To appear in *First Class*, a bimonthly publication of the Object Management Group, October 1994:

IBM's System Object Model (SOM):

Making Reuse a Reality

IBM Corporation
Object Technology Products Group
Austin, Texas

It is probably not too difficult to convince a reader of *First Class* that object technology holds the enormous potential to revolutionize the software industry. Realizing that potential, however, is a much more arduous task. The object industry today is a patchwork of islands of information residing within the confines of a myriad of incompatible object systems. As a particularly ironic example, consider binary C++ class libraries that cannot be shared among developers using different C++ compilers let alone by Smalltalk or Cobol programmers. In order to address some of the key inhibitors to the widespread acceptance of object technology, IBM created the System Object Model (SOM). This article describes SOM and how it attempts to make reuse a reality.

What is SOM?

SOM is object enabling technology that was designed specifically to overcome several major obstacles to the pervasive use of object class libraries. The general problem can be stated in the following way:

What is required to enable the development of "system objects" intended to be supplied either as part of an operating system, a vendor tools product, or an application that:

a) Can be distributed and subclassed in binary form. Class library developers would not need to supply source code to allow users to subclass their objects.

b) Can be used, including full subclassing, across languages. It should be possible to implement an object using one language, subclass the object using another language and use that class to build an application in yet a third language. Developers want to modify and build applications from class libraries in their preferred language not necessarily the one in which the classes were originally written.

c) Provides for the subsequent modification (fixes or enhancements) of these components without having to recompile preexisting clients that use them (upward binary compatibility). This is a key requirement because applications dependent upon system libraries cannot be rebuilt each time a change is made to a component in the library.

In order to solve these problems, the developers of SOM designed an advanced object model and implemented the object-oriented runtime engine necessary to support this model. SOM supports all of the concepts and mechanisms normally associated with object-oriented systems including inheritance, encapsulation and polymorphism. In addition, SOM possesses a number of advanced object mechanisms including support for metaclasses, three types of method dispatch with both static and dynamic method resolution, dynamic class creation and user intercept of method dispatch.

SOM can be used to provide object-oriented mechanisms for procedural languages (such as C or Cobol) or used in conjunction with the capabilities of object-oriented programming languages like C++ or SmallTalk. In fact, several C++ vendors (including IBM) are currently incorporating SOM into their language runtimes. SOM currently supports C and C++ with Cobol and SmallTalk support available this year from third party ISVs.

SOM has been commercially available in IBM's product line since 1991 when it first appeared in OS/2 2.0. In addition to OS/2, it is now available for AIX, Windows and Mac System 7. Over the next two years SOM is likely to appear on other Unix platforms and Novell's Netware, as well as IBM's Workplace, MVS and OS/400 operating systems. SOM has been selected by the Component Integration Laboratories as the underlying object model and runtime engine for the OpenDoc compound application technology.

SOM and DSOM

With SOM, IBM is striving to achieve many of the same objectives as the Object Management Group: To facilitate the interoperation of objects independent of where they are located, the programming language in which they are implemented, or the operating system or hardware architecture on which they are running. One way to view SOM is as a highly optimized, single address space, object request broker that provides interlanguage interoperability and supports binary subclassing and upward binary compatibility. Using SOM, objects implemented in different languages can be combined in the same address space.

SOM is compliant with the OMG's CORBA specification. What does this mean? It means that SOM classes are defined using the CORBA Interface Definition Language. It means that SOM supports all of the CORBA datatypes. It means that the C language bindings for SOM classes are CORBA-compliant (CORBA does not yet have standard bindings for C++, Smalltalk and Cobol). It means that SOM provides an interface repository supporting the CORBA functionality and programming interfaces.

In order to address the cross-process, cross-machine, cross-operating system, cross-architecture interoperability problem, SOM relies on its distributed object framework, sometimes called DSOM (for distributed SOM). DSOM is a set of SOM classes (shipped with the SOMobjects toolkit) that seamlessly extends the method dispatch mechanisms embodied in the SOM runtime engine to allow methods to be invoked, in a programmer transparent way, on objects in a different address space or on a different machine from the caller. DSOM is fully CORBA-compliant, supporting all CORBA data types, functionality and programming interfaces. Currently, fully interoperable versions of the DSOM framework are available for SOM on AIX, OS/2 and Windows.

How does SOM work?

Language-independence

SOM achieves cross-language interoperability by building its method dispatch mechanism based on system-defined procedure linkage conventions. This means that SOM follows the register and stack utilization conventions defined by an operating system for all programs, regardless of their implementation language. System linkage conventions also dictate how return values are passed from the callee back to the caller. By using the system linkage protocol, SOM can dispatch methods independent of the language in which the executable code was written. As a result, virtually any language that supports the system procedure call linkage conventions can use a SOM class or can be used to implement a SOM class.

Upward binary compatibility

The key to SOM's binary magic is the complete encapsulation of the implementation details of a class. A client binding to a SOM class has no information about the size and entry points to that class compiled into its executable. Method dispatch and access to instance data is effected through a set of data structures which are computed during the construction and initialization of a class. Two of the most important SOM data structures are the ClassData structure and the SOM method table. Because these structures are completely computed at runtime, the SOM class can be modified (such as refactoring the class hierarchy, moving methods up the hierarchy, adding methods or instance data) without requiring recompilation of the client code. In addition, the SOM data structures can be manipulated by the programmer at runtime giving the class implementor enormous flexibility in enhancing or controlling method dispatch.

By completely encapsulating the implementation of an object, SOM overcomes what has been referred to as the "fragile base class" problem - the inability to modify a class without recompiling clients and derived classes dependent upon that class.

Conclusion

Hopefully, this article has given the reader a flavor for how IBM's System Object Model addresses some of the key impediments to object interoperability. SOM is the linchpin of IBM's object enabling infrastructure. This infrastructure will eventually underlay all of IBM's object technology product offerings including OpenDoc, the Taligent frameworks and the Workplace family of operating systems.