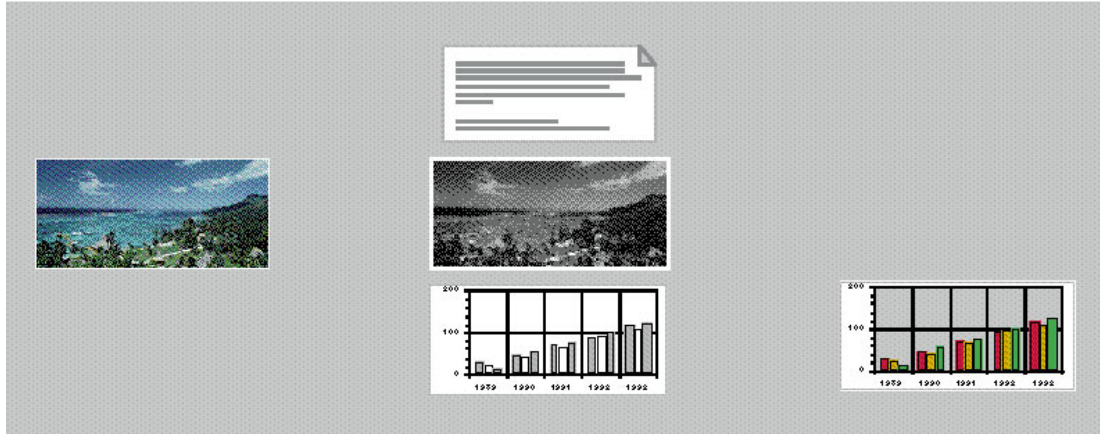


Figure 2. The 1980s—Tool level integration

document was tied to the local data and the tool.

- Items that could be placed into a document were limited to a small set of text and graphics, which were “pictures” of the data. There was no link back to the original data.

There was an additional problem with the tool-centric approach. As applications were able to support more and different kinds of data, the complexity of those applications increased significantly. Release cycles became stretched, requiring more programmers, increasing test time, limiting function between releases, and significantly increasing development costs. Applications also became complex and fragile.

The 1990s—Document-Centric Computing

With the advent of OpenDoc and other compound document architectures, such as Object Linking and Embedding (OLE) and the document framework in Taligent®, the model switched from tool-centric to document-centric.

In a document-centric computing model, users can focus on the documents they want to create rather than the output of a specific tool. Users can see a document that has several parts—all embedded and integrated into a single document, as shown in Figure 3. The type of parts that can now be included is dependent only on the availability of a part handler (editors and viewers). Some parts and features that we might first expect to see in these documents include the following:

- Formatted text
- 2-D and 3-D graphics
- Images

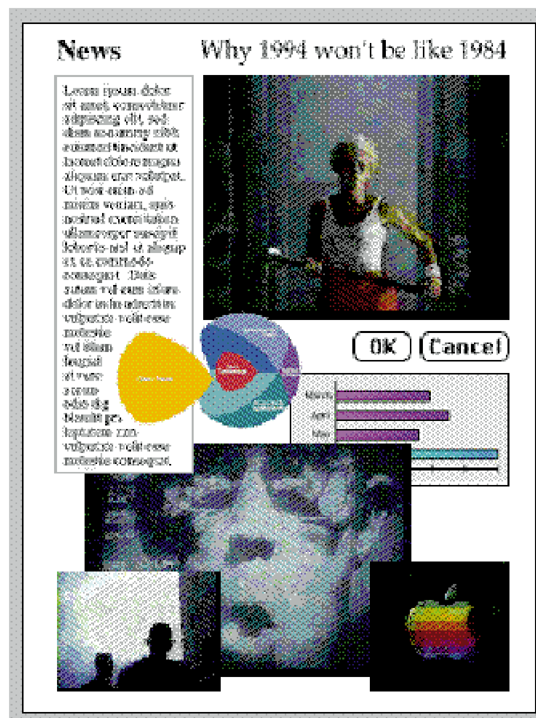


Figure 3. The 1990s—Document-centric computing

- Audio
- Video
- Structured data, such as appointments and calendars
- User interface control structures
- Inter- and intra-document links
- Document draft control

Part Handlers

The exchange of documents across platforms is greatly simplified, because the focus is on the various parts of the document—the data. The

only requirement is an available part handler (not a specific application) for the data contained in the document.

Applications are developed as small components called *part handlers*, which can be developed independently from other part handlers. In general, complexity is significantly decreased, allowing faster and less costly development cycles.

This approach also broadens the functionality that end users receive. Users can now acquire part handlers that integrate automatically, eliminating the dependence on one vendor for an integrated solution.

Compound Document Architectures

Currently, there are three available compound document architectures:

1. Microsoft's® Object Linking and Embedding (OLE)
2. CI Lab's OpenDoc and associated technology
3. Taligent's document framework

OpenDoc is a bridging strategy for application vendors who want to work in an OLE, OpenDoc, or Taligent environment with existing applications.

OpenDoc is a bridging strategy for application vendors who want to work in an OLE, OpenDoc, or Taligent environment with existing applications. With a quick port to OpenDoc, developers can extend the life of existing software products. They can also use OpenDoc to provide a migration path to Taligent-based products by introducing Taligent technology incrementally into their software base.

CI Labs

OpenDoc is an industry solution—not an IBM or an Apple® solution. The technology that makes up the integrated components of OpenDoc has been placed in an independent consortium: Component Integration Laboratories (CI Labs). CI Labs makes this technology, including the source code, available to all its members.

The initial technology base that will be contained in CI Labs consists of the five integrated OpenDoc components:

- **OpenDoc**: Compound documents
- **Bento**: Object container system
- **Open Scripting Architecture**: Policies, protocols, and software for scripting
- **ComponentGlue Technology**: The OpenDoc-OLE interoperability technology
- **System Object Model**: The single machine version of Common Object Request Broker Architecture (CORBA), which includes multi-process object invocation

CI Labs Mission

The form and function of CI Labs is similar to that of the X Consortium. Members of the consortium contribute toward the technical direction of the products. Members are allowed to modify and add value to source code reference implementations and to base products on this code. CI Labs will also play two other roles that have not been strongly part of the X Consortium. CI Labs will provide a certification service for developers. It will also provide a marketing role for its services and its technology base, along with other standards-based technology that becomes important to the goals of content integration, such as the technology from the Object Management Group® (OMG®).

Cross-Platform Coverage

Various companies will provide reference implementations for the CI Labs technology on the following platforms:

- **Macintosh**® (Apple)
- **OS/2**® (IBM)
- **Windows**™ (WordPerfect® from Novell®)
- **UNIX** (IBM)

Compound Document Architecture Concepts

Before looking at the specifics of the OpenDoc architecture, we will first consider the generic problem of designing a system to support embedding objects into a document and building component software. Most of this can be translated to both the technology of OLE and Taligent's document framework.

A fundamental concept of a compound document is that it can hold different types of data called *document parts*. Each document part is handled by an independent application or part handler. Each part handler understands its own intrinsic content, and even though other kinds of parts can be embedded into it, it need not know anything about the intrinsic content of that embedded part.

The integration and cooperation of the parts are handled by the architecture, policies, and protocols of the compound document system.

Other differences separate a document consisting of parts and a document created by a conventional application:

- **Content is not limited.** Users are not limited to the kind of content that can be put into a document. The only practical limitation is the availability of a part handler—an editor or viewer. As new part editors are acquired, they can be used immediately.
- **Content is built differently.** The user will rely more heavily on the clipboard and drag-and-drop mechanisms.
- **Part handlers can be changed.** Users can replace part handlers to fit their needs. This can be done independently of existing documents (which do not depend on a specific part editor).
- **There is a pervasive use of links for data transfer and navigation.** These links can be both internal to the document, such as hypertext type links, and external to the document for data transfer and navigation.

Compound Document Problems

Figure 4 uses a simple compound document to describe the problems faced in a compound document system and the interfaces for solving those problems. It takes the perspective of "If you were going to architect a compound document support library, what are the problems you would have to solve and the interfaces you would need to supply to the application developer?" Although there are several other problems associated with a compound document system that are not described here, these are the principal ones.

Data exchange—building the document. Many compound documents will be built up by the movement of data from one document (or the desktop) to another, either through cut-and-paste or drag-and-drop operations by the user. These interfaces determine how data is moved from one document to another. They must also define how data is put on and taken off the clipboard, along with passing type and attribute information so that the correct part handler can be accessed from the part handler database. Because the clipboard will be passing complex data objects, there must be an in-memory object container. The storage format for clipboard data will be a Bento container in the reference implementation of OpenDoc.

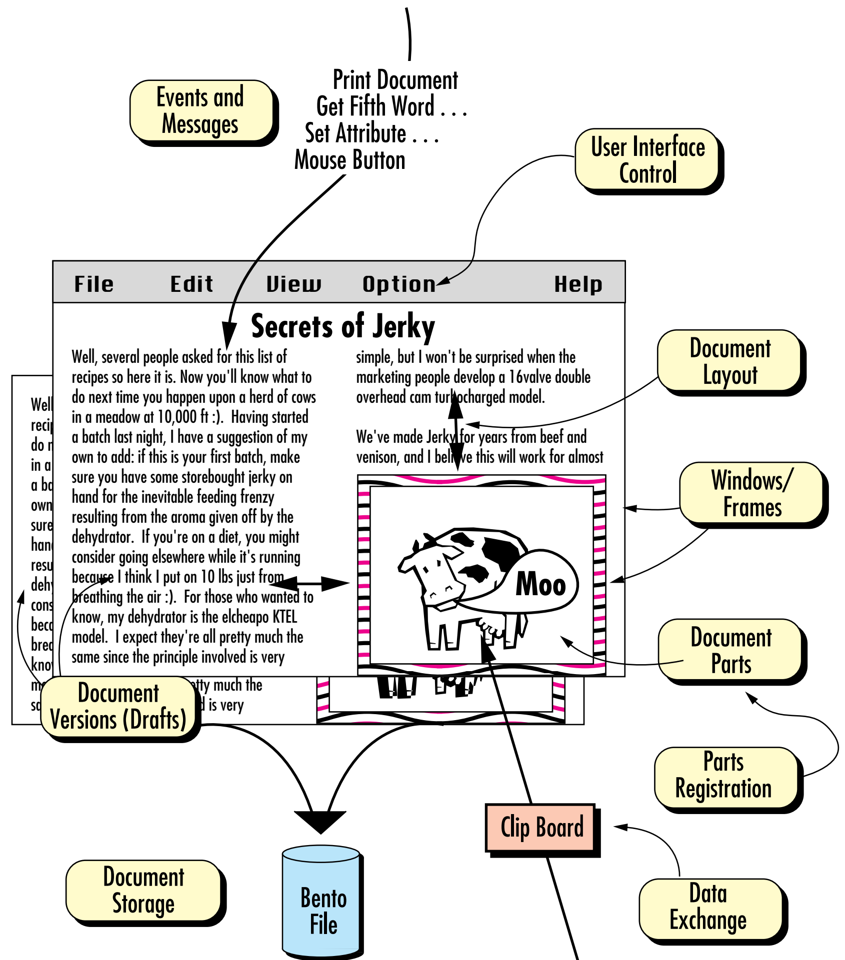


Figure 4. Generic compound document interfaces

Part registration—finding the right part handler for the data. When a container receives data of a particular type on the clipboard, it needs an interface to help it find the correct part handler for that type of data. Also, when a part handler is installed in the system, it needs an interface where it can register itself and the type of data (part types) that it supports. A part handler will often register itself claiming to handle several types of parts (such as different image formats), or there might be several part handlers that claim to handle the same part type.

For convenience, users can set up a preferences file to define which specific part handler is used for a particular type of part. An example of this would be a user who has two word processors on a workstation—MegaWrite and MuchoText. Although both have their own proprietary binary format, they also support the import of other formats, including the binary format of the other. The user could set up a part

OpenDoc Terminology

Term	Definition
Canvas	A surface to display frames in their facets. A canvas can be the display surface, a piece of paper, an off-screen display, and so on.
Container part	A part that can contain other parts. It has all the properties of a normal part, but it also tracks and responds to its embedded parts. The outermost or root container part is a somewhat special case. It takes up the entire document window and contains all other parts. An example of a root part would be a word-processing application. It has text as its native content, but can contain other parts to produce compound documents. These root parts often set the tone for document capabilities. A default root part contains only other parts and has no native content of its own. It simply provides a background.
Document draft	A version of a document in a single document structure. These drafts might be simple snapshots of the document as it is modified, or they might be drafts targeted for a specific audience (such as each draft localized to a language).
Event	There are two levels of events—low-level events such as mouse moves and key clicks, and high-level or semantic events, which are requests on an object to perform a function. The definition of the semantic events provides the basis for scripting.
Facet	The area on the canvas where the frame is displayed. Facets are not persistent—they are created and destroyed as frames are displayed and erased. For example, consider a multi-page document with many embedded parts. Parts that are not on the current displayed page do not have a facet. Parts that are displayed do have facets.
Frame	The area or border of a part. It encompasses the data in the part that is being viewed. It can be non-rectangular and displayed in one or more facets. In OpenDoc, parts can have multiple frames, with each frame representing a different view of the data. Frames are persistent and stored with the document.
In-place editing	The document-centric metaphor. When a user selects a part for editing or viewing, the edit can be done in place. A new window with the data need not be opened (although it can, if that is the policy of the part handler).
Link	A reference or pathway to an object. In compound documents, links are used in two ways: inter-document links such as hypertext links, and intra-document links that provide automatic updates of embedded information.
Part	The piece that gets embedded into the document. In OpenDoc, the embedded data is called the part, and the editor or viewer of that part is called the part handler. However, you will often hear the two combined with both the data and the editor/viewer referred to as a part. Conceptually, the part handler is an application program. Today, many applications, such as image editors, graphics editors, or audio players, will be modified to act also as OpenDoc part handlers.
Shell	The document shell and the root part are closely linked; however, it is useful to separate them functionally. The shell handles access to global information and the distribution of that information (by providing object references). It also receives low- and high-level events for the document.
Window	In OpenDoc, the outermost window of the document (the window for the root container). Other regions are called frames or menus, even though they might be implemented as windows in the native windowing system. This limitation in the use of "window" provides a cross-platform way of discussing OpenDoc regions.

handler preference with MegaWrite to be used for MegaWrite binary and MuchoText for MuchoText binary. This preference could be overridden if the user wanted to perform a conversion.

Documents—communicating between parts. If there are several embedded parts in a document, there should be an interface between parts and between the *containing part* and its embedded parts. Such an interface might provide the following.

- Access to internal part data
- Commands to the part, such as "save your data," "externalize yourself," and so on
- Requests to receive or give up control of some global resource, such as the menu bar or input focus

Geometry of part windows and frames—coordinating layout. With all of these parts sharing space on the screen and in the printed document, central coordination over layout is

necessary. This set of interfaces deals with the geometry of both the document window and the embedded parts. It contains information about the size, shape, position, clip extents, and transformations of the frames and windows, along with the means to set and manipulate them.

Document layout—determining the size. This interface negotiates space resources between the part handler and the container. It is used when the part needs to grow or shrink in size because of user interaction, such as editing the picture, or because the data changes after a link update of the data. The container, which sets the policy for document layout, has the final word. The result of this negotiation might be that the part is placed on a new page, the part is split into more than one piece, the document is rearranged to accommodate the new size, or even that the part is not allowed to grow.

User interface control—activating menus. This set of interfaces negotiates access to the user interface and other global resources. When a user selects a part with the mouse, the part handler needs to display its own user interface and menus to replace the previous ones. Additionally, there must be some policy set by the container for which menus are global to the entire document rather than just a selected part; for example, "print" under the "file" menu.

Events and messages—sending and receiving data. Several interfaces are needed for distributing, sending, receiving, and interpreting events and messages. These interfaces cover low-level events such as mouse moves or button pushes, middle-level events such as window manager events, along with high-level events or semantic messages. These interfaces are also the foundation for scripting, since scripts can be collections of semantic messages.

Documents—collaboration and versioning. These interfaces deal with multiple drafts or versions of a document. They handle the selection and creation of a draft, along with the management and reconciliation of multiple drafts. These interfaces are closely related to the storage interfaces.

Documents—storing and retrieving. These interfaces deal with the storage and retrieval of the document and its parts. They need to be abstracted from the underlying storage system so that a multiplicity of storage systems can be used. In the OpenDoc reference implementation, Bento containers will be used to build the storage system. It is likely that storage systems using OLE storage, Taligent storage, and object databases will be available.

OpenDoc Programmer Model

The programmer model for OpenDoc is a blending of function-based applications with classic object-oriented programming. OpenDoc was architected to allow easy conversion of existing applications to being OpenDoc part handlers. Externally, the application conforms to the OO interfaces of OpenDoc by using the `ODPart` object. Internally, the application continues to function as before. With Parts frameworks available, conversion can be straight-forward and rapid (see page 20 for some common OpenDoc terminology).

Class Library of Objects

In the previous section, the basic interfaces for a compound document architecture were generalized. In OpenDoc, all of the interfaces are realized as a class library of objects—System Object Model (SOM) or CORBA objects to be specific. Figure 5 shows the inheritance relationship of the OpenDoc class library.

The runtime relationship of the objects is more meaningful to the developer. Figure 6 shows the runtime relationship of the major OpenDoc objects. The shaded part of the figure represents the most important part for developers.

Part Developer Major Interfaces

The major programming task in creating an OpenDoc part editor is to subclass the class ODPart and override its methods.

The major programming task in creating an OpenDoc part editor is to subclass the class `ODPart` and override its methods. `ODPart` has 62 methods included in its interface. However, for simple parts and quick prototypes, only a dozen of these interfaces need to be changed. At a minimum, your editor needs to draw its part, retrieve its part's data, activate its part, and handle events sent to its part.

Since the editor will likely need a command or user interface to do this, it must provide menus for the menu bar as well as optional windows, dialogs, and palettes.

If the part editor will contain other parts, it must embed frames, create facets for visible frames, and store frames. The part editor can be extended to support asynchronous drawing, drag-and-drop and cut-and-paste, scripting, and links.

For converting an existing application, the simplest technique is to start with one of the provided sample parts and subclass it. This ensures correct default behavior for content-independent

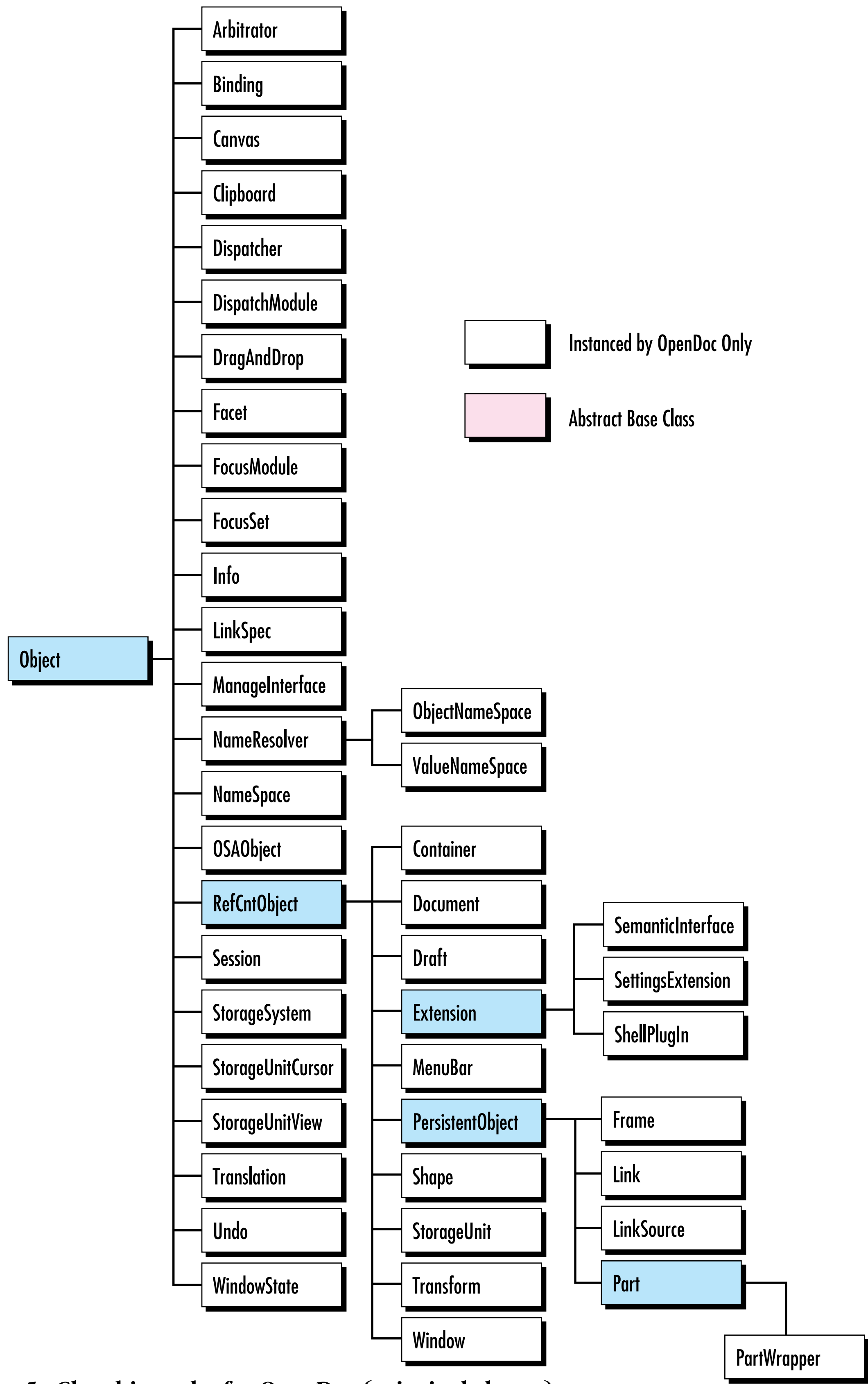


Figure 5. Class hierarchy for OpenDoc (principal classes)

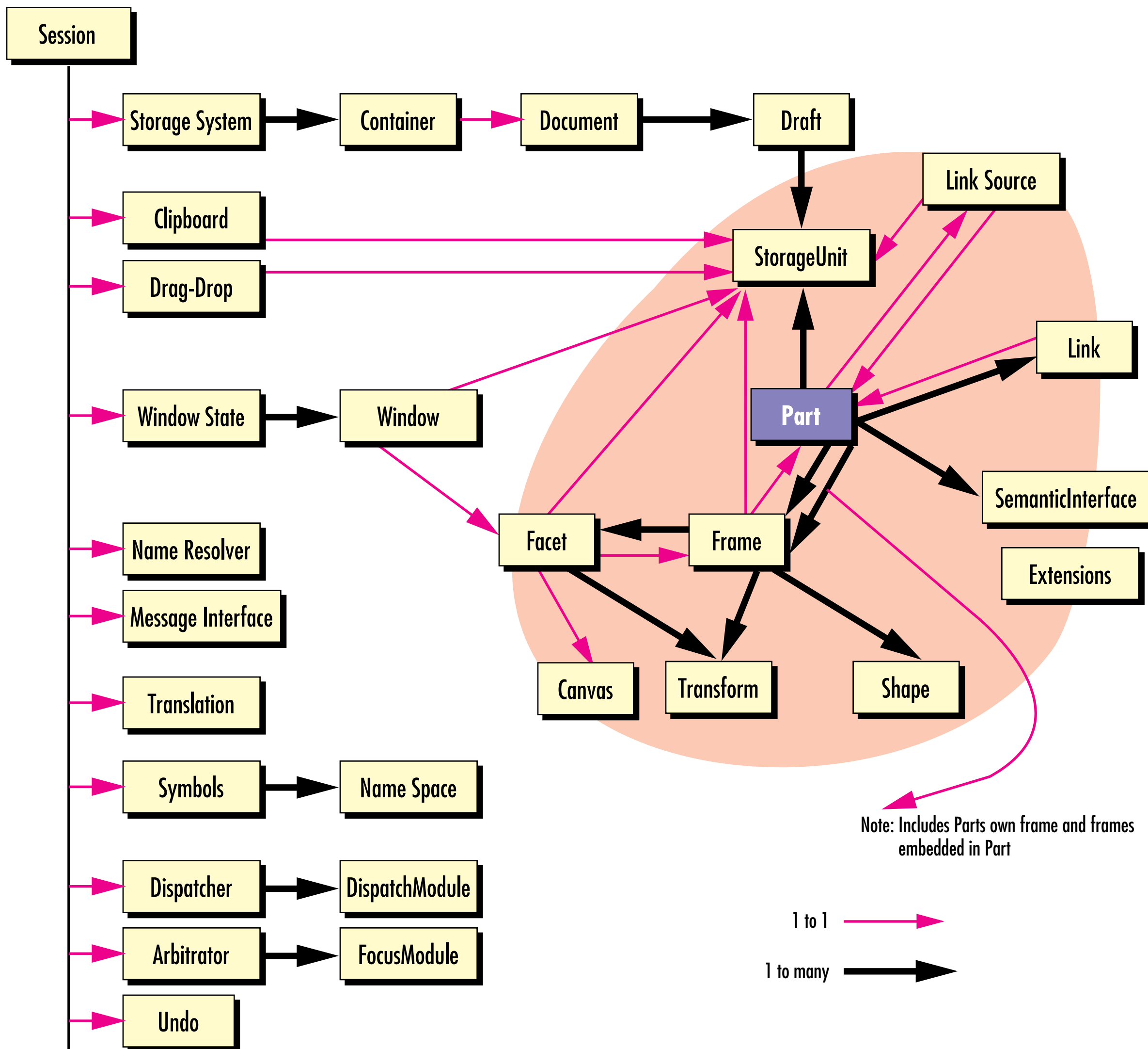


Figure 6. OpenDoc runtime object relationships

actions but allows for rapidly adding application function. In the past, OpenDoc developer labs (Parts Kitchens) have seen the majority of prototype applications converted in less than a week during lab times. Most of the platform vendors continue to offer Parts Kitchens.

Bento — An Object Container System

Bento is a simple piece of technology for containing objects (*Note*: The word container is used differently here than in OpenDoc). Think of it as a "Federal Express" envelope for objects. You can put anything into it, and there are established procedures for moving it around, looking at the inventory list, taking things out, and adding new things to the envelope—all independent of what is placed into it.

In OpenDoc, Bento has three primary roles:

1. A reference file storage system
2. An object container for items placed on the clipboard
3. Container of choice for transporting documents across platforms

You can think of Bento as a file system within a file; but rather than being a tree structure that reflects the embedded relationship, it is a flat structure. Figure 7 shows the structure of the Bento container with a series of objects. Each object has one or more properties associated with

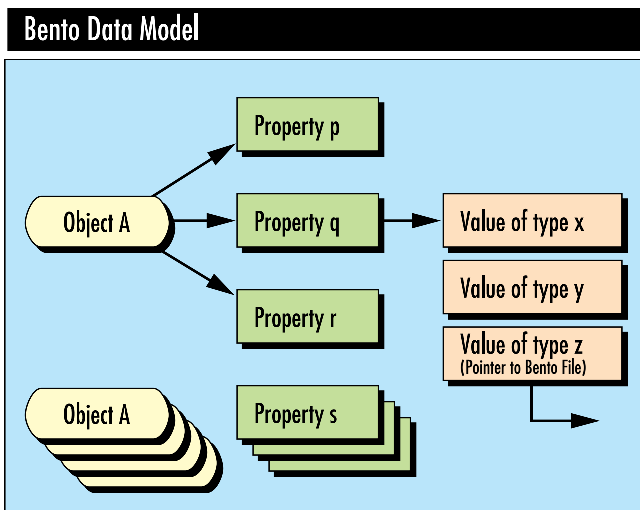


Figure 7. Bento data model

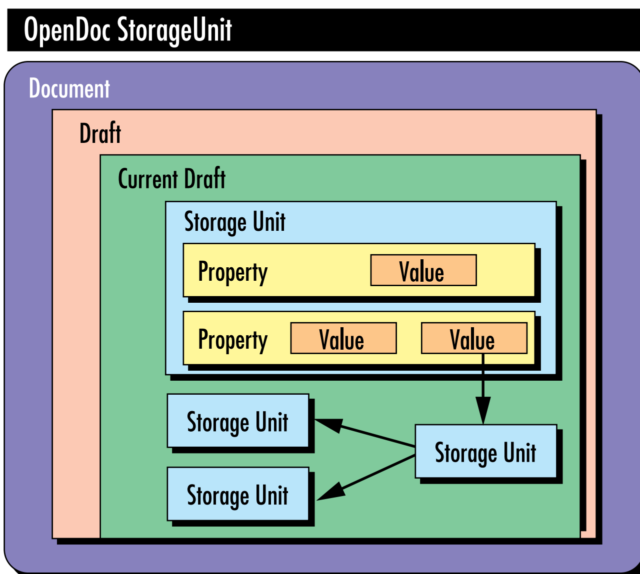


Figure 8. OpenDoc StorageUnit structure

the object, and each property has a set of values or data. In the example, property q might be formatted text stored in several different formats, such as RTF, SGML, and flat ASCII. Another valid value might be a reference to another Bento container.

Normally, the OpenDoc programmer does not deal directly with the Bento structure, but deals instead with the OpenDoc StorageUnit interface. The structure of the StorageUnit object is similar to that of Bento but can also be the front end for other types of storage such as an object-oriented database management system, or other object storage system such as the OLE Document File Format. Figure 8 shows the structure of the OpenDoc StorageUnit.

Open Scripting Architecture

The Open Scripting Architecture (OSA) provides an open architecture for scripting languages such as OREXX, AppleScript®, ScriptX®, or OLE Automation. The architecture is composed of two major components. The first component is the typing of script types, which allows the correct scripting engine to be included for script processing. The second component is the standardization of script semantic messages in the form of Open Events, which all OSA-compliant scripting languages will emit.

Scripts become part of the document, attached or embedded as needed. They can be the foundation for control structures such as front or back ends to operations, or they can be instructions that are embedded into a document that is mailed out.

Open Events

Open Events are based on standard registry of verbs and object classes. Apple Events provide the basis of this technology. Open Events are arranged in application suites, such as the following examples:

- Required
- Core
- Text
- Table
- Database
- Compound document
- New suites defined by user community

Each event is composed of three elements:

- **Events (the verbs):** Open, close, select, get, and put
- **Object classes or object specifiers:** Application, document, word, paragraph, and circle
- **Descriptor types:** Boolean, fixed, attribute greater than, 3rd, and bold

By combining these elements, single event messages of considerable complexity and richness can be composed. An example of a message for a text suite application might be the following:

```
Get all words that have the first character "w" and have the bold attribute in paragraph 5 to paragraph 12 of the document called "foo."
```

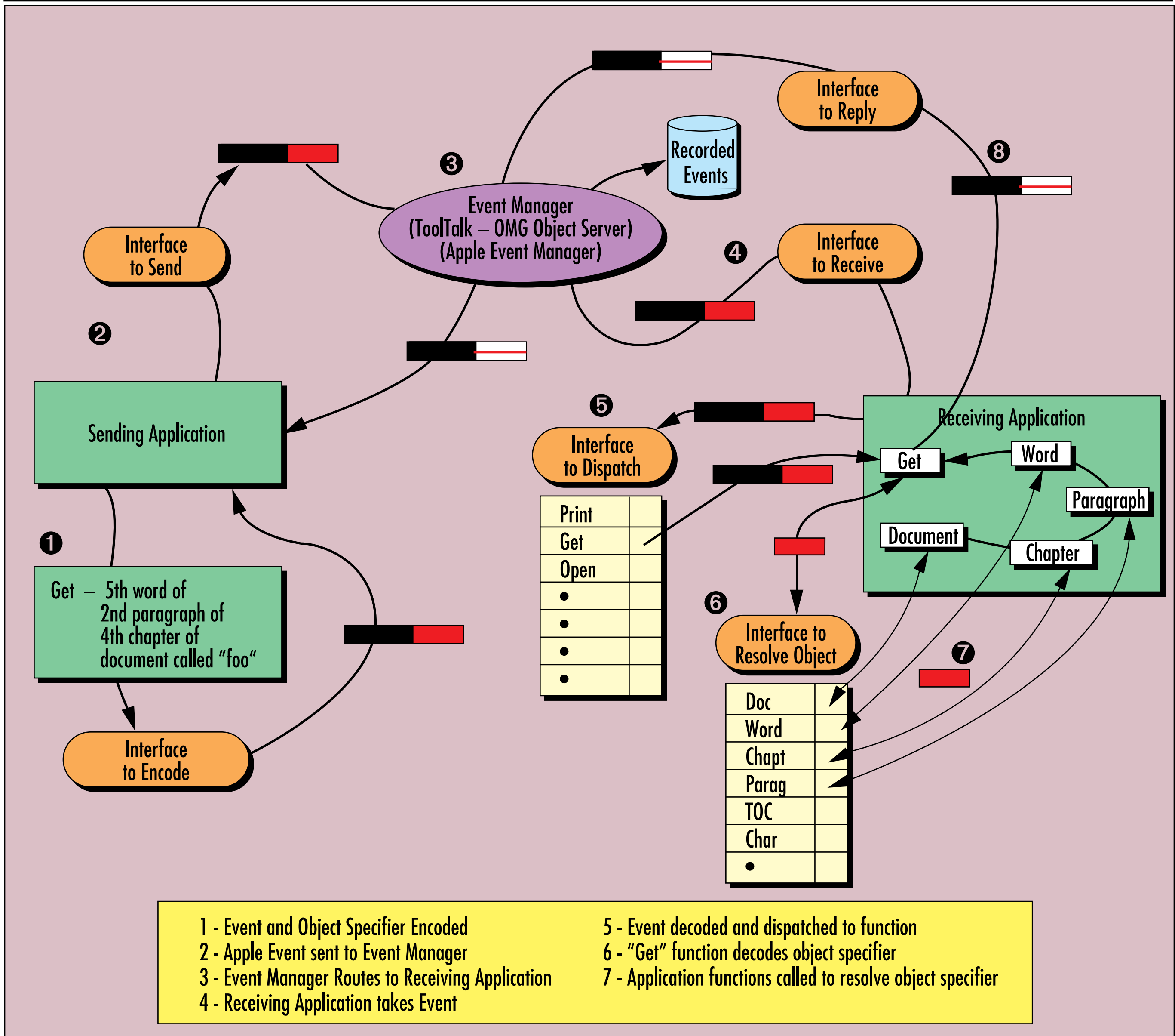


Figure 9. How Apple Events work

Parts that are OSA-scriptable register their capabilities by declaring the event suites that they support. Scripts can then be built for a particular set of suites, and those scripts will run on all parts that support those suites.

The set of event messages is open-ended. Parts developers can extend a suite's set of messages or define completely new suites. The extensions can remain proprietary or can be registered with CI Labs. The set of events for the suites beyond Required and Core is small. In most cases, the verbs from the Core suite are used with some of the semantics redefined. The object specifiers for each suite provide the real richness, and each suite has a hierarchy of objects on which it can act. The characteristics and attributes of each object specifier appear in the suite documentation.

Making a Part Scriptable

It is beyond the scope of this article to provide a detailed discussion on how to script an application or part, but some excellent documents and white papers on this subject are available via

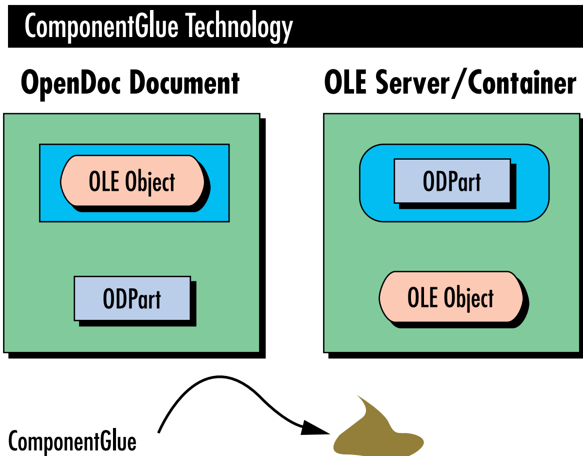


Figure 10. OpenDoc-OLE Interoperability—ComponentGlue technology

anonymous ftp directly from CI Labs (see the last section of this article for details).

Figure 9 shows the steps necessary for a script to be generated, transmitted, acted upon, and have information returned. There are three key points in this figure:

- OpenDoc has utility functions that help in building event records, delivering event records, and decomposing event records for action. For each of the event messages that a part supports, OpenDoc has registered a function that should be invoked—essentially a callback. This is also the mechanism by which it extends the set of event messages it understands. This is part of the semantic interface of OpenDoc.
- For each of the object specifiers and object descriptors that a part supports, OpenDoc has registered a set of functions that support them—essentially callbacks.
- Complex object specifiers are decomposed from outside-in or top-to-bottom. At each step, a reference to the acquired data is passed—first the file, then the chapter in the file, then the paragraph in the chapter in the file, and so on.

The example message `Get the 5th word of the second paragraph of the 4th chapter of the document called "foo"` perhaps looks somewhat contrived. But consider the following:

- The sending and the receiving applications might be the same application. The application has been factored into a UI component and a data engine component.
- The description of the example is a word chosen by a user and about to be put on the clipboard. Or perhaps it is a hypertext reference.
- The event message can be captured and stored (along with others) for future use.
- The stored events can then be fed into a scripting interface, generalized as needed, and turned into usable scripts.
- The developer's main task is to provide functions that support the event messages and object specifiers, and to factor their application into UI and data engine components.

System Object Model

SOM, IBM's implementation of the OMG CORBA standard, is an object-oriented technology for building, packaging, and manipulating binary class libraries. All interfaces for OpenDoc objects are written as CORBA Interface Definition Language (IDL) interfaces. SOM is one of the technology components that is part of the CI Labs OpenDoc technology, and all four of the current OpenDoc reference implementations use SOM as their Object Request Broker.

Features of SOM and CORBA

Some characteristics of SOM and CORBA are listed below.

Binary compatibility: SOM maintains binary compatibility between versions of class libraries. A binary class library such as OpenDoc shipped as a Dynamic Link Library (DLL) can be upgraded and changed without requiring a recompile or link of applications that use it.

Interface Definition Language (IDL) as defined by OMG: This is how CORBA-compliant objects are specified. All OpenDoc interfaces are defined in IDL, which has a similar look-and-feel compared to C++. IDL-specified interfaces are converted to a target language via compilers for the implementation of CORBA. SOM has IDL compilers for both C++ and C, with those for other languages such as Smalltalk® in process.

Language neutral: Because SOM is located in the middle of object transactions, objects written in one language can transparently make requests on objects written in another language.

Object Services: These services are needed to work effectively with objects. OMG is defining the interface and function of a set of services. Examples of Objects Services include life cycle—

create, delete, security, object naming, notification, and persistence. SOM has a framework for adding Object Services functions, so customers can select the capabilities they need and plug them in.

Object Management Group (OMG): A consortium set up to standardize an object architecture. The three main components of that architecture are an ORB, Object Services, and Common Object Facilities.

Object Request Broker (ORB): This entity brokers all requests one object makes on another. It is expected to transparently handle issues of location. It is also responsible for supporting object services such as creation/deletion and security. The OMG-defined ORB is called CORBA.

ComponentGlue Technology—OpenDoc and OLE

Technology being developed by Novell for CI Labs will allow OpenDoc and OLE parts to be seamlessly used in each others' documents by wrapping the parts with the appropriate object interfaces. The parts wrapper is called the ComponentGlue Technology or Component GT. This technology also supports the exposure of OSA-enabled part handlers as an OLE automation interface. Figure 10 depicts the ComponentGlue technology. A similar approach is being produced by IBM and Taligent for the Taligent document framework.

This ability to interoperate with both OLE and Taligent makes OpenDoc an ideal development platform for providing parts for all three architectures. It will allow developers to extend the life of existing software products with a quick port to OpenDoc. It also offers a migration path to providing Taligent-based products by incremental introduction of Taligent technology into the developers' software base.

Where to Get More Information

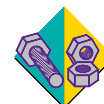
The best source of OpenDoc information is CI Labs. Documents, white papers, reference documents, and programmer guides that cover all of the technologies are available via anonymous ftp.

You can get additional information and subscribe to various CI Labs interest groups directly from the CI Labs mail server, cilabs.org. To access this server, do the following and you will receive a set of instructions via Internet mail.

1. Send a message to listsproc@cilabs.org
2. In the body of the note on a line by itself, put the word "help"

Information about developer CD-ROMs, Parts Kitchens, and beta programs is available from the following sources:

- **Macintosh:** opendoc@applelink.apple.com
- **Windows:** opendoc@wordperfect.com
- **OS/2:** 1-800-6DEVCON
- **UNIX (AIX®):** opendoc@austin.ibm.com



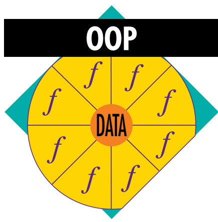
Chris Nelson, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: cnelson@austin.ibm.com. Mr. Nelson is an architect in IBM's AIX Desktop group. He has a BA in Chemistry from Ripon College in Ripon, Wisconsin and a BS in Electrical Engineering from the University of Texas at Austin.

Application Directories on the Web!

Looking for ways to advertise your software products? Or looking for software products that run on AIX, OS/2, or the PowerPC?

The answer is on the Web! The AIX, OS/2, and PowerPC application directories are now available on the Internet using URL: http://www.mfi.com/softwareguide.

To have your applications listed in the directories, complete and submit the product nomination form on the Internet. To have a nomination form sent to you, please call Miller-Freeman, Inc. at (415) 905-2721 or fax your request to (415) 905-2239.



Using SOM with C++

By Jennifer Hamilton, Robert Klarer, Mark Mendell, and Brian Thomson

This article introduces the IBM System Object Model (SOM) and describes how it can be used from C++. It also includes a description of the DirectToSOM (DTS) support provided by some C++ compilers.

A major goal of Object-Oriented Programming (OOP) is to write programs that can be more easily reused and extended than those written using conventional programming practices. The C++ language directly supports OOP techniques through language features such as classes and exception handling (which permit data encapsulation) and templates (a powerful mechanism for realizing generic types).

While there is considerable debate in the industry as to just how successful OOP languages have been with respect to achieving greater software reuse, such discussions typically apply to source code. What about binary reuse? Languages such as C++ not only fail to solve this issue, but actually make it considerably more difficult.

Jennifer Hamilton

Robert Klarer

Object Mapping in C++

The ANSI®/ISO® C++ standardization committee—whose mandate is to standardize the language itself and not the internal implementation of C++ compilers—explicitly chose not to specify a standard object mapping for C++. In fact, *The C++ Annotated Reference Manual* (ARM) suggests several different mapping schemes for handling virtual member function calls through virtual function tables. While most C++ compilers use a variant of this object mapping scheme, there is actually no requirement that the virtual function table scheme be used at all, although alternatives have not generally been used in commercial C++ compilers. Unless the object mapping is exact, the objects cannot be shared. So while the standard may ensure that C++ source code is relatively portable across implementations, it offers programmers no guarantee of the compatibility of binary objects that have been produced using different C++ compilers.

Even the C programming language enjoys an important advantage over C++ in its relative ability to support binary reuse. Most C language subroutine libraries can be exploited in client programs regardless of whether the same compiler, or even the same programming language, was used for both the library and the client code.

As a result, commercial C++ class library vendors cannot ship a single binary to their customers with the expectation that it will be useful regardless of which compiler their customers use. For example, it is common among vendors of class libraries for DOS to ship several binaries with each distribution of their product. Each such binary can be used with only one DOS compiler. Library vendors who want to serve the broadest market possible are thus forced to ship many precompiled versions of their wares to each customer.

Even within an implementation, it is difficult—if not impossible—to make changes to classes so that client code is not affected. For some cases, binary compatibility can be achieved by carefully managing class changes. But changing the size of a C++ class or migrating a member function up the class hierarchy will require recompilation of any clients of that class, including subclasses.

Clearly, we have a problem: there is no guaranteed way to share objects across different C++ implementations, let alone with other OOP languages such as Smalltalk. There is also limited ability to make common updates to classes without requiring recompilation of client code. Now imagine that you are responsible for maintaining a class library that is shared by 47 different applications. The fact that you have lots of shared

code is great, but if you change just one of those shared classes, you may be faced with having to recompile every client application.

System Object Model

System Object Model (SOM), like the conventional C++ object model, is a strategy for mapping instances of class types in memory. However, SOM was intended to support the reuse of binaries in ways that C++ was not. SOM has the advantage of Release-to-Release Binary Compatibility (RRBC), an important feature that breaks the tight dependency between the code that implements a class and the client code that uses it.

RRBC enables you to create and deploy a new version of a class without requiring that you recompile any unmodified application code. For example, you can add function or data members, or even inherit from additional base classes. In general, if you make a change to a SOM class that does not require a source code change in a client, then that client will not need to be recompiled. By packaging your class in a Dynamic Link Library (DLL), you can replace the old DLL with the new one, and all the applications that used it will continue to run without modification.

SOM also offers other features not available in native C++. The language-independent object model allows classes to be implemented in one language, instantiated in a second, and subclassed in a third. SOM supports extensive dynamic facilities, such as querying the properties of objects and classes, and using classes and methods whose names are not known until execution time. SOM also includes Distributed SOM (DSOM), allowing access to objects between processes or even across networks. It is fully compliant with the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) standards.

Experienced C++ programmers may find that the most surprising difference between SOM and the C++ object model is related to the role of object classes. In C++, a class is a syntactic entity that exists only at compile time; it has no representation outside of the source code that defines it. A SOM class, however, is also a SOM object, and always exists at runtime. Because SOM classes are runtime objects, they can provide a number of services to client objects at runtime. For example, each SOM class possesses a method named `somSupportsMethod` that, when invoked with any string, returns a value representing either true or false, depending on whether the class instances support a method whose name matches the parameter string. Other services supported by a SOM class include the following:

- Reporting its name to clients
- Identifying its base classes
- Indicating whether or not a given SOM object is one of its instances
- Reporting the size of its instances
- Reporting the number of methods that its instances support

All interactions with SOM are through standard procedure calls. Because public instance data is not directly supported, any language that supports external calls can use SOM. However, SOM is most heavily used today with C and C++, mainly because of the support for generating bindings for these languages, and the DirectToSOM support provided by some C++ compilers.

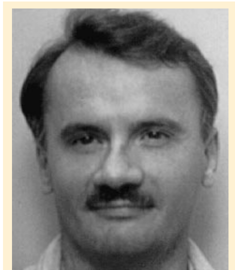
Interface Definition Language

C++ programmers can define SOM classes in one of two ways: either through the CORBA standard Interface Definition Language (IDL), or directly in C++ using a DirectToSOM C++ compiler. IDL is a language-neutral means for describing object interfaces, thereby enabling different compilers and even different programming languages to manipulate shared objects. The IDL definition describes the interface to, but not the implementation of, a SOM class.

The SOM Toolkit includes a compiler that generates bindings for a given target language from the IDL description. Bindings are language-specific macros and procedures that allow a programmer to interact with SOM through a simplified syntax that is more natural for the particular language. For example, the C++ bindings enable SOM objects to be manipulated through C++ pointers to objects, using any C++ compiler. DirectToSOM support, discussed in more detail later in this article, lets the programmer define and use SOM classes directly in C++ without needing IDL definitions.

As an example of C++ bindings, file `hello.idl` in Figure 1 shows the IDL definition for the class `Hello`, with the single method `sayHello`. SOM includes a compiler that will generate usage bindings, implementation bindings, and an implementation template for the class in the files `hello.xh`, `hello.xih`, and `hello.C` respectively.

Mark Mendell


Brian Thomson

hello.idl

```
#include <somobj.idl>

interface Hello : SOMObject
{
    void sayHello();
};
```

hello.C

```
#include "hello.xih"

SOM_Scope void  SOMLINK sayHello(Hello *somSelf,
     Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","sayHello");

}
```

Figure 1. Simple IDL class definition and implementation template

hello.C

```
#include <iostream.h>
#include "hello.xih"

SOM_Scope void  SOMLINK sayHello(Hello *somSelf,
   Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","sayHello");

    cout << "Hello world" << endl;
}
```

main.C

```
#include "hello.xh"

int main(int argc, char *argv[])
{
    Hello *obj;

    obj = new Hello;
    obj->sayHello(somGetGlobalEnvironment());
    delete obj;
    return(0);
}
```

Figure 2. Class implementation template and client code

The usage bindings define the public interface to a SOM class, and are included by clients to create and manipulate objects of that class. The implementation bindings, included by the class implementer, define the class and include private information that is not part of the class interface. Both the implementation and usage bindings files are regenerated completely by the SOM compiler when the class is modified, and should not be updated directly.

The implementation template (file `hello.C` in Figure 2 for the class `Hello`) contains procedure stubs for each method introduced or overridden by the class, and is updated incrementally by the SOM compiler when methods are added or a method signature is modified. The class implementer modifies the implementation template file to define the class behavior.

The first parameter to a method is the target SOM object, from which the address of the instance data can be retrieved by calling `classNameGetData`. The second parameter is an environment parameter that is required for CORBA compliance. In Figure 2 , we have updated the implementation file to make the method `sayHello` print the message "Hello world". The file `main.C` shows a client program that uses the class `Hello`. With the C++ bindings, SOM objects are declared and manipulated as pointers to the given class. You use the `new` operator to create instances of a class. The first time an object of any class is created, the SOM runtime environment will implicitly be initialized, and the first time an instance of a given class is created, the associated class object will be created, along with any parent class objects.

In this example, the second line of the function `main` will initialize the SOM runtime environment, create a class object `Hello`, and allocate storage for an instance of the class `Hello`, which will be assigned to the variable `obj`. The third line invokes the method `sayHello` on the object `obj`, while the final line deallocates the storage for the object. This program can be compiled and run with any C++ compiler. There is much more to the C++ bindings than we have shown here, but this should give you a feel for how SOM classes are defined, implemented, and used through the C++ bindings.

Release Order

The SOM interface does not support direct client access to instance data; all object manipulation is done through procedure calls, so the methods

form the only interface for a class. SOM supports RRBC by maintaining a list of all methods introduced by a class, called the *release order* for the class. Clients use the release order to access methods in the method table for the class. By keeping the order of methods in this list consistent, you can add new methods to the end of the list without forcing recompilation of client code. An IDL modifier is used to specify the release order for a class, listing each method in order by name, so new methods can be defined anywhere and simply added to the end of the list.

The usage bindings for C++ supply functions for each method that uses the release order to map the method invocation to the appropriate slot in the method table for the class.

DirectToSOM C++ Compilers

The capability to generate C++ bindings from an IDL description enables you to create and manipulate SOM objects with any C++ compiler, thereby gaining the advantages of the RRBC support provided by SOM. In addition, those objects can be shared across different C++ implementations or even with different languages such as Smalltalk. In using the C++ bindings, however, you are limited to a subset of the C++ language, which makes migration of existing C++ applications more difficult. You must also use two languages (IDL and C++) to define and manipulate objects.

DirectToSOM (DTS) C++ compilers support and enforce both the C++ and the SOM object models, enabling C++ programmers to take advantage of SOM through C++ language syntax and semantics so that the use of SOM is reasonably transparent and efficient. Instead of first describing SOM classes in IDL, the DTS C++ compiler translates C++ syntax to SOM. You can then have the compiler generate IDL from your C++ declaration, or you may find that you do not need to deal with IDL at all and can work exclusively in DTS C++. Finally, because you write C++ directly, you can use C++ features in your SOM classes that were not available before DTS. These features include templates, operators, constructors with parameters, default parameters, static members, public instance data, and more.

A C++ class is made into a DTS C++ class by inheriting from the class `SOMObject`, which is defined in the header file `<som.hh>`. You can do this explicitly, as shown in Figure 3, or implicitly through compiler switches or pragmas that insert `SOMObject` as a base class. The access

```
#include <som.hh>

 class Hello : SOMObject {
   public:
     void sayHello();
 };
```

Figure 3. A simple DTS class

specifiers—private, protected, and public—are supported for SOM classes and enforced following the C++ rules, as are constructors and destructors and most other C++ constructs.

Once you have defined a DTS class, what can you do with it? You can create SOM objects statically or dynamically, as simple objects, arrays, or embedded members of other classes (or anywhere else that the declaration of a C++ object is valid). Most C++ rules and syntax apply to DTS classes and objects, with some restrictions. Because the size of a SOM object is not known until runtime, compile-time constant expressions such as `sizeof` are treated as runtime constant expressions. Such operators can still be used with SOM objects, but not in contexts that require compile-time evaluation.

A major inhibitor to RRBC with C++ is the fact that so much information about an object is statically compiled into client code—in particular, the location of instance data and virtual function pointers. Data layout and method calling for a DTS C++ class are performed using the SOM Application Programming Interface (API) instead of the native C++ API. When you run a program defining a DTS C++ class, the compiler will create the corresponding SOM class object at runtime and use it to create and manipulate the object. As a result, unlike a standard C++ object, much of the information about a SOM object and its class—such as the instance size—is not determined until runtime, when the class object is created. This enables class evolution without forcing recompilation of client applications.

The SOM interface does not support direct client access to instance data. C++ instance data members in a DTS class are regrouped into contiguous chunks according to access, in the order of declaration within the class. This regrouping gives efficient access to data members from client code while enabling RRBC. The location of each chunk is determined at runtime. If the declaration order of public and protected data within a class is not changed, and new members are added after any preexisting members of the same

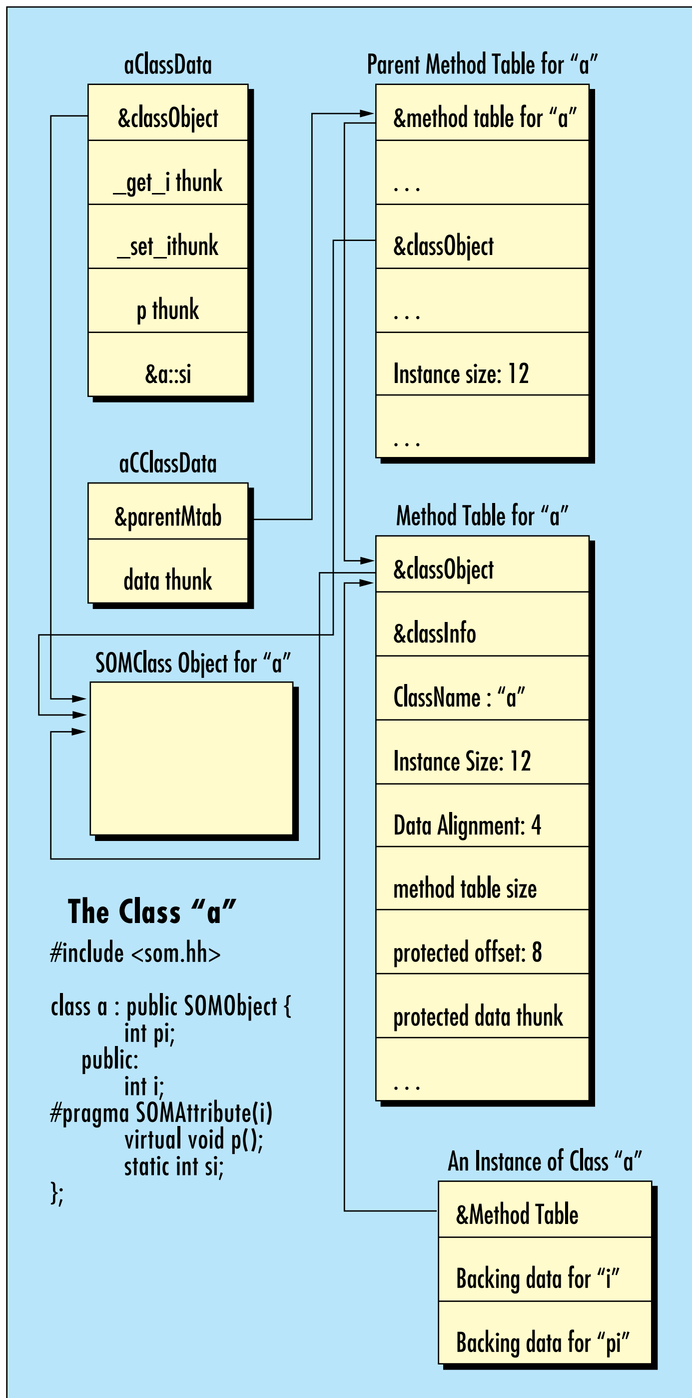


Figure 4. Runtime representation of a DTS C++ object

access, this scheme allows new data members to be added without requiring recompilation of any code outside the class (except for friends).

A DTS class also has a release order that, by default, will contain, in the order of declaration, all member functions and static data members introduced by the class, including those with private and protected access. In general, virtual functions that override virtual functions in a base class do not appear in this list, but will appear in the release order for the introducing class. Using the default, you must add any new member functions or static data members at the end of the class. Instead of relying on declaration order, you can use a pragma to specify the release order. In this case, you can add new release order elements anywhere in the class, but you must add their names to the end of the list.

For DTS classes, instance data and the release order list are created at runtime by SOM and used to manipulate SOM objects, rather than the statically defined compiler constructs used by standard C++. This provides for both RRBC and an implementation-independent object model. As long as the order of list elements does not change and new elements are added to the end of the list, you can add new data members and member functions—including migrating a member function up the class hierarchy—without forcing recompilation of client code.

DTS C++ Object Layout

The layout of a DTS C++ class differs from that of a native C++ class as follows:

- There is only one "method table" pointer in a class. This replaces the native C++ virtual table pointer. In a native C++ class, there may be many virtual table pointers in an object.
- There are no "virtual base pointers" in a SOM class.
- The data members for the SOM class have been reordered to place all the public data members first, then the protected data members, and finally the private data members.
- The order of the data instances for a derived class and that of its base classes is unknown at compile time and must be determined at runtime. This allows a base class to become larger in a new release, and still have the derived classes function correctly. The SOM runtime determines the class layout at startup time.

Figure 4 shows the definition and layout for the DTS C++ class `a`. Associated with every SOM class are two statically defined tables called `xClassData` and `xCClassData` for a given class `x`. A SOM class replaces the native C++ virtual table with the `ClassData` data structure. This table is initialized by the SOM runtime with a pointer to the SOM class object for the class followed by the

addresses of *thunks*, which are small code sequences that branch to the proper virtual function. To call a virtual function, the ClassData structure for the class that introduced the virtual function is indexed to get the address of a thunk. This thunk is then called, passing the this pointer and any parameters. It is then up to the thunk, which was created by the SOM runtime, to branch to the correct function.

To call the virtual function p for class a in Figure 4, for example, the function pointer found in aClassData[3] is called. The thunk will then call the appropriate method. The same code is used to call a non-virtual or static function. In this case, there is no thunk, but instead the address of the function is contained in the ClassData structure. Since the routine to be called never changes, there is no need for any generated code sequences. The address of static data members is also placed in the ClassData structure. The aClassData[4] structure in Figure 4 contains the address of static member si, thereby enabling other languages and implementations to access the data without knowing the external name of the variable.

The xCClassData structure contains a pointer to the parent method table for the class, followed by the address of a data thunk. Instance data for a class is accessed by calling the data thunk, which returns the location of the data introduced by the class. The use of the data thunk prevents any dependencies upon relative position or size of introduced data. The parent method table contents include pointers to the method table and class object for the class, along with the instance size. The method table itself contains class-specific information followed by the addresses of the actual methods for the class. Each class instance also contains a pointer to the method table, followed by the instance data for the object. The instance data for class a in Figure 4 contains the public data i before the private member pi, rather than the order of declaration within the class, because of the reordering of instance data by access.

Attributes

Note the #pragma SOMAttribute(i) directive for class a. A DSOM/CORBA attribute—logical data that is manipulated by get and set methods—is typically used when working with distributed objects for which direct data access is not possible. You can make a DTS C++ class data member into an attribute using the SOMAttribute pragma. The compiler will generate the methods _get_membername and _set_membername with the same access as the data member, and the actual backing data for the attribute will become private. By default, references to the members outside the class will not have access to the private backing data, so the compiler implicitly transforms member references into calls to the get and set methods. Within the member functions for the class, the private backing data can be accessed directly.

Differences between Native C++ and DTS C++ Classes

For a DTS C++ class, sizeof(SOM class) is a runtime value, not a compile-time value. That is because a SOM class may have a different size each time a program is run, if a shared library that defines a base class is changed to a new version. Within a given execution, the size is fixed. This means that int a[sizeof(B)]; is not allowed if B is a SOM class.

The offsetof(SOM class, member) value will give values that are somewhat unusual compared to native C++. The offset of a data member depends on the access mode; public members start at offset 0, but the protected and private members together also start at offset 0. Consider the following class:

```
struct a : SOMObject {
  private:
    int priv;          // offset 4
  protected:
    int prot;          // offset 0
  public:
    int pub;           // offset 0
};
```

This is because the protected and private members are allocated together as a chunk, separate from the public members. In addition, the offset is always relative to the class that introduced the member; offsetof(derived, base_element) is *always* equal to offsetof(base, base_element).

For CORBA compliance, SOM class member functions may have an extra hidden parameter (the Environment pointer) as a second parameter after the this parameter. Unfortunately, SOM Version 1 was developed before CORBA, and the environment pointer was not present. Older SOM classes do not have an environment pointer, but new ones do. This affects pointer-to-member functions. When calling through a pointer-to-member function, the compiler has to know whether or not to pass an environment pointer.

The compiler assumes that this is determined by the pointer-to-class function and prohibits mismatches. This is generally not a problem for existing C++ code, since all new classes will have the environment pointer.

A further difference lies in the area of inlining. Inlining member functions can inhibit RRBC, so the compiler generates inline functions "out of line." This preserves RRBC, but slows down execution.

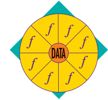
Conclusion

SOM provides a great step forward in achieving release-to-release binary compatibility and inter-language object sharing, with support for both remote and local objects. The C++ bindings generated by the SOM compiler from the IDL description enable SOM objects to be manipulated as C++ objects using any C++ compiler. Going one step further, DirectToSOM C++ compilers allow programmers to take advantage of the power of SOM through standard C++ language constructs and syntax, and to migrate existing C++ classes to SOM. Using SOM with C++, you can take advantage of the C++ support for OOP, using native C++ classes within your implementation and binary-compatible SOM classes for your interface.

SOM provides a great step forward in achieving release-to-release binary compatibility.

References

- Danforth, S. "A Bird's Eye View of SOM." *IBM OS/2 Developer* (Winter 1992).
- Danforth, S., Koenen P., and Tate, B. *Objects for OS/2*. New York, NY: Van Nostrand Reinhold, 1994.
- *C Set ++ Version 3.1 for AIX User's Guide*. (SC09-1968) 1994.
- *SOMObjects Base Toolkit User's Guide Version 2.0*. (SC23-2680) 1993.

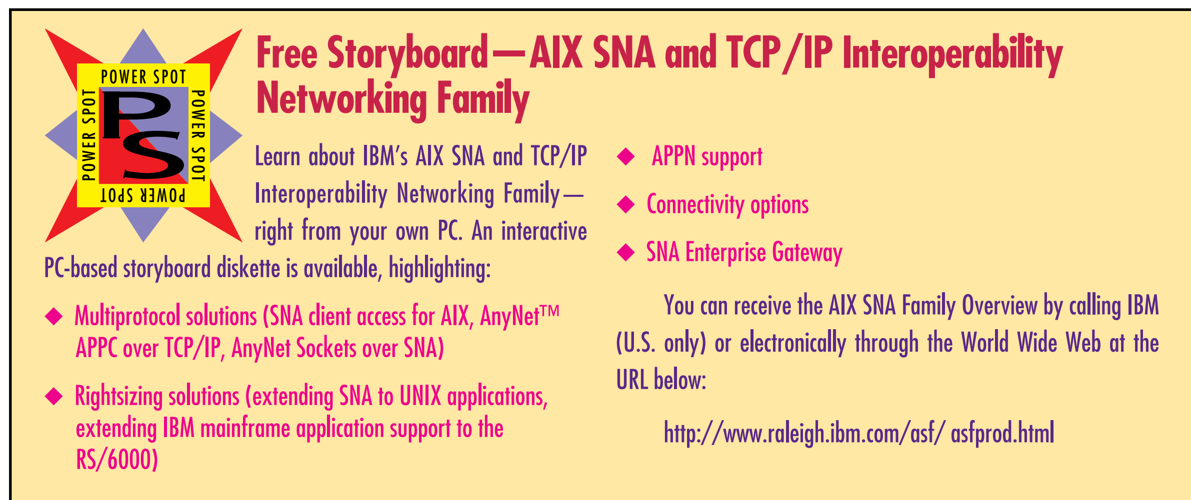


Jennifer Hamilton, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7. Ms. Hamilton has worked in compiler development since joining IBM in 1987, and currently develops the DirectToSOM support for IBM's C Set++ language products. She is the author of two books and numerous articles on programming language-related topics. She has a BSc in Computer Science from the University of Victoria, B.C.

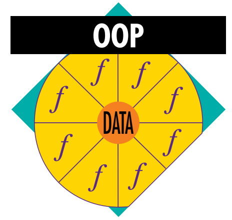
Robert Klarer, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7. Mr. Klarer has been involved in the development of several of IBM's C++ compilers. Recently, he has contributed to IBM's implementation of DirectToSOM C++. He has a BSc in Computer Science from McMaster University.

Mark Mendell, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7. Mr. Mendell has worked on compiler development since 1980. He helped develop compilers for Concurrent Euclid, Turing, and Turing Plus while working at the University of Toronto, and has been involved with C++ compiler development since joining IBM in 1991. He has a BSc in Electrical Engineering from Cornell University and a MSc in Computer Science from the University of Toronto.

Brian Thomson, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7. Mr. Thomson has worked in C++ compiler development since joining IBM in 1992. Previously he did research in operating systems, networks, and computer security, as well as languages. He is currently the DirectToSOM architect for IBM's C Set++ language products.



Major Features Adopted by the C++ Standard Committee



By Josee Lajoie

This article provides an overview of the new Runtime Type Information (RTTI) and namespace features and the C++ standard library, as well as a look at the current status of the C++ standard.

After five years of effort, the ANSI/ISO committee responsible for standardizing the C++ programming language is about to complete its task. Five years may seem like a very long time—but the C++ standard committee has accomplished an immense amount of work.

The committee decided to base the C++ Standard on two documents: *The C++ Annotated Reference Manual* (ARM)[1] and the *ISO Standard for the C Programming Language*[2]. It distributed tasks among the following groups:

Core Language Working Group: Led at first by Andrew Koenig and later by me, this group was responsible for clarifying the C++ language described in the ARM and ensuring that the semantics enabled efficient C++ implementations. The group made substantial improvements to many areas of the language, including the name lookup rules, the rules for type conversions and function overload resolution, the memory model, and the object model.

Extensions Working Group: Led by Bjarne Stroustrup, this group was responsible for addressing problems frequently encountered by the C++ programming community. This group only considered problems that could not be solved with existing language mechanisms or for which existing mechanisms were so cumbersome as to make C++ an unacceptable programming solution. This group's work resulted in four major extensions to the base language: templates, exceptions, Runtime Type Information (RTTI), and namespaces.

Library Working Group: Led by Mike Vilot, this group was responsible for providing class libraries that could be used as building blocks for more ambitious class libraries or for other C++ applications.

This article will provide an overview of the major features adopted by the committee. Because templates and exceptions were introduced by the ARM, only an overview of the new RTTI and namespace language features plus an introduction to the C++ standards library are discussed in this article.

The ANSI/ISO committee responsible for standardizing the C++ programming language is about to complete its task.

Runtime Type Information

The basic RTTI support provided in C++ can be broken down into two new facilities: the `dynamic_cast` operator, which allows type conversions that are checked at runtime, and the `typeid` operator, which returns the runtime type of an object.

The dynamic_cast Operator

The syntax for this new operator is `dynamic_cast<T*>(p)`, where `T` represents a type and `p` represents a pointer. The operator converts its operand `p` to the desired type `T*` only if `*p` is really a `T`—that is, only if the runtime type of `*p` is really a `T`. Otherwise, the value of the expression `dynamic_cast<T*>(p)` is 0.

Thus, the `dynamic_cast` operator performs two operations at once. It verifies that the requested cast is valid, and only if it is valid does it perform the cast. This verification renders the `dynamic_cast` much safer than the static (that is,

```
class employee {
public:
  virtual int salary();
};
class manager : public employee {
public:
  int salary();
  virtual int bonus();
};
class programmer : public employee {
public:
  int salary();
};
```

Figure 1. Class hierarchy of polymorphic classes

```
void payroll::calc (employee *pe)
{
   manager *pm =
     dynamic_cast<manager*>(pe);

   // employee salary calculation
   if (pm) {
      // use of manager::bonus()
   }
   else {
      // use of employee's member
         functions
   }
}
```

Figure 2. The dynamic_cast operator

```
class employee {
public:
  virtual int salary();
  virtual int bonus();
};
class manager : public employee {
public:
  int salary();
  int bonus();
};
class programmer : public employee {
public:
  int salary();
  int bonus();
};

void payroll::calc (employee *pe)
{
   // use of pe->salary() and pe->bonus()
}
```

Figure 3. The dynamic_cast replaced by a virtual function call

not runtime-checked) cast used in some situations to cast a base class pointer to a derived class pointer.

The `dynamic_cast` operator can be useful in situations where virtual functions cannot be used. It enables users to safely convert a base class pointer to a derived class pointer, ensuring that the interface of the derived class is only used when it is appropriate. Figure 1 shows a class hierarchy of polymorphic classes.

With a pointer to the base class `employee`, a user can now use the `dynamic_cast` operator to obtain a pointer to the derived class `manager` to use its member function `bonus`. With the `dynamic_cast`, the cast to the derived class pointer is only performed if the cast is valid—that is, if the pointer actually points to a `manager` object. Figure 2 shows what a user of the class hierarchy described above could do.

In the example in Figure 2, if at runtime `pe` actually points to a `manager` object, the `dynamic_cast` operation is successful, and `pm` is initialized to point to a `manager` object. Otherwise, the result of the `dynamic_cast` operation is 0 and `pm` is initialized with the value 0. By checking the value of `pm`, the function `payroll::calc` ensures that the appropriate action is taken for `manager` objects. That is, the `manager::bonus` member function calculates the salary of a `manager` object, while the more general case available calculates the salary of other types of `employee` objects.

Note that the virtual function mechanism already present in C++ is a better choice than RTTI to solve the problem presented. To use virtual functions, however, a user must have access to the source code for the class definitions. To extend the classes with additional virtual functions, a user must modify the base class `employee` and its derived classes to provide new virtual functions (in this particular case, a new `bonus` member function) and provide the necessary overriding definitions for the functions in the derived classes. See Figure 3.

The use of the virtual function mechanism as shown in Figure 3 is much more elegant than the use of the `dynamic_cast` in the earlier example. With the virtual function, the mechanics for the selection of the appropriate virtual function in the derived class are hidden from the users, and the code for the function `payroll::calc` is much cleaner. This mechanism should be preferred over RTTI whenever possible.

RTTI becomes necessary when the user extending the library does not have access to the

source code for the base class or some of the derived classes. For example, the base class could be part of a vendor library, and the source code for this library may not be available to library users. A user who wants to add functionality to the library classes through derivation will be unable to add new virtual member functions to the base classes and derived classes provided in the library. In this case, the user will need to take advantage of RTTI as shown earlier.

The typeid Operator

In addition to the `dynamic_cast` operator, a `typeid` operator is also provided as part of the new RTTI features. The syntax for this operator is as follows:

```
typeid(type_name)    or
typeid(expression)
```

The result of a `typeid` operation has the following type:

```
const type_info &
```

where `type_info` is a class provided in the C++ Standard library that describes the runtime type information provided by the implementation.

The operand of the `typeid` operator can be a type name, in which case `typeid` returns a reference to a `type_info` that represents the type name. The operand of `typeid` can be an expression, in which case `typeid` returns a reference to a `type_info` that represents the type denoted by the expression.

When the operand of `typeid` is a polymorphic type (a class type with virtual functions), the actual object is examined and its dynamic type is returned by the `typeid` operator. When the operand of `typeid` is not a polymorphic type, the `typeid` operator returns the operand's static type.

Suppose there is a need to use the previously described classes `employee` and `manager` with the `typeid` operator. Figure 4 shows how the operator could be used.

These examples show how the `typeid` operator can be used in expressions to compare the runtime type of C++ objects without directly manipulating `type_info` objects.

Let's examine the results of the comparisons provided in the previous example. Each expression compares the result of using the `typeid` operator with an expression operand with the result of using a `typeid` with a type name operand.

```
employee *pe = new manager;
   typeid(pe) == typeid(employee*)    // 1:   True
   typeid(pe) == typeid(manager*)     // 2:   False
   typeid(pe) == typeid(employee)     // 3:   False
   typeid(pe) == typeid(manager)      // 4:   False

   typeid(*pe) == typeid(manager)     // 5:   True
   typeid(*pe) == typeid(employee)    // 6:   False
```

Figure 4. Uses of the typeid operator

First, in `typeid(pe) == typeid(employee*)`, because `pe` is a pointer (not a polymorphic type), the type returned by the expression `typeid(pe)` represents `pe`'s static type, or pointer-to-employee. Note that the expression (`pe`) is examined, not the object type. Therefore, the conditions presented on lines 2, 3, and 4 are false.

This may seem surprising to users accustomed to writing the following:

```
// call to a virtual function
    pe->salary();
```

This results in calling the `salary` member function of the `manager` derived class. In that sense, `typeid(pe)` behaves differently from the virtual function call mechanism. Even if `pe` points to an object of polymorphic type, `typeid(pe)` represents `pe`'s static type, `employee*`.

When the expression '`*pe`' is used with `typeid`, the polymorphism of the object pointed to by `pe` is taken into account. In `typeid(*pe) == typeid(manager)`, because `*pe` is an expression that represents a polymorphic type, the type returned by `typeid` represents the dynamic type of the expression, or `manager`.

As mentioned above, `typeid` can be used with operands of non-polymorphic type. This implies that built-in type expressions as well as constants can be used as operands for the `typeid` operator:

```
int I;
... typeid(I) == typeid(int) ... // True
... typeid(8) == typeid(int) ... // True
```

To use the `typeid` operator, you must include the C++ standard header file, `<typeinfo.h>`. This header file defines the class `type_info`, which describes the runtime type information available and is used to define the type returned by the `typeid` operator. The exact definition of the class `type_info` is implementation defined, but certain features of this class are guaranteed by the

```
class type_info {
    // Implementation dependent representation
  private:
    type_info(const type_info&);                //1
    type_info& operator= (const type_info&);  //2
  public:
    virtual ~type.info();                       //3

    int operator==(const type_info&) const;    //4
    int operator!=(const type_info&) const;    //5

    const char * name() const;                  //6

    int before(const type_info&);               //7
};
```

Figure 5. The type_info class

Vendor A

```
vendor_a.h:
    class complex {  /* ... */  };
    complex sqrt (complex);
    double sqrt (double);
    int sqrt (int);
```

Vendor B

```
vendor_b.h:
    class complex {  /* ... */  };
    complex operator+ (complex c1, complex c2)
       { /* ... */ }
    complex sqrt (complex);
    const double pi = 3.1416;
```

Figure 6. Global names from different libraries

standard. The class is a polymorphic type, which supplies comparison operators and provides a member function that returns the name of the type represented.

Figure 5 represents the interface imposed by the standard for the class `type_info`.

Because the copy constructor and the assignment operator are declared private (lines //1 and //2), users cannot copy `type_info` objects. Because the destructor is declared to be virtual (line //3), the class `type_info` is a polymorphic class type. Lines //4 and //5 declare the overloaded comparison operators. These functions allow two `type_info` objects to be compared, and hence, allow the results obtained by using the `typeid` operator to be compared. The `name` function declared on line //6 returns the name of the type represented by the `type_info` object. C++ users have highly varied requirements for the run-time type information they want an implementation to provide. Some users want to minimize the space overhead resulting from the addition of RTTI to the language. For these reasons, the type name is the only information that the standard mandates that implementations provide.

Namespaces

What problem do namespaces solve? Like the C language, C++ has a single global namespace in which all the names declared in global declarations are entered. This is a difficult situation for library providers and their users. Global names in a library may collide with the global names in a user application, or with global names in other libraries provided by other vendors. For example, vendor A may provide a header file containing certain declarations and definitions, while vendor B may provide a similar header file, both shown in Figure 6.

Given these two header files, a user cannot easily write an application that uses both of these libraries.

Workarounds do exist that allow users to access these two libraries in one application. For example, Figure 7 shows that the problem can be solved when a prefix containing special characters is added to the names in the libraries.

This solution is not very elegant. Using prefixed names in an application can be quite cumbersome, especially if the prefixes are long. Namespaces are a better choice. They do not completely eliminate the global namespace pollution problem, but they significantly reduce the impact of the problem.

Namespace Definitions

A *namespace* is a mechanism for defining a scope. A namespace (user-defined scope) contains the traditional global C++ declarations. A namespace must have a unique global name; it is an error if any other global entity has the same name as a namespace.

A namespace can be used to encapsulate the declarations shown in the previous example:

```
vendor_a.h:
   namespace lib_a {
      class complex {  /* ... */  };
      complex sqrt (complex);
      double sqrt (double);
      int sqrt (int);
   }
```

The names declared within the namespace's braces belong to the `lib_a` namespace and do not collide with global names or names in any other namespaces.

In a namespace, you can declare or define anything you can declare or define at global scope: classes, typedefs, global variables (with their initialization), global functions (with their definition), and templates. The meaning of these declarations and definitions, once wrapped into the namespace, is the same as if the declarations and definitions appeared at global scope. The only difference is that the names declared are different. Namespace members must be referred to by the traditional notation for class members:

```
namespace_name::member_name
```

The members of the `lib_a` namespace can therefore be used as follows:

```
lib_a::complex func(lib_a::complex c) {
  return  ... lib_a::sqrt() ... ;
}
```

It can be cumbersome to always refer to a global name using the notation `namespace_name::member_name`. For this reason, the namespace features offer mechanisms to facilitate the use of namespace members.

Using Declarations

Using declarations enable users to make certain members of a namespace visible without requiring the names of these members to be qualified. For example, the code example above can be rewritten as follows:

```
using lib_a::complex;       //1

complex func(complex c) {
  using lib_a::sqrt;        //2
  return  ... sqrt() ... ;
}
```

Line //1 indicates that from now on, the name `complex` will refer to `lib_a::complex`. Line //2 indicates that, from now on in the body of function `func`, using the name `sqrt` will refer to `lib_a::sqrt`. Line //1 and line //2 are using declarations.

Using Directives

A *using directive* makes the names from the namespace available as if they had been declared at global scope where the namespace

```
vendor_a.h:
   class a_complex {  /* ... */  };
   a_complex a_sqrt (a_complex);
   double a_sqrt (double);
   int a_sqrt (int);


vendor_b.h:
   class b_complex {  /* ... */  };
   b_complex operator+ (b_complex c1, b_complex c2)
      { /* ... */ }
   b_complex b_sqrt (b_complex);
   const double b_pi = 3.1416;
```

Figure 7. Workaround to global name collisions

was defined. It does not declare local aliases for the namespace member names. For example, the following:

```
namespace A {
   int I, j;
}
```

looks like `int I, j;` to code for which `a: using namespace A;` is in scope. For example:

```
namespace A {
  int I, j, k;
}
void f() {
   using namespace A;
   I = 0;  //1: A::I
   int I;  //2: local declaration of I
}
```

With the using directive `using namespace A;` in function `f`, the reference to `I` on line //1 refers to `A::I`. A using directive is not a declaration; the using directive `using namespace A;` causes the name used on line //1 to appear as if it had been declared outside of `namespace A` in global scope, where `namespace A` was defined. Because of this, a local declaration of the variable `I` is allowed (line //2). It is not a redeclaration of the name `I` in function `f` since no variable `I` is declared in `f` by use of the using directive.

C++ Standard Library Components

An overview of the C++ standard library will show the following components:

Language support: Describes the types and functions needed to support the C++ language

```
class exception {
 public:
   exception(const exception e);
   virtual ~exception();
   virtual const char * what() const;
   // const char * describes nature of
exception thrown
 };
```

Figure 8. The exception class

features. This portion of the library includes the functions needed for dynamic memory management (`new`, `delete`) and exception processing (`unexpected`, `terminate`), as well as the class needed for RTTI (class `type_info`).

The technical work of the committee should be completed by the end of 1995 and the C++ Standard published by the end of 1996.

Predefined exceptions: Describes classes that support uniform error reporting from library components. These exceptions can be used in any kind of C++ application.

A common base class called `exception` is provided. This class defines a `what` function that can be used to identify the kind of error encountered, as shown in Figure 8.

Two classes are derived from this `exception` class. These two classes describe the two categories of exceptions that can be identified by library components. The class `logic_error` is used for exceptions due to the internal logic of the program (for example, a `logic_error` occurs when a string is accessed past the end of the string), while the class `runtime_error` is used to signal events beyond the scope of the program.

Strings library: Describes components that manipulate sequences of `characters`, where a character can be a `char`, a `wchar_t`, or any other C++ type that has a character-like behavior. This library defines the `basic_string` template, which gives a string the following properties:

- First element of the string is located at position 0
- Number of elements is fixed at construction time
- Elements are stored in contiguous storage

Localization library: Describes components that perform localization of strings. It provides internationalization support for character classification and collation, numeric representation, currency punctuation, and date and time formatting.

Containers library: Describes components that organize collections of objects. It provides template classes that describe the container types (list, vector, stack, queue, and so on).

Iterators library: Describes components that are used to iterate over containers, streams, and stream buffers. The access methods provided by this library vary in functionality and performance. The access methods provide functionality such as insert, reverse, and iterate.

Algorithms library: Describes components that perform algorithmic operations on containers and other sequences. Operations include the following:

- Non-mutating operations (find, count, search)
- Mutating operations (copy, swap, replace, fill, remove reverse, rotate)
- Sorting (binary search, sort)
- Merge
- Heap (push, pop)

Numerics library: Extends the C++ support for numeric processing. It provides a template class to define complex numbers, as well as other types and algorithms heavily used in numerical analysis applications. In general, this library is provided for users who want to use C++ for programming á la FORTRAN.

Iostream library: Describes the components needed for C++ input/output operations.

C library: Included in C++ primarily by reference. Some modifications were needed to ensure that the functions provided follow the rules required by C++ for static type safety. The description of the C library is sprinkled throughout the C++ Standard library sections; each C library component is associated with the C++ library components that provide related functionality.

The components offered by the C++ Standard library are extensive. With this variety, applications in a wide variety of domains can be written to be portable to all implementations supporting the C++ Standard library.

Schedule for the C++ Standard

When are we going to have a C++ standard? Since September 1994, the document that will eventually become the C++ standard has been passing through various phases required by ANSI and ISO before the document can be called a standard. The technical work of the committee should be completed by the end of 1995, and the C++ Standard published by the end of 1996.

In September 1994, the document was sent to ISO for the Committee Draft (CD) registration ballot. ISO distributed the document to the participating member countries, and the C++ technical experts of each country evaluated the document based on the following two criteria:

- Does the document describe a complete set of language features?
- Does the document describe sufficient support for class libraries?

The ballot was successful, meaning that the language and library features were frozen and the C++ Standard committee could no longer accept any new language extensions or class libraries.

In March 1995, the document (revised according to comments received during the CD registration ballot) was sent to ISO for the publication of the Committee Draft. At this point, the member countries sent out the document for public reviews. ANSI, the U.S. representative to ISO, began its public review period on May 26, 1995. The goal of this review is to decide whether the description of the language and library features is clear and unambiguous and whether the document can become an international standard. The ANSI public review period is scheduled to end on July 26, 1995. ISO must receive the position of each member country by the middle of August 1995, and then determine whether the document is technically good enough and clear enough to become an international standard.

After that happens, the committee must review the document and clarify some sections according to the public comments received. Four to eight months are scheduled for this work.

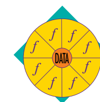
The committee should be ready to send the document to ISO for the last phase of revisions at the beginning of 1996. During this review, ISO will verify the format of the document to make sure it meets all the ISO requirements for an international standard document. The member countries will be asked to review the document one last time—for typographical errors, font errors, and so forth. The results of this last review will be known by the end of 1996. If the ballot is successful, the international standard for the C++ programming language will then be published.

Acknowledgment

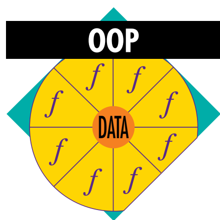
The information provided in this article has appeared in other forms in the "Standard C++" columns I write for the *C++ Report* magazine published by SIGS Publications. Stan Lippman, the editor, has kindly agreed to let me reuse the material for this article.

References

1. Ellis, M. A. and Stroustrup, B. *The Annotated C++ Reference Manual.* Reading, MA: Addison-Wesley, 1990.
2. International Standard for Information Systems. *Programming Language C.* ISO/IEC 9899:1990.



Josee Lajoie, IBM Canada Ltd., 1150 Eglinton Ave East, North York, Ontario, Canada, M3C 1H7. Internet: josee@vnet.ibm.com. Ms. Lajoie is a staff development analyst in the VisualAge C++ Compiler group. She is vice-chair of the ANSI/ISO C++ Sandard committee and the chair of the Core Language Working group for the committee. She writes the Standard C++ columns for the C++ Report magazine. Ms. Lajoie received a BEng in Electrical Engineering from L'Ecole Polytechnique of the University of Montreal.



Why Invest in Object-Oriented Programming?

By Dan Hattenberger

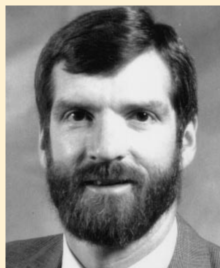
Object-Oriented Programming (OOP) seems to be the latest programming technology with promises of increased programming productivity, quality, and flexibility. However, software development managers charged with making money, and Information Technology (IT) managers charged with living within a budget need to know whether investments in new technologies will pay for themselves in either increased revenue or cost savings. This article examines OOP from a business perspective with some real-world examples.

Object-Oriented Programming (OOP) can provide considerable benefits to many programming companies or departments. Any software development organization that is plagued with numerous updates and changes, offers (or wants to offer) customized software solutions, or tries to reuse software in order to save money should consider object-oriented programming.

But before they do, that organization needs to address some real-world issues:

- **Initial investments:** Getting started
- **Ongoing costs:** New ways of doing business
- **Savings:** Productivity, quality, and reduced maintenance
- **Potential revenue:** Making money with OOP

Although many OOP articles and books address the first three issues, some discuss only the savings, and only a few address how to make money with OOP. This article addresses each of these topics, discusses some good ways to approach them, sets some realistic expectations for each based upon history and case studies, and outlines ways to get into OOP without breaking the bank.


Dan Hattenberger

Initial Investments

The three largest initial investments are training, software, and equipment. The first, and most important, is training.

Training

Programmers accustomed to traditional programming (where programs reside in one place and data resides in another—all development follows a design-code-test sequence) must learn to think in terms of objects and classes. Dr. Tom Love, a well-known OO consultant and author, recommends an education plan for OOP that should include the following elements[1]:

- OO Concepts (1 week)
- Language (2 weeks)
- Pilot Project (4 to 6 weeks)
- Initial Design (1 week)
- Initial System (20 to 40 weeks)

This plan reflects system development versus application development. However, business partners experienced in OO development also recommend the same elements with shorter times. OO Concepts should include basic OO concepts as well as analysis and design. New OO programmers must understand OO analysis and design. Without this paradigm shift, they revert to traditional methods in a new language and never

make the transition to "object-think."

Language training can be done many ways. Love suggests learning in two weeks. At IBM Rochester, however, where large portions of OS/400® are developed by using C++, programmers learn the language through a university course with one four-hour session each week plus assigned exercises over a span of eleven weeks. Programmers then take a twelve-week, four hours per-week design and analysis class. This new training method enacted after the first attempt at language training—a two-week course—failed because students could not absorb all the material at once, and they had forgotten much of it when they tried to use it.

The next phase, the pilot project, is extremely important. Programmers learning OOP should be assigned a meaty project that will challenge them without breaking the company in case it fails (as many projects do). An experienced object-oriented programmer (a mentor) should help the new programmer in this effort, providing suggestions for good OO designs that take advantage of the class libraries and that foster reusability. This will be described in more detail later.

Software and Tools

The second investment is in software and tools. OOP tool and framework development is in full swing today. By the middle of 1996, developers should have an excellent selection of development tools, a range of frameworks, and basic OOP support on all platforms. Ever-increasing competition and rising demand make it difficult, however, to predict the pricing of these tools and frameworks.

A software development company or department contemplating OOP needs at least one person in training today who can evaluate and select tools and frameworks. Many tools are available today, including evaluation versions. When evaluating tools, keep in mind that many are available in single-copy and team versions. Evaluate the team version—it is generally more powerful and is designed for multiple programmers.

Equipment

The last consideration is hardware for development. OOP tools run on PCs, and, depending upon the tools chosen, require lots of memory and power. Visual development tools, such as VisualAge™, are extremely powerful and allow the computer to do much of the work. Therefore, they require a powerful computer such as a 50 MHz (or higher) system with 20 MB (24 MB or 32 MB is better) of memory, and at least 540 MB of disk. Less robust tools cost less and require less computer horsepower. In general, the more powerful the tool, the more it costs to purchase and run.

Ongoing Costs

There are always costs involved in writing and maintaining applications. The large number of ready-to-use classes can take a significant amount of time to learn. To leverage reuse, programmers must design reusable parts, which takes even more time, particularly for inexperienced OO programmers. In fact, a company's first project often takes longer than a traditional implementation (sometimes as much as twice as long). The real cost savings happen in succeeding projects as the reusable libraries grow. Programmers will always create new objects and methods, so reuse is never 100%, and the amount differs for everyone. Over time, the ongoing costs for learning the classes and doing OOP design decline with more experience.

The three largest initial investments are training, software, and equipment.

Savings

Most OOP benefits are defined in terms of savings or increased productivity. Unfortunately, most of the benefits of any programming technology—3GL, 4GL, CASE, code generators, Graphical User Interface (GUI) tools—are defined this way. After a while, they all sound the same. What makes OOP different and what can it provide?

Code Reuse

The significant savings come from reuse. OOP "languages" are more than compilers; they include class libraries, editors, browsers, and debuggers. The class libraries contain thousands of pre-defined classes designed for reuse and tested for quality. These libraries contain only very generic classes—the trick is to use these small classes to create other, more robust classes that define objects specific to the customer world. As the number of "business-specific classes" grows, the application programmer reuses more code and writes less (OOP productivity comes from coding less, not faster), and the savings increase with each succeeding project.

Many software companies are trying to do this today but lack the appropriate tools. They enforce code reuse, scan existing programs, recall programs from old tapes, or just plain remember what they did last time. On the other hand, OO

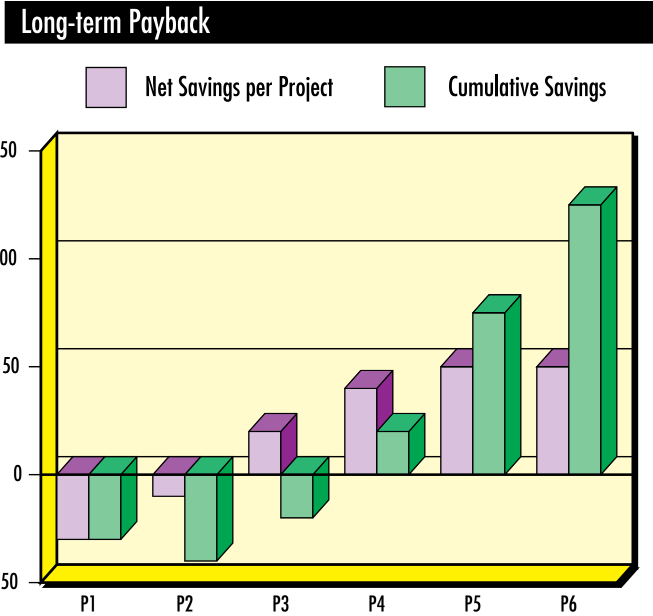


Figure 1. The long-term payback of OOP vs. traditional programming

Amount of Reuse	Development	Effort to Reuse	Development Effort	Savings
	(person-months)	(person-days)	(person-years)	(%)
0	100	0	8.3	0
10	90	10	7.5	9
30	70	30	6.0	28
50	20	80	2.0	76
80	20	80	2.0	76

Figure 2. Estimated effort to create a system of 100 classes

programmers use a browser to search for existing classes that fit the current need.

Figure 1 compares the potential savings of OOP versus traditional programming. Although the first few projects show negative savings as the reusable parts are developed, savings begin to accumulate after that.

Frameworks are large, reusable parts that come in several varieties and contain the logic, flow, and most of the key business objects that make up the backbones of applications. Programmers customize the framework by providing additional objects, modifying the objects supplied, and/or rearranging the parts and flow. By starting with larger parts created by experienced OOP programmers and designers, development shops can move to object-oriented applications with less effort and less risk. The key to success is to shop around for the framework that most closely matches the application requirements in function, platform portability, and ease of adaptation.

Productivity Gain

How much productivity gain can be expected? In one example[2], a new realty company in California wanted a state-of-the-art application suite that included a listing database, market analysis, client management, personal marketing, agent messages/status, and an interface to the Multiple Listing Service. An experienced OO programming team installed all the applications in 45 days and 1,000 person-hours. They succeeded because they made extensive use of an existing class library that they knew very well.

In a carefully controlled study[3], Electronic Data Systems (EDS) Corporation compared traditional programming to OOP. They replicated an existing manufacturing system that was originally implemented in PL/I and a relational database. EDS formed a team of Smalltalk programmers comparable in experience to the PL/I team. This team started with the same set of specifications, wrote the system, and tested it with the original test suite. The Smalltalk team completed the project in 3.5 months (versus 19 months for the PL/I team), with 10.4 person-months (versus 152), and in 22,000 lines of Smalltalk (versus 265,000 lines of PL/I). The EDS example shows a key characteristic of OOP productivity: the reduction in effort (14:1) closely resembles the reduction in code (12:1). This example is a widely quoted one, but might not be typical (14:1 is unusually high).

A better example is a company that develops command and control systems for the Swedish defense services[3]. They were faced with providing similar, but different, command, control, and communications applications across different ships. Recognizing early that they needed modularity and flexibility, they used an object-oriented design to model the shipboard system while ensuring development of reusable software components for future use.

Because the applications were designed for reusability, succeeding implementations were easier and cheaper. In fact, after completing two installations, this company estimated a 6:1 productivity improvement in succeeding installations, due mainly to 70% code reuse.

Managers contemplating OOP should consider the amount of potential reuse in their application development. The chart in Figure 2 shows some estimates of cost reductions for different levels of code reuse[1]. The savings ratios are not as high as the reuse ratios, because there are other costs associated with design, integration of parts, and testing of the whole application.

About 20 years ago, the IT manager of a large corporation stated, "Users never know what they want—they only know what they don't like when they see it." Customers seldom do a good job of defining their needs at first. Analysts define the requirements and provide specifications, designers design applications, programmers code the application, and then, often several months later, the customer gets a first look at the product. It was probably shortly after one of these "first look" sessions when the IT manager made her statement.

OOP design is an iterative process of designing, prototyping, examining or testing, then repeating those steps as necessary. Reusable classes, particularly the GUI classes, make it easy to prototype the front end of an application and review it with the customer at the beginning of the development cycle. As more code is added to the prototype, other reviews are possible (highly recommended). These iterative prototype/review cycles help eliminate costly rewrites due to changing customer desires or changes in the customer business.

Maintenance

Poor quality is expensive in terms of rework and customer sales. A survey taken several years ago revealed that customers often made a product choice based upon talking to someone who had already purchased the product. Even though there are few quantitative measurements of OOP quality versus traditional program quality, new OO programmers often notice an increase in quality—fewer bugs are created and most are more easily fixed when using OOP. Many companies do not rigorously measure quality during the development cycle, but most programmers tend to feel that initial quality is higher with OOP.

Many software development companies and departments attack costs in application development, but ignore the costs of maintenance and upgrades later. Based upon informal polls at presentations and classes, most programmers and managers estimate that programmers in their companies spend from 50% to 80% of their time on maintenance, meaning that only 20% to 50% is spent doing new development. Other studies indicate that only 30% of programming time is spent on new development, 14% on fixing bugs, 14% in adapting applications to environments, and 42% in adding to the application (enhancements)[4]. So, in one way or another, as much as 70% of a programmer's time can be spent on maintenance. Over the life cycle of an application, the total cost of maintenance can be four to eight times the original development cost.

Maintenance never seems to make programs smaller—every change generates more code. And more code generates the possibility of more bugs. The effect is similar to compound interest. OOP reduces this code bulk while contributing fewer bugs. This combination reduces both the number of update iterations and the effort required for each. Over the life of an application, these savings are substantial. In the real estate example cited earlier, maintenance during the first year was limited to half a person, but that included three major revisions. This aspect of OOP is being measured, but the results are not yet available. One thing is clear—managers striving to cut costs should aim at the biggest contributor: maintenance.

Managers striving to cut costs should aim at the the biggest contributor: maintenance.

Revenue

Save, save, save! All software companies and IT departments want to save money by producing high-quality software and services. However, if the customers stop buying the product, the software company dies. If the end users turn elsewhere for computing needs, the IT department is budgeted out of existence. Cutting costs will only help these businesses to survive. To expand, they need ways of increasing revenues or convincing someone else to increase the budget. How can OOP help?

Increased revenues generally result from expanding into wider markets or from increasing current market share. Most companies widen their software market by increasing the scope or number of applications to cover more business segments, by increasing the number of platforms supported, or by doing both. Increased market share results from either offering more value-add, outselling the competition, or a combination of both.

Platform Portability

Moving to more development platforms requires software portability. Today, industry-wide standards are nearly completed for Smalltalk and C++, and both languages are available on most

The OOP Business Case Return on Investment

Return on Investment

Initial Investment			– Initial Investment
Project 1	Reuse savings + Maintenance + Revenue	– Project Cost	Net
Project 2	Reuse savings + Maintenance + Revenue	– Project Cost	+ Net
Project 3	Reuse savings + Maintenance + Revenue	– Project Cost	+ Net
Project n	Reuse savings + Maintenance + Revenue	– Project Cost	+ Net
			Return on Investment

Figure 3. A postive return on investment is built over time

system platforms. With a traditional language, these two facts imply portability. With OOP, applications are made up of objects and statements. The standards address the statements, but not the class libraries from which objects are created. Therefore, developers shopping for an OOP language need to consider what platforms the language provider supports and how consistent the class libraries are across these platforms. Most providers, including IBM, are trying to create libraries that are consistent across a variety of platforms, thereby allowing developers to develop and test on one platform and easily move the application to another one.

The Object Management Group (OMG), a consortium of hundreds of software companies, has already drafted standards for object interoperability across languages and platforms. The Common Object Request Broker Architecture (CORBA), from OMG, defines standards that will allow different objects created in different languages on different platforms to communicate with each other consistently and cleanly. At this point, portability becomes reality. IBM developed System Object Model (SOM) based upon these standards.

The tools and languages currently available and those being developed support client/server applications and graphical user interfaces. While nonprogrammable terminals are supported, the technology is geared toward flexible GUIs through intelligent workstations connected to data and application servers. Many developers with host-based applications are being pressured to connect to the desktop PC as end users become more comfortable with iconic, graphical interfaces. OOP languages generally come with a rich library of GUI classes that make it relatively easy to create, and later change, the user interfaces.

Icons map well to GUI objects. With a visual programming tool such as VisualAge, programmers can construct or modify screen layouts quickly and accurately. The resulting applications are much more appealing than text-based applications, making them much easier to sell.

Customization

The greatest justification for moving to OOP lies in customization. Customers have made the transition from, “I will take whatever you can offer” to “Here is my list of requirements—meet it, or I can probably find it elsewhere.” While a product’s functions and features are still key buying factors, many customers want, need, and, most importantly, will pay for applications tailored to their specific needs.

OOP enables programmers to quickly add to and change an existing object without tearing the application apart. The new subclass contains only the changes, inheriting the rest of the data and methods from the original class (superclass). Later, the programmer can use the browser to scan through these newly created classes to see what changes were made. Encapsulation and good OOP design enforce clean object interfaces—it is more difficult to create “spaghetti code” and “patch on patch” messes, even in the heat of customer crises. Polymorphism allows the interface to the surrounding application to remain the same in most cases, isolating changes and reducing the chances of introducing a new problem while fixing another.

For developers doing customization today, this technology is a natural next step to increase the number of customers that can be supported. For others, OOP might be the way to raise their value add to the customer, close more business, and make more of that lucrative services revenue.

The Business Case

Anyone considering object-oriented programming should remember that the benefits are primarily long-term ones measured across a number of projects. It will take time to build up a library of reusable parts. But as programmers gradually learn their way around the class libraries, the application offerings will gradually expand. Figure 3 shows a summary of the business case.

What's Next?

Even when the business case shows a positive return, software companies must have a plan to move from one technology to another without disrupting the revenue stream. IT managers must make the transition without a loss of service and within budget. Most developers who have successfully made the transition used one of two methods.

Method 1. The most popular method is to take a small subset of people, remove them from day-to-day responsibilities, send them to some classes, give them a project to work on as part of their training, and find a mentor who can help steer them in the right direction. Using a mentor will greatly decrease the learning time and greatly increase the chances for eventual success. New OOP programmers might create what they think is an elegant, object-oriented design but, without experience, cannot even determine whether it is good or bad. This is where a mentor comes in. Some companies hire mentors, and others contract with them to varying degrees throughout the first application design. After the initial design, many companies begin to add others to the team, using experienced programmers to mentor their peers.

Method 2. The second method is a quicker method for companies that face tighter deadlines. These companies hire skilled mentors and OOP programmers up front to do the initial design and much of the high-level implementation. While this is happening, other employees begin training and are added to the team later for the bulk of the implementation. For this to be successful, it is important to employ skilled, experienced (and successful) programmers from the start. The investment level in doing so is high—good OOP people are not inexpensive—but the success ratio is also very high.

Regardless of the method chosen, those managing an OOP project must be trained along with the programmers. The iterative, prototyping nature of design and implementation does not match up well with the traditional design/code/test checkpoint system. Tossing out an initial prototype design is not a setback—progress has been made. On the other hand, managers must recognize when iteration becomes over-analysis. Measurements change from lines of code written to code reused.

Many management adjustments are required. Often, companies follow the "recipe for disaster"—they set an aggressive schedule, use inexperienced and recently trained people, try to save money by not hiring anyone with skills, track the project like any other important project, and, in the end, blame the failure on the technology (or the tools or other individuals). Programmers and managers who are asked to take on such a challenge (or already have) should stop and reconsider reality. It takes time to develop OOP skills, and these skills are essential for success. One way to turn these companies around is to hire a good OOP consultant to evaluate the project, educate management, and recommend changes.

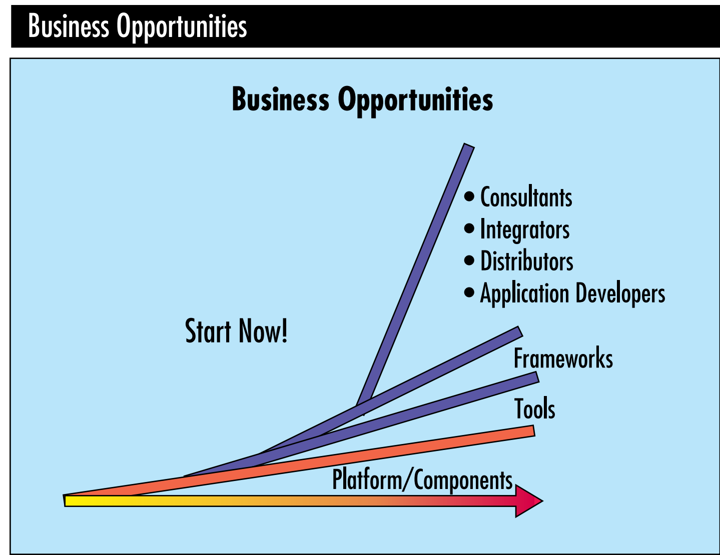


Figure 4. OOP is creating many business opportunities

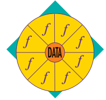
Summary

The object-oriented business is expanding rapidly, and, like PCs, the offerings will only grow in number and function, and prices will constantly change over the next several years.

Figure 4 shows the OO industry as it matures. The platform and basic components are nearly in place, and frameworks are now under development. Once this occurs, widespread use will begin to increase rapidly. The companies that invest today to understand this technology will be

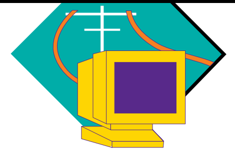
in the best position to evaluate these new offerings and take advantage of them quickly. This technology is like a train in motion—you can hop on anytime, but you have a better chance if you get a running start. At the very least, even if you are not yet sold on object-oriented programming, make sure you understand it enough to make a good decision.

1. Love, Tom. *Object Lessons: Lessons Learned in Object-Oriented Development Projects*. New York, NY: SIGS Books, Inc. ISBN 0-9627477-3-4.
2. Harmon, Paul and Taylor, David A. *Commercial Applications of Object-Oriented Technologies*. Reading, MA: Addison-Wesley Publishing Company. ISBN 0-201-63336-1.
3. International Data Corporation. *Object Technology: A Key Software Technology for the '90s* (white paper). Steve McClure, New Software Technology, at International Data Corporation.
4. Goldstein, Neal and Alger, Jeff. *Developing Object-Oriented Software for the Macintosh: Analysis, Design, and Programming*. Reading, MA: Addison-Wesley Publishing Company. ISBN 0-201-57065-3.



Dan Hattenberger, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Hattenberger recently moved from the AS/400 Laboratory in Rochester to join the Object Deployment group within Solution Developer Operations to create OO solutions for AIX and OS/2 . He has a BA in Mathematics from St. Mary's College of Minnesota.

SNA Server/6000 Performance on SMP



By Dov Bulka, Michelle Hermanson, Chris Selvaggi, and Julia Sime

This article describes how IBM achieved extremely high scalability during the port of the uniprocessor SNA Server/6000 to a Symmetric Multiprocessor (SMP) environment. Lessons learned should help other developers porting to an SMP environment.

AIX SNA Server/6000 is a high-function, high-performance communications stack for AIX. It supports legacy System Network Architecture (SNA) applications, such as 3270 emulators, LU0 home-grown applications, and LEN-level LU6.2, to enable connections to OS/400®, OS/2®, MVS™, and many other platforms. SNA Server/6000 supports Advanced Peer-to-Peer Networking® (APPN®) to reduce configuration complexities and overhead while maintaining reliability and robustness.

AIX 3.2.5 has supported SNA Server/6000 for several years. SNA Server/6000 2.2 became available several months ago in an AIX 4.1 Uniprocessor (UP) configuration. SNA Server/6000 3.1, which has efficient SMP performance, is now available under a beta-test program and will be generally available later in 1995.

Design Background

Before going through the porting process, let's look at the design before the port. Some design decisions made during the product's infancy worked well into our SMP plans; others did not. Some design features became major hurdles in our attempt to use the SMP configuration efficiently.

SNA Server/6000 consists of a set of user space daemons, kprocs, pseudo-device drivers, and callback functions provided to lower-level pseudo-device drivers. The number of processes active in SNA code can range from a dozen to thousands at any given time. The user space daemons provide all types of control and start/stop services. The kprocs, device drivers, and callback functions participate with the daemons in start/stop services. They also provide steady-state services on their own, completely within the kernel.

Once links and sessions have been established, only three threads participate in the steady-state flow of data over the network, as shown in Figure 1.

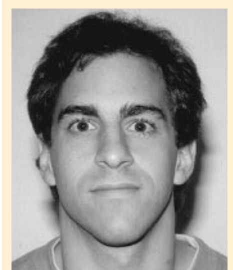
- **The Transaction Program (TP)** is an application that uses SNA Server. Provided by the customer or another product, the TP's thread calls the SNA Server interface that invokes the SNA Server/6000 device driver.
- **The Half Session (HS) kproc** provides data flow control and transmission control services at one end of a session. SNA Server provides one HS kproc for every active session.
- **The Data Link Control's (DLC's) kproc** is provided by the DLC pseudo-device driver.

These three threads—TP, HS, and DLC—are involved in sending data over the network. The TP thread, running in SNA Server's device driver code, puts the data into system memory buffers (`mbufs`) and queues it to HS. After performing its transport layer function, HS calls the DLC device driver interface. This, in turn, calls the adapter device driver interface, passing the data to the adapter while still on the HS thread.

When data is received from the link, the communications adapter passes the data to the bottom half of the DLC device driver, which quickly queues it to the DLC kproc. The kproc enters the bottom half of the SNA device driver and queues the data to HS. Once HS completes its processing, it then queues the data to the TP thread, which receives the data when it enters the read call of the SNA device driver. At this point, the

Dov Bulka

Michelle Hermanson


Chris Selvaggi

Julia Sime

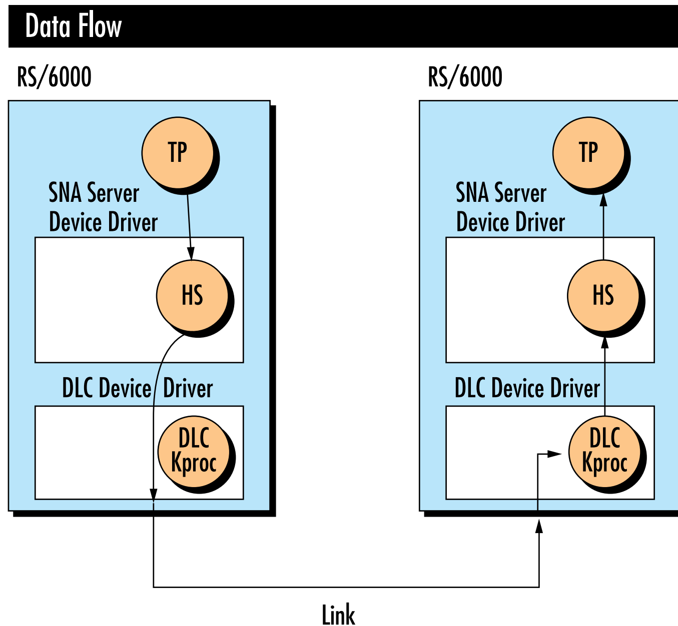


Figure 1. Steady state data flow

data transfer is complete, and the TP can process the data as it chooses.

Many of these data transfers can be active simultaneously. Memory shared between these active threads is protected by locks. This method of concurrency control functions correctly on both UP and SMP systems.

Although the code ran correctly on the SMP machine, we encountered a substantial performance problem. The early performance measurements showed that TP response time over 4-way SMP was worse than UP response time by a factor of 10.

Fortunately, the design did not require major changes to improve scalability; however, the changes made were not always easy or obvious. The changes included converting lock calls, refining lock granularity, shortening path length within locked sections, restructuring to eliminate excessive locking, reducing cache misses, and pooling memory. The primary benefit of these changes was to reduce contention for shared resources.

The Thundering Herd

Performance analysis showed that lock contention was a major cause of the initial poor SMP performance. Several critical locks had high *miss rates*—the result of a thread failing to acquire a lock, which requires the thread to wait. The high miss rates resulted from the `lockl()`—a blocking lock—system call used by SNA Server/6000. A blocking lock suspends the thread until the lock is free. When the lock is freed, all suspended threads are placed on the run queue to contend for the lock. When lock contention is heavy, this behavior is very inefficient.

In a distributed application, it is common to have several threads waiting for a critical lock. When a lock is freed, the suspended threads are signaled; all but one are put back to sleep, resulting in many wasted context switches. A specific thread bounces between the suspended and run queues before actually acquiring the lock. This problem is sometimes called the *thundering herd* because all the threads vie for the lock each time it is freed.[1]

AIX 4.1 provides a new type of lock called a *simple lock*, which can be used to solve the thundering herd problem. An important feature of the `simple_lock()` call is spinning. A spin lock allows the waiting thread to keep its processor, repeatedly checking the lock in a tight loop (spin) until the lock becomes free. This is ideal when locks are held for a short time. Spin locks avoid costly context switches. By converting to simple locks, we significantly reduced our lock miss rates. The move from `lockl()` to `simple_lock()` was the most important modification we made—giving us the largest performance boost on SMP.

The simple lock performance improvements introduced several new problems into the existing code: two benefits of `lockl()`—signal delivery and recursion—were not compatible with simple locks. Our code was heavily dependent on signal delivery to alert the kproc to conditions that require termination. The `lockl()` allows a thread that is waiting for a lock to receive signals; simple locks do not. To avoid deadlock on kproc termination, the code had to release all locks before it could signal a kproc.

Lock recursion or nesting is the other `lockl()` feature not available with simple locks. In SNA Server/6000, code modules are called from several different paths. When using `lockl`, these shared modules could lock a resource without worrying if the resource might already be locked on this path. If the lock was already held, it would simply be nested. To convert to simple locks, we analyzed each location to decide if the lock was already held. We restructured the code so that many nested calls were removed.

[1] Campbell, Mark et al. *The Parallelization of UNIX System V Release 4.0.* USENIX. (Winter 1991).

Other Lock Issues

After overcoming the thundering herd obstacle, we had more work to do to achieve high SMP scaling. Next, we refined locking granularity, reduced the time for which locks were held, and eliminated some locks.

First we eliminated the coarse grain locks from several places in HS. Instead of using one lock to protect a region of code, we protected the integrity of various shared data structures using locks specific to those data structures. Because unrelated structures maintained separate locks, they could provide simultaneous access. In a sense, we traded region locks for data locks.

We also reduced the time that locks were held. We converted linked lists to hash tables to speed up search, access, and update operations. Since those operations were critical code sections, they were protected by locks. The faster an operation completes, the faster the lock is released to allow the next operation to proceed.

Converting linked lists to hash tables significantly increased the speed of execution on UP and SMP configurations. It also reduced the instruction count—always a welcome change.

Time-consuming operations, such as copying application (TP) data from user space to kernel space, were removed from the scope of critical sections. This significantly reduced the length of time in the locked section of code. The combination of the short time during which locks were held and the use of simple locks enabled waiting threads to obtain the lock after a short spin and to avoid a costly context switch.

We eliminated some lock calls. For example, SNA associates each system `mbuf` (communication buffer) with its own structure—the `snaipcm`. A one-to-one mapping exists between a `snaipcm` and an `mbuf`. As the `mbuf` pointer travels along the data transfer path, it is passed as an argument from one routine to another.

A macro, `MBUF_SNAIPCM`, is used in these routines to retrieve the `snaipcm`, given the `mbuf`. This macro call was not a performance problem on AIX 3.2, because the `snaipcm` was stored in the `mbuf` itself, so the `MBUF_SNAIPCM (mbuf)` call translated to a simple pointer dereference.

Changes made in AIX 4.1 made it difficult, if not impossible, to embed the `snaipcm` in the `mbuf`. Since only the `mbuf` was passed from routine to routine, it would have been necessary to change hundreds of code paths to pass both the `mbuf` and the `snaipcm`. For the first pass, we redefined the macro to map the `snaipcm` to the `mbuf` using a hash table. Since several different processes could access this hash table concurrently, it required a lock. We measured the performance hit, and it was considered acceptable on a UP system.

The performance hit of the new lock calls was much greater on an SMP than on a UP system. With four processes running at the same time, lock contention for this hash table lock was very high. We eliminated this bottleneck by creating the `mbuf` to `snaipcm` mapping when the `mbuf` arrives, then passing the mapping around instead of the `mbuf` pointer alone.

We constructed a new structure for `mbuf` arrival that contained both pointers to `mbuf` and the associated `snaipcm`. This specific mapping occurred in hundreds of places throughout the code. Each modification demanded caution to prevent errors from being introduced into working code. In fact, we limited the scope of the modifications to code paths considered critical to performance. The result was a very significant reduction in the number of lock calls.

The number of processes active in SNA code can range from a dozen to thousands at any given time.

False Sharing

When unrelated entities share a memory unit such as a cache line, main memory frame, or a disk page, it is called *false sharing*. The sharing is false because the entities are functionally unrelated and their proximity is usually unintended.

Two "hot" lock variables were declared beside one another, and the compiler placed them accordingly. Consequently, both locks shared the same cache line. It was not uncommon for these two locks to be updated by different threads simultaneously. On an SMP machine, that creates a "cache consistency storm" in which each CPU invalidates the other's cache line. This results in wasted bus traffic while the hardware attempts to keep the caches consistent. To resolve this issue, we separated the lock definitions so each occupied a separate cache line—a typical cache line size on our RISC System/6000 platform is 32 bytes.[2]

Posting While Holding a Lock

SNA Server/6000 has several cases in which one process performs a task on behalf of another, which requires Interprocess Communications (IPC). Consider the example of the Half Session kproc, which transfers data over a link for a TP.

[2] Debora Blakely-Fogel. "Porting Applications to the AIX 4.1 OS SMP Environment," *AIXpert* (November 1994).

Tips for Porting to an SMP Environment

We learned the following lessons in porting SNA Server/6000 to SMP:

- Prevent the thundering herd by replacing blocking locks such as `lockl()` with spin locks such as `simple_lock()`.
- Eliminate locking if possible.
- Reduce the size of a critical section by moving unrelated code outside the scope of the critical section.
- Reduce the duration in which locks are held by implementing more efficient algorithms.
- Avoid false sharing.
- Do not post another thread while holding a lock; post after releasing the lock.
- Manage your own memory pools to reduce allocation time on critical paths.

To do this, the data is first placed on a queue under the TP thread. Next, HS is notified via the kernel service `et_post`. This system call is considered to be the fastest IPC method. It is so fast that the post recipient can be scheduled to execute almost immediately, especially on an SMP machine with an available processor. However, if the recipient must acquire a lock to access the data and the sender holds the lock, then `et_post` loses its efficiency. This was the case with HS and TP data. We resolved this by performing the `et_post` after relinquishing the lock, which enabled the HS to run unimpeded.

Memory Pooling

A *heap* is a pool of memory chunks of various sizes managed by the OS. When it is accessed via the kernel service `xmalloc()`, a system lock is held. SNA manages several of its own pools of same-size memory chunks, which has two important benefits:

- Avoids the overhead of contention on a system lock
- Avoids the path length required to manage the complexity of the system heap

We can return memory from a pool of same-size chunks with very few instructions. We chose memory allocated frequently on critical paths and converted from `xmalloc()` calls to our own internal memory pooling.

Results

Before beginning the development work, we benchmarked SNA Server/6000 performance on a simple network. The objective was to assess the scalability of our code on an SMP system and to estimate performance gains going from UP to SMP. To keep all other variables intact and focus on scaling issues, we used the same two RS/6000 Model J30s, both 4-way SMP machines. We measured first with only one processor enabled on each machine. Then we took the same measurements with all four processors enabled. Since the AIX 4.1 SMP kernel is estimated to be 15% slower than the AIX 4.1 UP kernel, the one processor configuration is only an approximation of UP performance.

To create high-volume traffic, we invoked 150 sessions in which a 2 KB data buffer was repeatedly bounced back and forth between source and target TPs. With that level of traffic in the background, we measured the response time of a 1 MB data transfer from the local to remote machine over a dedicated 16 Mbit/second Token-Ring connection. Since we had no interest in the performance of the DLC and the adapter, we separated the measured data transfer TP from the other 150 sessions by routing them over distinct DLCs. The data transfer TP ran over a Token Ring while the rest of the sessions ran over a different link. If all traffic had been run over a single communications link, the link would probably have become the bottleneck and negated the advantage of having multiple processors.

When we began our work, our performance was 10 times slower on four processors than on one processor. When we were finished, response time on the 4-way configuration was 3.3 times faster than the 1-way configuration. This is a scaling factor of 3.3 out of the theoretical maximum of 4.0 on a 4-way SMP. This significant speedup—from .1 to 3.3—results from multiple SNA Server/6000 threads executing simultaneously on multiple processors.

Future Plans

We did not implement all performance enhancements because of tight deadlines; some, such as frequently accessed hash table locks, were deferred. During development, we considered replacing the table locks with fine grain bucket locks that would lock individual buckets as opposed to locking the whole table. That would enable two distinct buckets to be accessed simultaneously. Even though this feature was deferred,

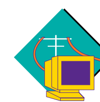
it is an important tool for SMP developers. It could become useful when contention for a table lock is high.

What about processor affinity? It would be good if threads would dispatch on the same CPU that they last visited. If a thread can keep running on the same CPU, chances are good that the cache still contains relevant data. Otherwise, the cache must be drained to make room for the current thread's data. Although the AIX 4.1 kernel attempts to help with processor affinity, it is not perfect.

We considered binding the HS kprocs to the various available processors. The disadvantage of this design would be that a kproc may occasionally remain on the run queue since its target processor is busy, even though other processors might be available.

Acknowledgments

Many people contributed to the SNA Server/6000 product family described in this article. Some design decisions made years ago played an important role in our successful SMP implementation. The authors would like to acknowledge the work of past and present members of the AIX SNA organization who are too numerous to list. Special thanks also goes to Herman Dierks from the IBM Austin Performance group, who contributed many valuable insights into SMP efficiency.

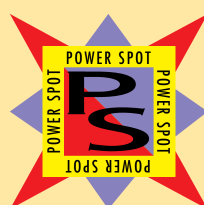


Dov Bulka, IBM Corporation, 4205 South Miami Boulevard, Research Triangle Park, NC 27709. Internet: dov@ralvm5.vnet.ibm.com. Dr. Bulka is an advisory programmer working on SNA products for AIX. He has a BA in Mathematics from Brandeis University and a PhD in Computer Science from Duke University.

Michelle Hermanson, IBM Corporation, 4205 South Miami Boulevard, Research Triangle Park, NC 27709. Internet: mmh@vnet.ibm.com. Ms. Hermanson is a developer for the SNA Server/6000 group and works primarily on kernel extensions and device drivers. She has a BS in Computer Science from the University of Michigan.

Chris Selvaggi, IBM Corporation, 4205 South Miami Boulevard, Research Triangle Park, NC 27709. Internet: zamboni@vnet.ibm.com. Mr. Selvaggi, a senior associate programmer, is a member of the AIX SNA team developing SNA Server/6000 and AnyNet/6000 Sockets over SNA. He has a BS in Computer Science from the Georgia Institute of Technology.

Julia Sime, IBM Corporation, 4205 South Miami Boulevard, Research Triangle Park, NC 27709. Internet: julia@vnet.ibm.com. Ms. Sime is a development programmer working primarily on kernel extensions and device drivers using C and C++. She is currently the chief programmer for the AIX SNA Server/6000 Version 3.1. She studied computer science at California Polytechnic University in San Luis Obispo, California.



High-Performance World Wide Web Server for 1996 Olympic Games in Atlanta

Want to know what's happening at the Atlanta Olympics next summer? Now you can surf your way to a world of information on the 1996 Olympic Games World Wide Web Server at the URL below:

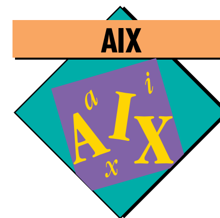
http://www.atlanta.olympic.org

This new server will provide continuously updated facts, photos, videos, and audio content—all aimed at providing the latest possible news on Olympic events.

Many Internet users from around the world are expected to access the new Olympic information site. In order to handle this enormous workload, the site will use IBM's high-performance RS/6000 Scalable POWERparallel (SP) system as a server. As the volume of online Olympic Games information expands, the SP server can scale up easily—with additional processors, memory, and disk storage—to accommodate the additional processing and data needs.

The RS/6000 SP will interface with IBM ES/9000™ and AS/400 systems, which run the sporting event results and information/communications applications for the 1996 Games. Working together, the systems will offer Internet users access to real-time news and information about the 1996 Games.

Re-engineering the Time-To-Market Process



By Eddie Ho, Eric Dunn, and Peter Stoll

The RISC System/6000 ®(RS/6000™) running AIX is the premier platform for re-engineering manufacturing processes. The open system framework enables the best applications to be integrated into a manufacturing environment. This article describes the benefits of combining several IBM software applications to reduce the time to market for manufacturing companies.

As we approach the 21st century, the dominant themes in the computer industry are integration and interoperability. All areas of business—manufacturing, finance, human resources, and distribution—are moving toward sharing information. This article describes how the RS/6000 and AIX can combine two manufacturing applications that use a common database. The resulting improvement in communications between the product design and manufacturing organizations in a company can dramatically reduce the time to get new products to market.

The Computer-Integrated Manufacturing (CIM) initiative in the manufacturing industry brings together software packages that integrate the end-to-end development and manufacturing process. Figure 1 illustrates shifting the paradigm from a sequential to an iterative workflow model. As part of CIM, IBM's ProductManager and Computer Integrated and Interactive Manufacturing (CIIM) from Avalon Software work together with the associated Computer-Aided Design (CAD) and shop-floor applications to manage the flow of information between product design groups and manufacturing.

One of the keys to successful manufacturing is the speed at which new products are brought into the marketplace. A winning strategy for any manufacturing company is to obsolete its own products with new ones before the competition does. ProductManager/CIIM integration provides tools to enable a company to accelerate the pace of new product introductions and to remove products quickly when their replacements are

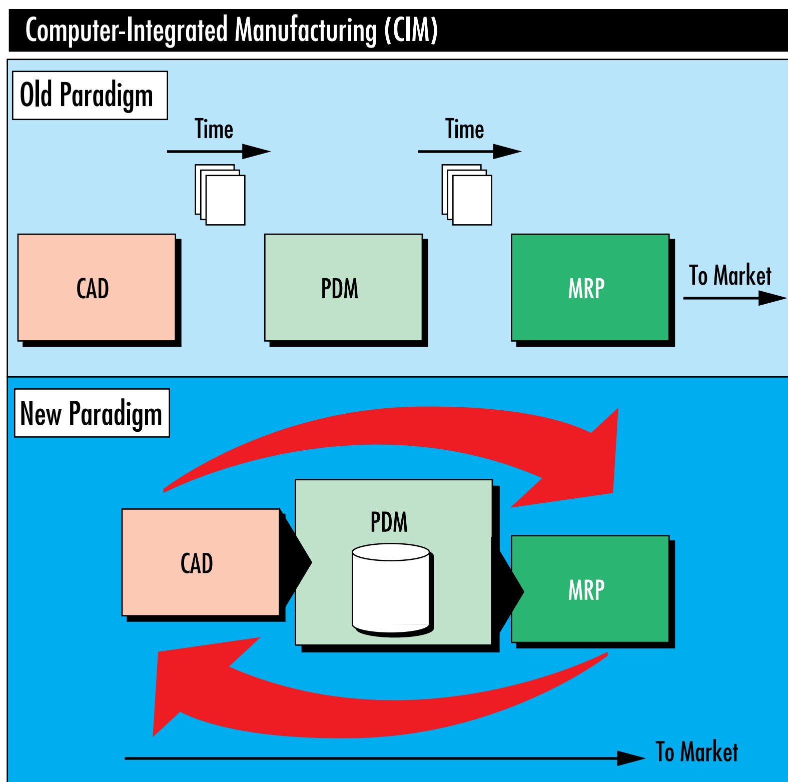


Figure 1. Computer-integrated manufacturing

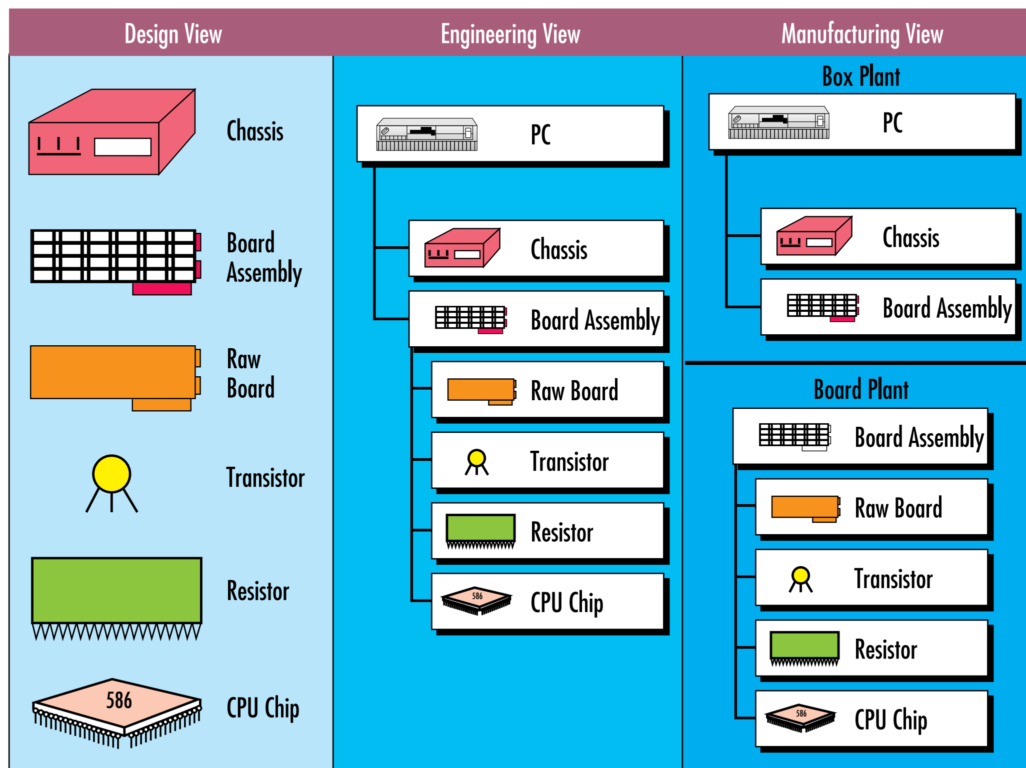


Figure 2. Enterprise bill-of-material integration

ready. PM and CIIM tie together existing software packages that cover all relevant development and production functions including CAD, product structure definition and engineering records, manufacturing resource planning, and shop-floor support activities such as routings and operator instructional aides.

IBM ProductManager

Collapsing the cycle-time-to-market presents the manufacturing industry with a fundamental opportunity to re-engineer its core business process to its competitive advantage.

IBM's ProductManager manages the engineering information associated with the product development and manufacturing processes. It provides the seamless integration for most of the existing CAD and MRP products in the marketplace. Its three major functions, along with other services, enable product development groups to control the design phase of a new product and introduce it smoothly into the manufacturing process.

Engineering change control process: Enables developers to design and release products incrementally to manufacturing. All information related to any product—such as materials, tools, and drawings—can be stored by using a version control process to account for changes to the specifications.

Bill-of-material management process: Enables developers to build a bill-of-material and associated manufacturing process information during the new product design phase. It also provides a robust set of product structure functions for building and changing the content of a product bill-of-material, and then releasing the final version to manufacturing. Figure 2 represents the vertical view of each group to their respective work type.

Electronic foldering: This state-of-the-art workgroup application enables development and manufacturing groups to share any data related to managing the life cycle of a product: design, production, service, and end-of-life. Authorized individuals can check out documents from the Document Manager vault, rework the document to fix a design flaw, then send it to all the appropriate persons who need to know about the change via an E-mail messaging system.

The traditional approach to introducing a new product involved organizations only when the product reached a development stage that required their service. This sequential approach increased the time to bring a new product to market. Figure 3 illustrates the workflow using the new foldering concept.

Electronic foldering changes the paradigm for the new product introduction process by introducing the concept of *concurrent engineering*—the development and manufacturing departments working in parallel during the early phase of product development. ProductManager uses this methodology to collapse the total cycle time. When design engineers, manufacturing engineers, cost engineers, purchasing buyers, production schedulers, documentation writers, and marketing and advertising people can all communicate concurrently instead of serially, the time between decision and execution is sharply decreased.

ProductManager provides a turnkey solution for facilitating the cycle time reduction initiatives for the new product introduction process.

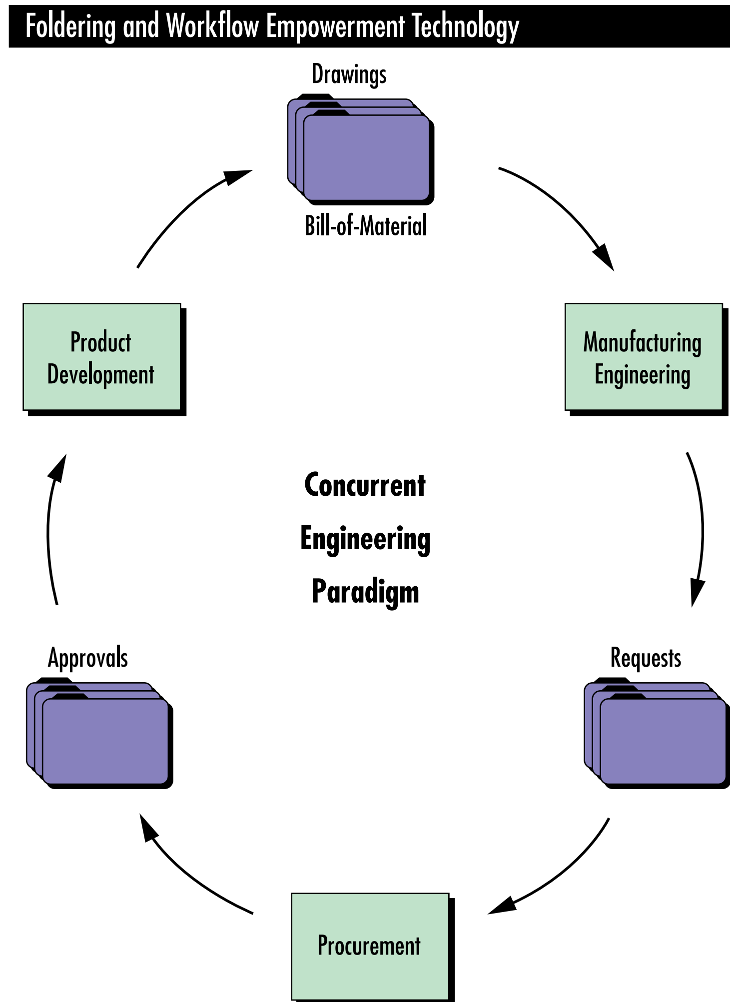


Figure 3. Foldering and workflow empowerment technology

Computer Integrated and Interactive Manufacturing (CIIM)

CIIM is an Enterprise Resource Planning (ERP) product from Avalon Software Company designed for manufacturing logistics and inventory control. Figure 4 shows the modules in CIIM.

The CIIM product enables manufacturing organizations to plan production schedules, order the appropriate materials and components, and control the fabrication of assemblies and the subsequent aggregation/shipment of a completed customer order.

The interaction between ProductManager and CIIM is confined to the inventory control and bill-of-material module. Product information produced by ProductManager is communicated to a manufacturing system like CIIM. The product design area is responsible for defining its content and structure as well as the process required to fabricate or assemble it. This data is sent to the ERP system to drive the component planning and production scheduling process.

The communication between an engineering records system and an ERP or manufacturing system has traditionally been sequential, as described earlier. The formal communications relationship between many development labs and their manufacturing counterparts could be viewed as “throwing the design over the wall” from the lab to the factory. The lack of integration systems was part of the barrier between the two.

The ProductManager/CIIM integration is specifically designed to remove those barriers between development and manufacturing. Sharing processes such as assigning part numbers, engineering changes, and bill-of-material data is one way to accomplish this. Data is updated in CIIM as it is created or modified in ProductManager.

The electronic foldering process in ProductManager and shared data between the development and manufacturing organizations represent the critical success factors in enabling the new product introduction cycle time to shrink dramatically. Consider the possibility of a design person, with a new idea, signing onto a CAD tool at 8:00 A.M. and having the idea with associated documentation approved by everyone

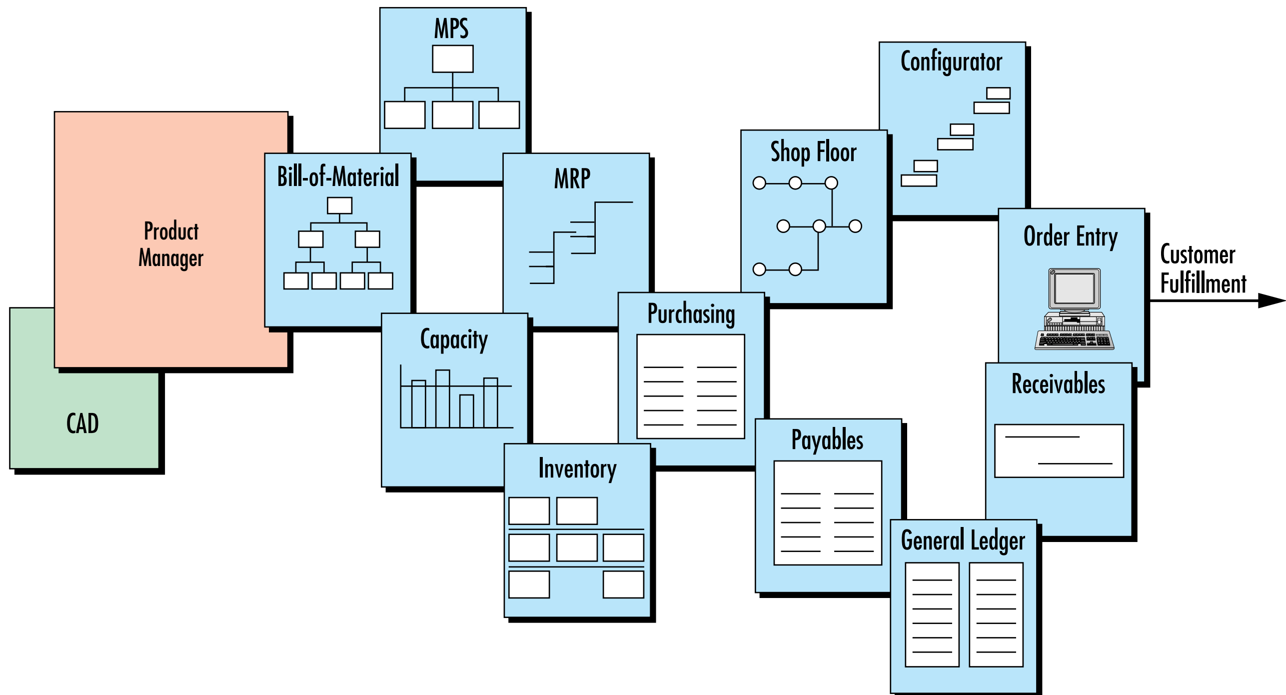


Figure 4. Enterprise resource planning

in the development/manufacturing/marketing/ distribution communities by the end of the day. That could produce a true competitive advantage.

Eddie Ho, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Ho is a senior programmer in the AIX Executive Briefing Center. He has a BS in Computer Science from the University of Wisconsin and an MS in Computer Science from North Dakota State University.

Eric Dunn, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Dunn is an staff development analyst in the RS/6000 Technical Services area. He has a BS in Management Information Systems and an MBA from Old Dominion University in Norfolk, Virginia.

Peter Stoll, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Stoll is a senior manufacturing analyst in the RS/6000 manufacturing area. He has a BA from the University of California at Los Angeles.

Shortcut to IBM Product Information on the Internet

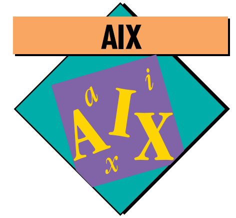
Now, you have an easy route to the very latest IBM product information. With an Internet E-mail address, you can subscribe to an IBM U.S. Announcement Listserver. You need only take a few minutes to set up your profile, and the service will automatically send you news from the categories you select as soon as it becomes available. News is sent in ASCII text format.

To initiate the service, send an E-mail note to the following address:

announce@webster.ibmlink.ibm.com

Leave the subject line blank and type the keyword subscribe in the body of the note. Detailed instructions and a category form will be sent to you shortly.

Signals in Multithreaded Programs



By Chary G. Tamirisa

POSIX.1c, the standard for parallel programming, is part of the POSIX series of standards. The behavior of POSIX.1 signals in a multithreaded program has undergone dramatic changes during the standardization process of POSIX.1c. This article compares the traditional POSIX signals model with the new model in a multithreaded environment. The article describes the signal model in AIX 4.1 and provides some guidelines for signal handling in applications. It also describes the DCE signals model with tips on porting DCE pthreads-based applications to AIX 4.1.

AIX 4.1 provides POSIX™ threads support based on draft 7 of the POSIX.1c standard for multithreaded programming. The `pthreads` standard defines several Application Programming Interfaces (APIs) that are provided in the `pthreads` (`libpthreads.a`) library.

The POSIX.1c draft 10 has become the final standard. The basic signal model has not changed from draft 7. This article describes the changes to the traditional signal model in POSIX.1 and provides some important guidelines on how to handle signals in the presence of multiple threads in a process.

The article also addresses the porting issues of applications written to the `pthreads` library supported on AIX Version 3. Although AIX Version 3 did not support kernel threads, the `pthreads` library from DCE provides threads support. Since the DCE `pthreads` library was based on the earlier draft 4 of POSIX.1c, the article explains how to modify existing programs when porting from AIX Version 3 to Version 4. This article has several parts. A discussion of the traditional UNIX (POSIX.1) signal model presents an overview and provides the context necessary to explain the changes made to it in the multithreaded programming model in POSIX.1c. The new signal model is then described with an example. Programming guidelines provide help in writing well-behaved multithreaded programs with respect to signals. Finally, we discuss migration issues related to signals for DCE-based multithreaded programs.

See "Introduction to Multithreaded Programming" (*AIXpert*, November 1994) and "Porting DCE Threads Programs to AIX 4.1" (*AIXpert*, August 1995) for details of the POSIX.1c interfaces.

Traditional (POSIX.1/UNIX) Signal Model

A traditional POSIX.1 process has one signal thread of control. POSIX.1 specifies the signal model.

Signal Handlers

A process can register a signal handler through `sigaction()` to capture signals. There are two types of signals:

- **Synchronous signals** result when an instruction executes in the process. The synchronous signals are `SIGILL`, `SIGTRAP`, `SIGIOT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, and `SIGPIPE`.
- **Asynchronous signals** are delivered to the process irrespective of the current instruction it is executing. Examples include `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGCHLD`, and `SIGUSR1`.

Signal handlers are installed using `sigaction()`, defined as follows in POSIX.1.

Chary G. Tamirisa

```
void catcher(int sig)
 {
        foo();
 }
 foo()
 {
        /* Add an element to a linked list */
  }
 main()
 {
        sigset_t set;
        struct sigaction action;

        action.sa_handler=(void)catcher;
        action.sa_flags = 0;
        /*
        ** You can specify a set of signals
           that need to
        ** be masked while in the catcher
           routine.
        ** Note that the signal that is
           delivered is
        ** already blocked while executing
           the catcher()
         ** routine.
        */
        sigemptyset( &action.sa_mask);
        sigaction(SIGINT, &action, NULL);

        sigemptyset(&set);
        sigaddset(SIGINT, &set, NULL);

        sigprocmask(SIG_BLOCK, &set, NULL);
        /* Signal Safe code */
        foo();          /* Add an element
          to a linked
            list */
        sigprocmask(SIG_UNBLOCK, &set, NULL);
 }
```

Figure 1. Code sequence for a single-threaded process

```
#include <signal.h>
int sigaction(int signal,
   struct sigaction *newaction,
   struct sigaction
*oldaction);
```

Signal Masks

A signal can be masked by specifying a block signal mask through the `sigprocmask()` interface. Typically, a process in a critical section that must not be interrupted by a signal handler could block the required signals until the critical section is done. Signal handlers are registered via `sigaction()` by specifying a mask that must be installed when the signal handler executes. Since a signal handler can prevent signals from interrupting, it can guarantee signal safety.

Signal masks are installed by using `sigprocmask()`, defined as follows in POSIX.1:

```
#include <signal.h>
int sigprocmask( int how,
   sigset_t *newset, sigset_t *oldset);
```

Signal Safety in a POSIX.1 Process

For a program to provide signal safety in a single-threaded POSIX.1 process, it could invoke `sigprocmask()` to block the signal when a chosen code sequence is executed, then later unblock the signal. For example, the code sequence in Figure 1 will work in a single-threaded process.

When a signal handler such as `catcher()` is invoked, the operating system installs a signal mask that blocks the received signal from further occurrence. The correct mask is restored when the signal handler returns.

To protect a signal handler from signals other than the currently handled signal, a program can specify additional signals to be masked (`sa_mask`) when the signal handler (`catcher`) executes. This enables a signal handler to safeguard itself against further signals.

POSIX.1c Signal Model

Since a multithreaded program can have more than one thread, several questions arise:

- What happens if more than one thread calls `sigaction()` for the same signal?
- What is the behavior when a thread blocks a signal? Does it block the signal for the whole process, or does it block the signal for the current thread?
- How do we ensure signal safety?

Understanding the new programming model can help answer these questions.

New Programming Model for Signals

The POSIX.1 signal is modified as follows:

- All signal handlers are per-process. Use `sigaction()` to install a process-wide signal handler for all signals.
- All signal masks are per-thread. Use the `sigthreadmask()` function to install a per-thread signal mask. The `sigthreadmask()` is identical to `sigprocmask()` except it now works on a per-thread basis. Use of `sigprocmask()` is not defined in multithreaded programs.

Signal Handlers

Because the signals handlers are installed on a per-process basis, there is one signal handler for a given signal. If more than one thread installs a signal handler for the same signal, the last one installed is used. This means if predictable behavior is needed in a program, signal handlers must be used carefully.

It is good practice not to install signal handlers in libraries; however, if signal handlers are needed, the library must restore the previous handler before returning to the application code. When multiple libraries are invoked from multiple threads, there is always a possibility for one library to overwrite the handler installed by another library, leading to a non-deterministic behavior if the signal handler is supposed to perform different actions.

Signal Masks

Signal masks are per-thread. In fact, POSIX.1c draft 7 defines a new function— `sigthreadmask()`—to install the signal mask on a per-thread basis. This means that a thread cannot modify the signal mask of another existing thread. If a thread installs a signal handler and masks it later when it is executing a function (such as `foo()` in Figure 1), it is possible for another thread, which has not masked it, to receive the signal.

The signal handler can be invoked even though the current thread has masked the signal. Therefore, it breaks the assumption under which the code ran successfully in a single-threaded process. To fix this problem, the signal must be blocked in all the threads. The only way to do this in AIX 4.1 is to rely on the inheritance property for signal masks when threads are created. If the signal of interest is blocked in the main or the initial thread, any thread created thereafter will inherit the signal mask. This is the only way to install a signal mask in all threads in POSIX.1c.

Waiting for Signals (sigwait())

The new API `sigwait()` allows the calling thread to be the focal point for receiving asynchronous signals. This API enables the action on asynchronous signals to be deferred, allowing other threads to continue their work in the presence of asynchronous signals.

If a process wants to receive asynchronous signals, it should invoke `sigwait()`, specifying a signal mask that indicates which signals to wait for. Before invoking `sigwait()`, the thread must block the signals in the mask. This dedicated thread waits for the signals specified. When any signals are received, the `sigwait()` call returns with the signal number. The following sections provide more details.

The New Signal Model

The new POSIX.1c signal model has several important features that allow proper signal handling in the multithreaded environment. The signal handler and signal mask functions are easy to understand because they are extensions of an older POSIX.1 model into the multithreaded environment. However, `sigwait()` is a new API to most programmers. The following sections cover the proper use of the signal action and the signal mask functions.

The signal handler and signal mask functions are extensions of an older POSIX.1 model into the multithreaded environment.

Sigwait()

The `sigwait()` API is specified as follows in POSIX.1c draft 7:

```
#include <signal.h>
    int sigwait(sigset_t *set, int
    *signal);
```

The first argument set specifies the signals to be waited—an input argument to `sigwait()`. The second argument is the signal that is received by the `sigwait()` call—an output argument. The `sigwait()` returns zero on successful return, or returns an `errno` value on error.

It may be surprising to note that `sigwait()` must be called with the signals that it is waiting for blocked! How will the signals be delivered if the signals are masked from the beginning? It is up to `sigwait()` to do whatever is necessary to receive signals delivered to the process or to the `sigwaiter` thread. Some implementations install a signal handler when `sigwait()` is called and unblock signals that are specified. When one of the signals it is waiting for is delivered, it reblocks the signals and returns with the signal. It is important to remember that there is an invariant that is maintained: the signal mask of the calling thread is the same before and after `sigwait()` returns.

How to Use sigwait()

To use `sigwait()`, create a dedicated thread in which `sigwait()` is invoked to capture signals of interest. The main issue with `sigwait()` is that signals specified in the set argument must be blocked—at least in the calling thread. Two interesting scenarios describe the problems related to blocking.

Scenario 1: Using `sigwait()` with multiple threads that can receive signals. It is possible to have multiple threads that are eligible to receive a `sigwaited` signal if they do not block the signals being waited for in `sigwait()`. The AIX 4 library ensures that the signals received are directed to the `sigwaiter` thread, but there is a side-effect of signal delivery. A thread in a system call that is interrupted will return with errno global variable set to EINTR. The application must handle this return properly. If the application does not want the system calls to be interrupted this way, it must block the signals in the threads that issue system calls.

Scenario 2: `Sigwait()` in a loop. Imagine that you want to receive asynchronous signals in a thread, process them, then return to `sigwait()` in a loop. During the time between the return of `sigwait()` and when it is invoked again, the dedicated thread is not really in `sigwait()`. If a signal of interest occurs during this time, the kernel will deliver the signal to any other thread that has the signal unmasked (assuming the `sigwaiter` thread has blocked the `sigwaited` signals). If there are threads that have not masked the signal, the signal delivery can cause default action to take place, which might include termination of the process.

It is up to sigwait() to do whatever is necessary to receive signals delivered to the process or to the sigwaiter thread.

To safeguard against the two scenarios, block the `sigwaited` signals in all the threads.

POSIX.1c does not provide a global signal mask, yet a program needs to block signals in all the threads. To accomplish a global mask, the first action is to block the signals in the main or initial thread before creating any threads. Using the property of inheritance, the block signal mask will be propagated to all the threads in the process. If no thread unblocks the signals in the mask, a process has been created in which the signal mask is set up in all of the threads. A `sigwaiter` thread can then receive signals and process them as needed.

Guidelines for Using sigwait()

There are certain rules to follow for obtaining predictable behavior from `sigwait()`.

- For handling asynchronous signals in a multithreaded program, the recommended method is to convert asynchronous signal actions to synchronous ones by dedicating a thread to do `sigwait()` for them. In the `sigwaiter` thread, once the signal is received, it can be processed.
- The signals of interest must be blocked in the main or initial thread of the process.
- If a new process is created via `fork()`, ensure that the required signals are blocked in the initial thread of the child process if asynchronous signal handling is required. Use `sigwait()` in a dedicated thread in the child process.
- If signal handler-to-thread synchronization is required, synchronize in the `sigwaiter` thread after `sigwait()` returns. Signal-to-thread synchronization is mapped to thread-to-`sigwaiter` thread synchronization.
- Do not use any of the `pthreads` APIs from signal handlers installed through `sigaction()` or `signal()`. It is not safe to do locking from signal handlers installed this way. This alone is a good reason why multithreaded programs must not use signal handlers to modify or set global data that threads also modify.
- If your program needs to ensure atomicity with signals, use `sigwait()`. Capture asynchronous signals, then try to acquire locks as needed within a `sigwaiter` thread. It is not possible to wait for synchronous signals, and blocking (or masking) synchronous signals is probably unwise.
- In general, it is not a good idea to have a `sigwaiter` thread for a signal and to install a signal handler through `sigaction()` or `signal()` for the same signal. POSIX.1c leaves the behavior undefined in this case, and programs depending on a certain behavior in this case are not portable.
- If you replace an asynchronous handler, ensure that there is no conflict with any handler installed previously for the same signal.

Signal Safety in Multithreaded Processes

In multithreaded processes, the mask specified in `sa_mask` for the `sigaction()` call will block the signal only for the current thread. That is because the mask itself can be installed on a per-thread basis. If another signal arrives, the signal handler can be invoked in the context of another thread that has not masked it. For this reason, signal handler-to-signal handler safety cannot be guaranteed by just masking one or more signals. Because of this characteristic, it is difficult to protect signal handlers from other signals.

Thread-to-signal handler safety also cannot be guaranteed. Even if the current thread masks a

signal while entering a critical section that a signal handler also tries to use, it is possible for another thread that does not block the signal to receive the signal and run the signal handler in its context. You might think that this critical section can be enforced by trying to acquire a lock from the signal handler. However, this is not supported in a threaded environment. Signal handlers cannot invoke any POSIX threads locking functions, such as `pthread_mutex_lock()` and `pthread_mutex_trylock()`, because these functions are not required to be signal safe. In fact, they are usually not signal safe in most implementations.

The pthread Locks and Signal Handlers

The `pthread_mutex_lock()` is not signal safe, because when a call to this function results in blocking, the calling thread is the one that gets blocked. Imagine a thread that has locked a mutex and has a signal unmasked. Let the signal be delivered while the thread holds the lock. If the signal handler also tries to acquire the same lock, the call has to be blocked. Since the only way to get blocked is to block the calling thread, the current thread gets into a blocked state waiting for the mutex to be released. However, because the mutex is already locked by the current thread, it can never be released, resulting in a deadlock. This type of deadlock is known as self-deadlock.

It is clear that it is not possible to safeguard a critical section from asynchronous signals. Masking signals in the current thread alone is not sufficient in a multithreaded program. You will have to mask signals in all the threads, then ensure that a signal is selectively unmasked in one specific thread. Thus, a signal can invoke critical sections. In most applications, this might be difficult (if not impossible) to implement, since a thread does not control the signal mask of another thread. Solving this problem requires a method of directing asynchronous signals to a dedicated thread that waits for them. When a signal is received, it can try to acquire a lock and then invoke the critical section. POSIX.1c provides the `sigwait()` API for this purpose.

DCE Signals

The services provided in DCE, such as the DCE Remote Procedure Calls (RPCs), use POSIX thread interfaces extensively. Because DCE is based on draft 4 of POSIX.1c, a compatibility library provides the necessary function (wrappers to map the draft 4 APIs to the draft 7 APIs, except support) for DCE and its applications. The compatibility library does not support the draft 4 signal behavior in AIX 4.1.

DCE's signal model, based on POSIX draft 4, specified the following signal behavior for the AIX DCE 1.1:

- Per-process signal handlers for asynchronous signals
- Per-thread signal handlers for synchronous signals
- Per-process signal mask (`sigprocmask()`)
- Wait for asynchronous signals (`int sigwait(sigset_t *set)`—based on draft 4)

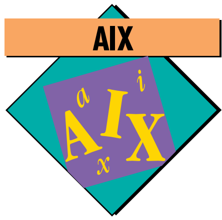
It is possible to implement signal handlers, but the per-process signal mask is difficult to implement without kernel support. Even if draft 4 semantics can be implemented in the compatibility library, conflicts in the signal model will occur when libraries written to draft 7 and draft 4 are used in the same process. To eliminate these problems, we have decided to base DCE signal handling on the draft 7 signal model: per-process signal handlers and per-thread signal masks.

All DCE application developers should evaluate the signal needs and make suitable modifications to applications based on the above guidelines for POSIX.1c draft 7. This generally involves blocking the `sigwaited` signals in the main or initial thread, and not much more. If applications used the global mask to protect critical sections, the same functionality can be achieved with a dedicated `sigwaiter` thread and by acquiring a mutex after the call to `sigwait()` returns with the signal.

The services provided in DCE , such as Remote Procedure Calls, use POSIX thread interfaces extensively.

Chary Tamirisa, IBM Corporation, LAN Systems Division, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com. Since 1993, Mr. Tamirisa has been the team leader for the threads package on AIX and OS/2 DCE. He has also worked in the fields of communication protocols, system software, and National Language Support. Mr. Tamirisa has an MS in Computer Science from McGill University and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.

Threads Programming in AIX Version 4



By Marc Miller

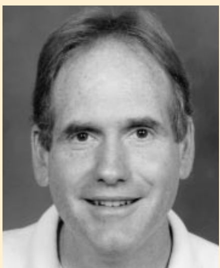
AIX Version 4 provides developers with the power and flexibility of threads programming, an ideal tool for implementing concurrent processing on multiprocessor hardware. This article describes the various threads implementations in AIX, the basic concepts of threads programming, and the differences between developing with threads and with processes. It also describes some basic calls to help developers start creating and managing threads.

Now that multiprocessor hardware is becoming more common and application developers require concurrent processing, many systems require multiple threads of control. To exploit new hardware and software technology while maintaining compatibility with existing systems, the operating system requires new features that help developers split application processing onto multiple processors. Threads is one feature that AIX provides developers to exploit the powers of parallel processing.

Threads

Threads allows developers to partition applications easily. It is an Application Programming Interface (API) defined by POSIX. The threads API in AIX is based on the `pthreads` interface specified in POSIX 1003.4A draft 4 and 7. AIX Version 3 supports threads in the Distributed Computing Environment (DCE) License Program Product (LPP), an add-on product. It is based on the POSIX draft 4, whereas AIX Version 4 supports a draft 7 version of the POSIX specification. Developers can use any of these programming APIs to create and manage multithreaded processes.

Threads has a number of advantages, including the following.


Marc Miller

- **Concurrency**: Developers can use threads to implement parallel processing in an application for performance improvements.
- **Overlapping I/O:** Threads provides easier tools for managing asynchronous I/O, which speeds up application performance.
- **Simple Programming Model:** Threads is one of the easiest programming models to learn because of its similarity to C programming.

Single Versus Multithreaded Processes

In a single-thread process, the process is controlled by one stream of instructions at a time. In a multithread process, the process starts with one stream of instructions and creates other instruction streams called *threads* to do tasks. This is similar to one process forking another process. The main difference is that when a process forks off a child process, a hierarchical relationship exists between the parent and child processes. Within a multithreaded process, all threads are peers with no dependency on a parent process. For example, any thread can kill another thread within the same process.

AIX provides many facilities such as thread creation, termination, synchronization, communication, error recovery, and management to help the developer control how threads behave within a process. These facilities help developers write parallel applications.

Just as a simple process can make system calls, so can a thread when it is running in user space. To make system calls, a thread must be able to switch between user and kernel space. In a multithreaded process, a user thread runs in user space, but it is attached to a kernel thread that runs in kernel space. The kernel thread is managed by the scheduler and handles the user thread's kernel requirements. A kernel thread can be attached to a user thread to do user work or it

can be unattached and running as a kernel thread (similar to a kernel process or kernel extension in AIX Version 3).

The appearance and behavior of kernel threads in the kernel of AIX Version 4 are similar to processes in AIX Version 3. The real differences between threads and processes appear in the user space.

Multithreaded Processes

Figure 1 depicts a typical multithreaded process. Notice that the threads share one version of process-related kernel data, but each thread has its own copy of the registers, some data related to the kernel thread, and a private stack. Therefore, data can be passed via global variables.

The shared process-related kernel data is stored in the proc and user structures. In AIX Version 4 these control blocks have been split into process- and thread-specific structures. The user and proc structures now contain only data that is maintained at the process level. A new thread structure now contains the thread-specific data that used to be in the proc structure. In addition, a uthread structure contains thread-specific data that used to be in the user structure. Figure 2 shows these new control blocks and some sample fields.

In AIX Version 3, the kernel maintained data and scheduled and managed processes. In AIX Version 4, all of these functions must be performed on both processes and threads. Because the thread is now the unit of work, the scheduler needs to manage each kernel thread individually. Some functions, however, still operate on the process level, and these functions affect all threads within a process. For example, setting the `nice` value for a process affects all the threads in the process.

Threads Models

AIX has three basic threads models: M:1 DCE threads, which are implemented in AIX Version 3; 1:1 kernel threads, implemented in AIX 4.1; and M:N threads, which will be available in a future release of AIX.

M:1/DCE Thread Architecture

In AIX Version 3, threads is supported through the DCE `pthreads` LPP, which provides a pure user space implementation of threads. Developers can create and manage multiple threads within a process without any connection to the kernel. The `pthreads` library dispatches and manages all

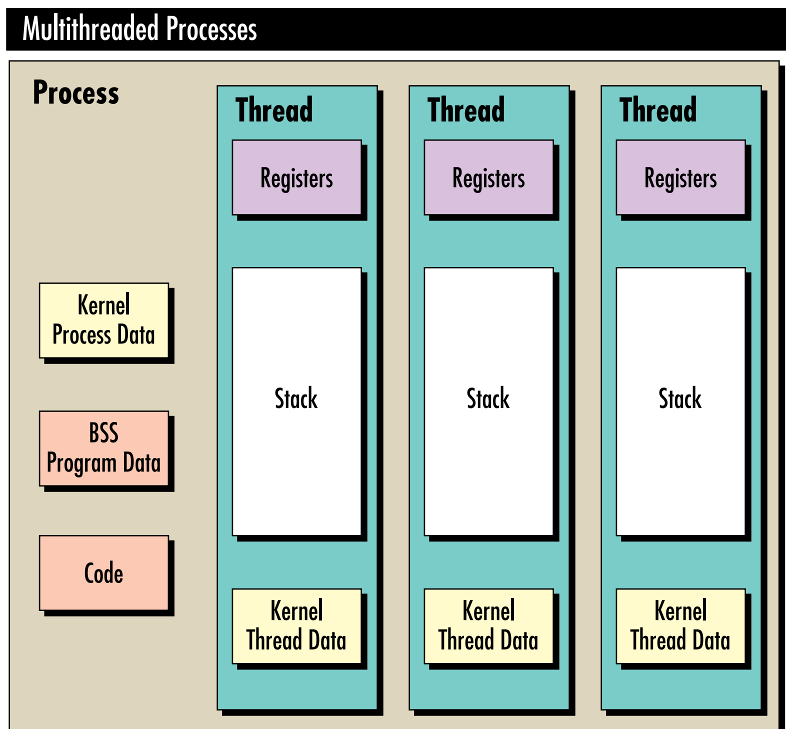


Figure 1. Multithreaded processes

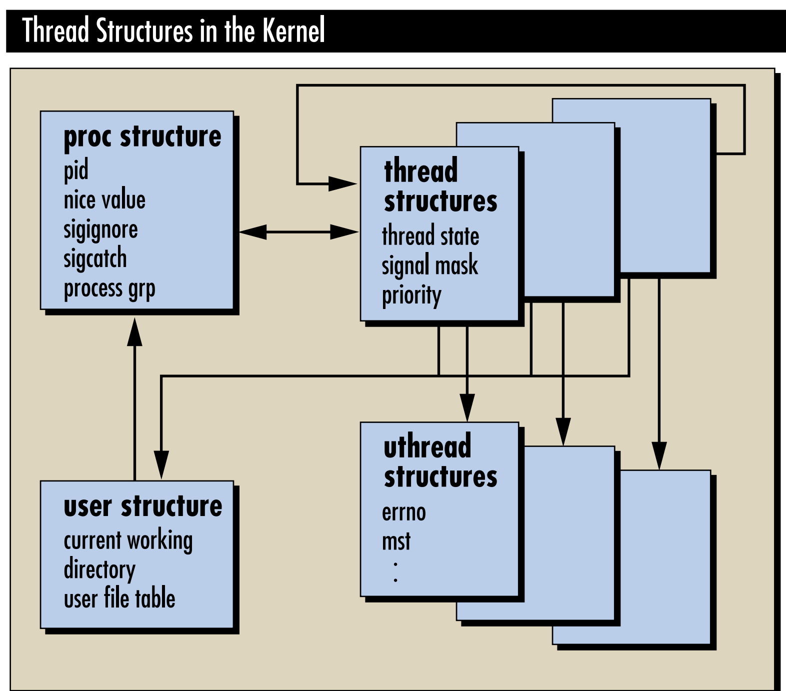


Figure 2. Thread structures in the kernel

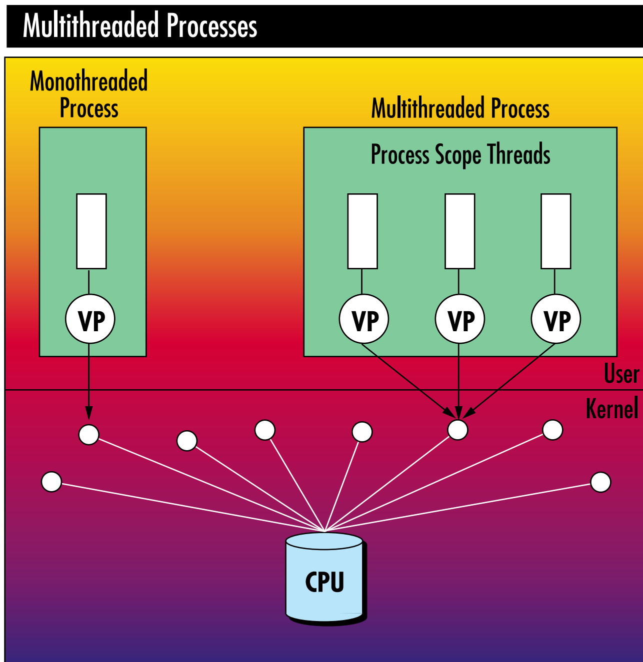


Figure 3. Multithreaded processes

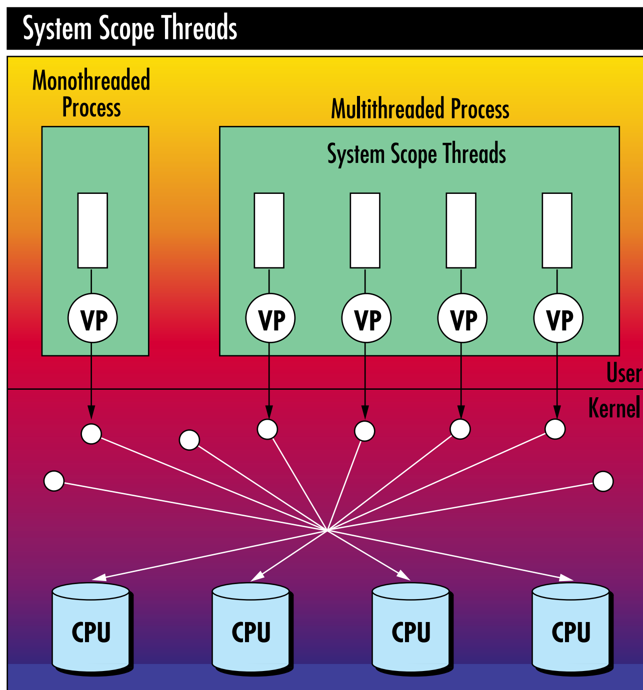


Figure 4. System scope threads

threads using System V signals. These kernel-independent threads are called *process scope* threads (because the scope of the thread does not extend beyond the process).

This model, shown in Figure 3, requires no changes to the kernel for threading to occur—which is a real advantage. Since all of the threads dispatching is handled in the user space, overhead is very low. There is one disadvantage, however. Since the threads dispatching requires the creative use of signals, the time slices are very large—approximately .1 second. These time slices may create problems in time-sensitive applications such as X-Windows applications.

The 1:1 Thread Architecture

AIX 4.1 implements the 1:1 thread architecture. In this model, every user space thread has a corresponding kernel space thread to support it. A virtual processor binds the user space thread to the kernel space thread. The one-to-one correspondence between user and kernel threads enables the kernel to know about each user thread. These permanently bound user threads are called *system scope* threads. All system scope threads contend with each other for resources, just as processes do in AIX Version 3 (see Figure 4).

The advantage of the 1:1 thread architecture is that each thread gets real-time scheduling. And in a symmetric multiprocessing environment, each thread can be run on a different processor to improve performance.

M:N Thread Architecture

The kernel and libraries in future releases of AIX Version 4 will support an M:N implementation, which combines the advantages of the M:1/DCE and 1:1 threading models. In this implementation, developers can use both system scope threads as well as user threads that are not permanently attached to kernel threads, as shown in Figure 5. Multiple user threads can be scheduled and multiplexed onto a smaller number of kernel threads. Since thread management is handled in the user space, overhead is reduced.

The advantage of this model is that developers can balance and control the application environment by using the type of thread ideally suited for the task. For example, developers can use system scope threads for foreground events that require real-time scheduling on the processor and process scope threads for background tasks that do not require immediate system resources.

Programming with Threads

Threads programming is similar to other types of parallel programming in that developers must create and manage the various tasks. However,

threads gives developers a higher level of control over the programming environment than processes do. Specifically, developers can control how threads pass information to each other, specify how a task gets scheduled, and assign a priority level to the task.

The following sections describe a few basic calls that developers can use to begin creating and managing threads. See `libpthreads.a` for more information about these library calls and others.

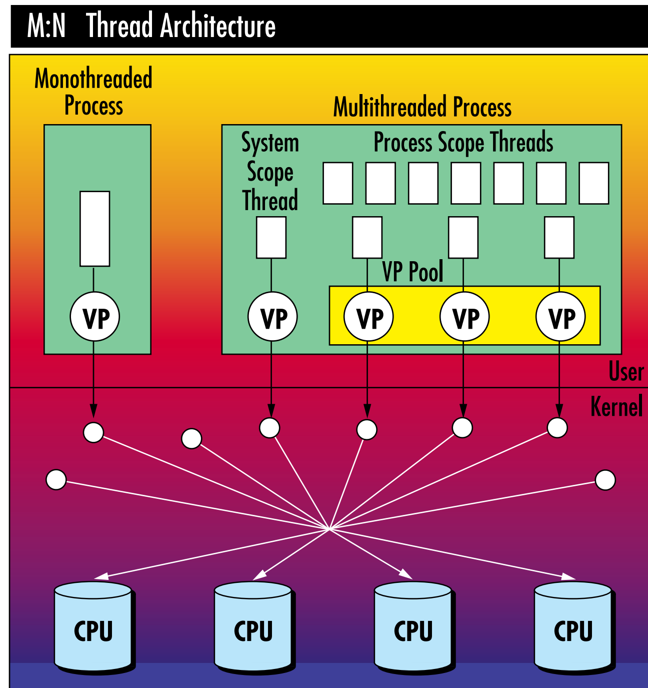


Figure 5. M:N thread architecture

Creating and Managing Threads

Creating a thread is easier and more straightforward than forking a process. Only one single call is required to create a thread, whereas forking a process requires two calls with several possible variations. Use the `pthread_create()` call to create a thread, as shown in Figure 6.

```
int pthread_create (thread, attr,
start_routine, arg)
pthread_t *thread;
const pthread_attr_t *attr;
void *(*start_routine) (void *);
void *arg;
```

Figure 6. Using pthread_create() to create a thread

When a thread is created, a thread ID (TID) is assigned to that thread. TIDs are similar, if not identical, to process IDs. The `pthread_create()` call returns the TID value at thread creation time. The developer can assign characteristics such as stack size, priority, and scheduling algorithm to a thread when it is created by creating and passing a thread attribute block to `pthread_create()`. If the developer specifies NULL, then the thread will be assigned the default behavior. The developer can also pass a `start_routine`, which is simply a pointer to the subroutine the thread should execute, and a pointer to the arguments to be passed to that routine.

Killing Threads

Since all threads are peer processes, any thread can kill any other thread. The `pthread_kill()` call, which is used to kill a thread, behaves very much like the `kill` system call.

```
int pthread_kill (thread, signal)
pthread_t thread;
int signal;
```

To make a thread terminate itself, use `pthread_exit()` with the status code. Remember that `pthread_exit()` terminates only the current thread, whereas `exit()` terminates the entire process, not just the currently executing thread.

```
void pthread_exit (status)
void *status;
```

Any other thread in the process can retrieve the status code from a terminated thread if it knows the terminated thread's TID. Use the `pthread_join()` routine to retrieve the terminated thread's status code:

```
int pthread_join (thread, status)
pthread_t thread;
void **status;.
```

If `pthread_join()` is called with a TID for a thread that is still running, `pthread_join()` blocks until the thread has terminated. The `pthread_join` call provides nearly the same function for threads that `wait()` provides for processes. The system does not free a thread's storage until `pthread_join` is called for that thread. Another difference between processes and threads is that when a process terminates, the parent process must call `wait()` to free the resources; but with threads, any sibling thread can perform this task.

Threads Scheduling

In AIX Version 4, the thread is now the unit of work that the scheduler dispatches, so each thread can have a different scheduling priority. Threads also allows developers to specify the scheduling policy for each thread, providing a higher level of control over scheduling than processes do. There are three scheduling policies:

- **SCHED_FIFO**: Denotes fixed-priority first-in first-out (non-preemptive) scheduling
- **SCHED_RR:** Denotes fixed-priority round-robin (preemptive) scheduling
- **SCHED_OTHER**: Denotes the default AIX scheduling policy; this is the default value

Note that `SCHED_RR` is nearly identical to fixed-priority scheduling for processes on AIX Version 3. However, `SCHED_FIFO` now gives developers the choice of using a non-preemptive scheduling algorithm. This can be a valuable but dangerous tool. Both methods require root authority to run and should not be mixed on the same priority level.

Threads Communications and Synchronization

Two different mechanisms can be used to pass communication between threads: mutexes and condition variables.

A *mutex* is a mutual exclusion lock that protects data or other resources from concurrent access. When one thread holds the lock, no other threads can access the locked resource. When a thread tries to acquire a lock already held by another lock, it has two choices: to wait for the lock to be unlocked, or to return with a return code indicating that the lock is held by another thread. Use the `thread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` routines to handle these basic locking tasks.

Condition variables are powerful but complex tools. Condition variables allow threads to wait until some event or condition has occurred. A condition consists of three parts: a variable, usually boolean, that indicates a condition that can be tested; a mutex, which is used to serialize access to the variable; and a condition variable that causes a thread to wait until the condition is met. Although condition variables require more effort for the developer, they can be quite powerful.

To learn more about condition variables, read the section in InfoExplorer™ on "Using Condition Variables."

Conclusion

Threads programming in the UNIX environment is still relatively new, so the design, development, and debugging tools are still evolving. Nevertheless, the power and flexibility of threads demand a developer's consideration.

Marc Miller, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Miller works in the Austin AIX Executive Briefing Center, where he delivers presentations on future client/server directions, AIX, and DCE. He has a BS in Computer Science from Northwestern University.

Free AIX Hints and Tips

For free hints and tips about AIX, just call 800-IBM-4FAX from a touch-tone phone and select from 120+ items, including:

- AIX 3.2.5 Installation Tips (#2505)
- Replacing a Fixed Disk (#1895)
- Disk Quota System Setup and Use (#2929)
- Installing Nodelock & Floating Licenses (#1400)
- Reducing a File System in AIX 3.2 (#1746)