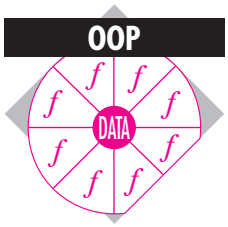
Object-Oriented Programming with SOM/6000



By Debora Blakely-Fogel

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

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

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

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

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

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

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

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

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

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

Debora Blakely-Fogel

Data Encapsulation

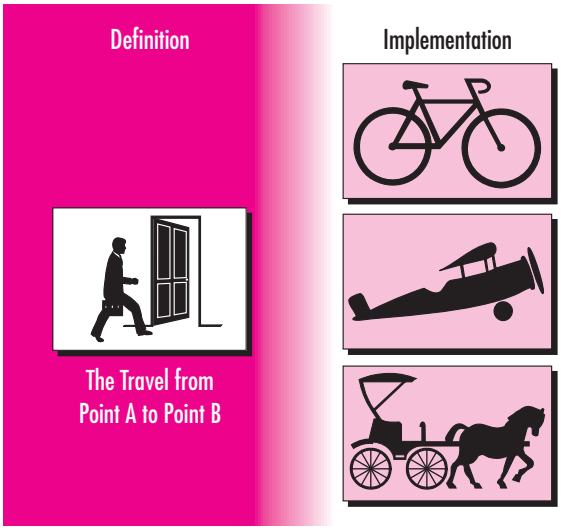


Figure 1. Data encapsulation—the key to object-oriented programming

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

Frameworks

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

DSOM Framework

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

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

Interface Repository Framework

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

Persistence Framework

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

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

Inheritance

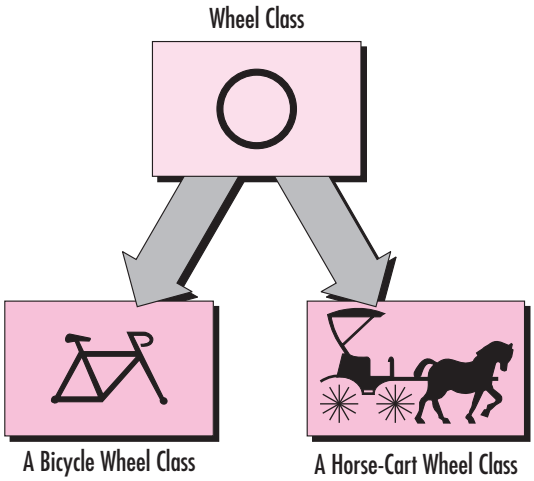


Figure 2. Class inheritance enables programmers to reuse existing functionality

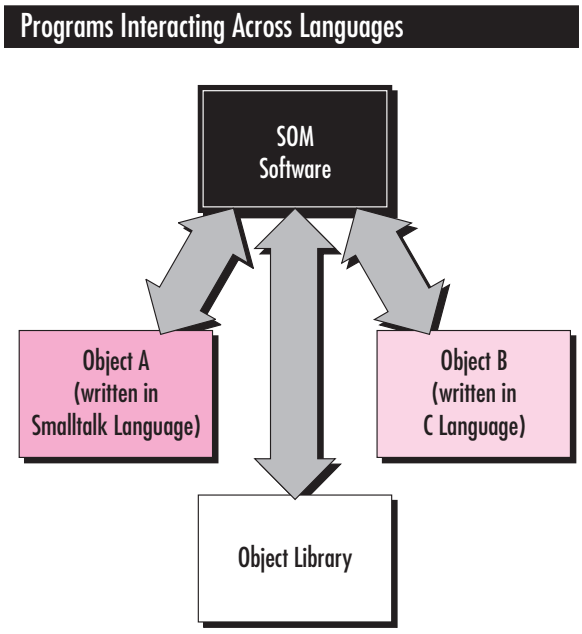


Figure 3. SOM enables programs to interact across language barriers

Replication Framework

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

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

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

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

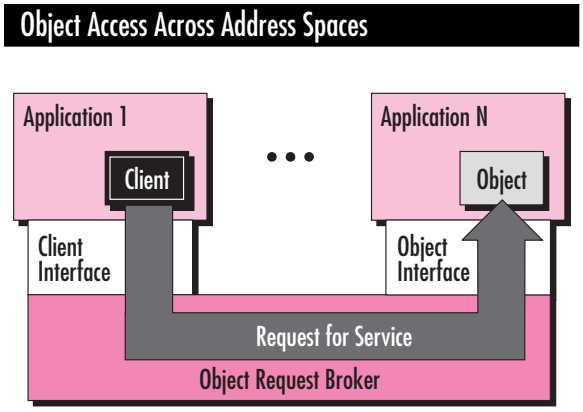


Figure 4. DSOM enables object access across address spaces

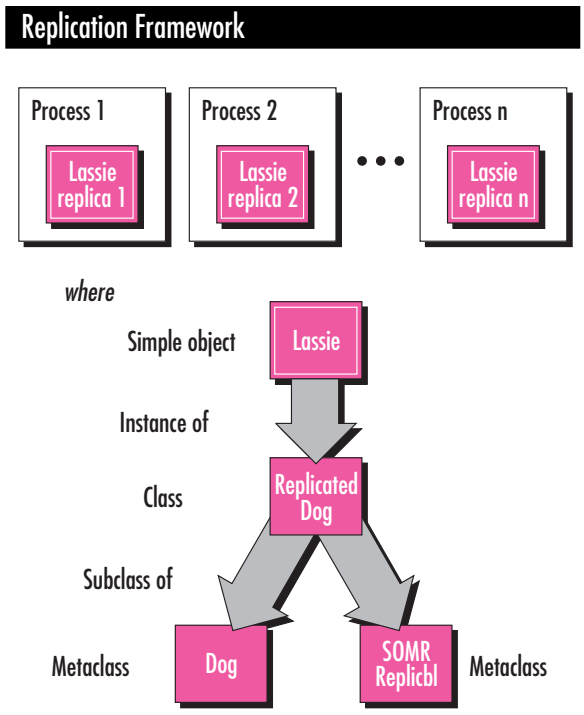


Figure 5. Replication framework

SOM Compiler Translates IDL File

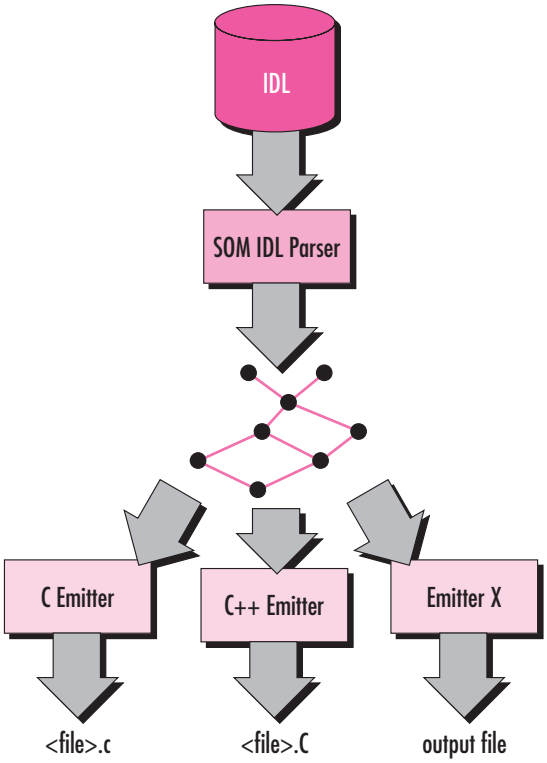


Figure 6. SOM compiler translates IDL files to required output

Emitter Framework

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

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

Event Management Framework

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

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

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

Implementing SOM Classes

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

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

The Implementation Description and Template

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

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

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

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

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

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

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

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

Figure 7. Interface definition file

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

Figure 8. Implementation file

before the parameters declared in the IDL specification:

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

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

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

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

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

Extending the Implementation Template

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

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

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

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

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

- Hello_parent_Printer_somInit
- Hello_parent_File_somInit
- Hello_parents_somInit

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

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

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

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

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

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

Using SOM Classes

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

Declaring a SOM Variable

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

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

Creating Instances of a SOM Variable

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

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

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

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

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

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

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

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

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

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

Figure 9. Object variable declaration and class instantiation

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

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

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

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

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

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

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

Invoking Methods on Objects

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

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

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

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

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

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

Obtaining a Method's Procedure Pointer

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

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

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

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

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

`className`: Name of the class that introduces the method

`methodName`: Name of the desired method

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

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

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

Using Class Objects

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

Getting the Class of an Object

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

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

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

Creating a Class Object

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

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

Vendors That Have Announced Plans to Support SOM

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

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

Implementing SOM Classes:

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

Using SOM Classes:

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

Figure 12. Steps for programming with SOM

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

Referring to Class Objects

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

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

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

Summary

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

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

IBM World Wide Web Servers Online

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

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

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

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

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

Services and Support
POWER Developer
Technical Library
AIXpert