The System Object Model (SOM) and the Component Object Model (COM):

A comparison of technologies from a developer's perspective

IBM Corporation
Object Technology Products Group
Austin, Texas

Recently, both IBM and Microsoft have been wooing developers with their object strategies, and both claim to have solved the problems necessary to permit the construction of commercial grade software components or objects. But the solutions offered are quite different. This white paper compares and contrasts IBM's System Object Model (SOM) with Microsoft's Component Object Model (COM) to assess the implications for developers building software components using these technologies.

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 no longer 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 build an object using one language, subclass the object using another language and use that class to build an application in yet a third language. Users of class libraries want to modify and build applications from these classes in their preferred language not necessarily the one in which the class was originally written.

c) Provides for the subsequent updating (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 provides all of the concepts and mechanisms normally associated with object-oriented systems including inheritance, encapsulation and polymorphism.

SOM can be used to provide object-oriented mechanisms for procedural languages or used in conjunction with the capabilities of object-oriented programming languages like C++ or SmallTalk. In fact, several C++ vendors are currently incorporating SOM into their language runtimes.

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 be appear on other Unix platforms and Novell's Netware, as well as IBM's Workplace, MVS and OS/400 operating systems.

SOM and DSOM

Unfortunately, there has been a great deal of confusion about the relationship between SOM and DSOM. SOM is packaging technology and runtime support for building language independent class libraries. It embodies an advanced object model with a complete runtime implementation.

DSOM is a set of SOM classes (shipped with the SOMobjects Toolkit) that extends the method dispatch mechanism embodied in the SOM runtime engine to allow methods to be invoked, in a programmer transparent way, on SOM objects that exist in a different address space from the caller. The DSOM class library is compliant with the Object Management Group's Common Object Request Broker Architecture (CORBA) specification. DSOM is a framework built using the SOM technology that allows developers to construct distributed object applications. DSOM does not implement a separate object model or runtime since DSOM is built with the SOM object model and runtime.

SOM and OpenDoc

Another source of confusion, intentional or otherwise, when comparing IBM and Microsoft technologies has been the relationship of the OpenDoc compound document technology to SOM and DSOM. SOM is object enabling technology, it was never intended to provide compound document functionality. OpenDoc, which is developed and distributed by the Component Integration Laboratory, is built upon the SOM object model and runtime and the DSOM framework and provides a framework specifically designed for building components or "parts" that can be integrated into compound documents.

What is COM?

The Component Object Model (COM) purports to be Microsoft's answer to SOM. Microsoft refers to COM as a "language-neutral" binary interface specification for "Windows Objects" and a set of run time functions for instantiating them. Windows Objects are different from the objects in an object oriented programming language in some important ways that will be examined shortly.

Interestingly, COM provides no formal representation of an object at all, which in itself is a pretty glaring omission for an "object model". Instead a COM "object" or "Windows Object" exists only in a conceptual sense. The programmer is responsible for providing the representation of the Windows Object - which is arbitrary as long as it obeys the rules for how COM expects its objects to behave.

In addition, since programs using COM obtain and manipulate only "interface" references and not objects at all, it is tempting to conclude that COM is a misnomer and reflects Microsoft's attempt to redefine widely used (and understood) terms to suit the technology it has available.

COM has been commercially available on Microsoft's 16-bit Windows platform since Spring '93 and is currently available in beta form for Windows NT and Chicago. Microsoft promises that COM will also be available on Mac System 7 and other platforms sometime in the future.

COM provides very little in the way of runtime support for programmers building Windows objects. Essentially, COM is a set of rules programmers must interpret and follow in order to build these components.

Distributed COM

Although frequently discussed, COM currently lacks the cross-machine distributed object capabilities of SOM. Microsoft plans to introduce distributed COM in their Cairo operating system slated for availability in beta form sometime in late 1995.

COM and OLE

Microsoft's linking and embedding technology OLE 2.0 is built using COM. However, Microsoft does not go to much effort to distinguish COM from OLE 2.0 (the two are bundled together). The major reason for this is that OLE currently represents the only concrete instantiation of COM, which is almost entirely a set of rules for programmers to follow. This creates confusion for developers of Windows objects who have difficulty separating the underlying object model from its manifestation in the OLE implementation. It is like telling someone how to build an engine by giving them a car and neglecting to tell them what portion is the engine and what pieces constitute the rest of the vehicle.

A comparison of technologies

The remainder of this white paper contrasts various aspects of SOM with COM. The major areas of comparison center on the level of implementation support provided to developers building software using these technologies and to what degree these technologies actually support an object-oriented paradigm.

What is an "object model" - specification, implementation or both?

One of the most important differences between SOM and COM is the amount of support provided to developers building components using these technologies. In examining the runtime support provided in SOM and COM it is very clear that IBM and Microsoft have very different interpretations of what the term "object model" means.

Comparing SOM and COM is a bit like comparing the engine of an automobile with its specifications. A car's engine is not a specification; it is the essential piece of the car that generates the vehicle's motion. Similarly, the engine's specification will not impart motion to the car, rather it must first be used to build an engine before a car can be expected to actually move. In this crude analogy, the engine corresponds to IBM's SOM and the specification corresponds to Microsoft's COM.

An immediately observable SOM/COM difference that arises from these disparate uses of the term "object model" is a distinction in the type of code that a developer must write in every application. With SOM, a programmer writes code that uses the object infrastructure SOM provides; with COM the programmer must also write the code that implements many of the rules and guidelines that comprise the COM infrastructure (regardless of whether this code is written manually or can be partially automated via some development tool, it still must appear in every COM program).

Let's examine some specifics.

Support for a binary standard

SOM is a complete implementation of a syntax-free object-oriented run-time engine -- one that has been carefully engineered to have a robust binary interface that completely encapsulates implementation detail. This is underscored by the fact several C++ compiler vendors that are currently using SOM for their run-time library. Object-oriented language compilers that utilize SOM as their run-time are referred to as "Direct-to-SOM" compilers. And because SOM has been designed to support a broad set of OO semantics, other languages (both object-oriented and procedural) can utilize the SOM runtime through intermediary mapping layers referred to as SOM "bindings."

Regardless of which approach is utilized (direct-to-SOM or language bindings), the advantage to a developer is that class libraries can be built which sport robust binary interfaces. Client programs may be constructed that are derived from the classes in the library using normal object oriented inheritance techniques without compromising the ability of the class library implementor to make evolutionary changes in the library's internals, and without requiring all client programmers to use the same development language. In short, SOM objects are similar to the normal objects in an object-oriented programming language (OOPL), except that their binary interfaces have been made more robust and replaced with language-neutral mechanisms.

Microsoft's COM, on the other hand, while equally effective at hiding an object's implementation details, does not attempt to be a run-time engine for object oriented programming. In fact, Microsoft questions the appropriateness of today's object oriented programming languages for exposing the interfaces of an interoperable software component. The COM specification is a way of hiding the OOPL notion of an object and exposing instead the different abstraction called a Windows Object.

We promised earlier to look at how Windows Objects differ from typical OOPL objects. Object identity and inheritance are two important examples.

Object identity

Windows objects are not accessed in the same way that OOPL objects are. Whereas an OOPL would allow you to designate a particular object with an "object reference" (or a pointer), a programmer never actually obtains a reference to a Windows object. Instead, Windows objects are accessed exclusively through their interface reference (pointers) and one must obtain a separate interface pointer for each of the object's interfaces that a programmer needs to use.

For example, if an object O supports three different interfaces (I1, I2, and I3), you could obtain and use references for O's I1, O's I2, or O's I3, but never obtain a reference for O itself. If you had a reference for O's I1 and O's I2, the only way you could even be sure that both of these referred to the same underlying object, would be to query O's I1 for a reference to I2 and then see if it was the same I2 reference you already had. In general this is always possible because COM requires every interface to support the "IUnknown" protocol. The IUnknown protocol specifies three functions that should appear in every COM interface and the first of these functions should permit you to obtain a pointer to any of an object's other interfaces.

Notice that in speaking of COM we use words like "should." This is because COM is largely a set of rules that are not actually enforced anywhere. When creating a COM Windows Object, it is the programmer's responsibility to implement all of these rules and to get them right. This is yet another difference between SOM and COM. SOM's semantics are implemented by within the SOM run-time, while almost all of COM's semantics must be implemented by the developer in each COM Windows Object. The opportunity for making mistakes when following the complex rules for COM is obvious.

Inheritance

An even bigger difference between a COM Windows Object and the typical definition of an OOPL object is that COM does not support (either by recognizing the concept or providing the mechanism) the notion of inheritance. That is, there is no way for programmers to create their own subclass of a Windows Object class using COM. This is intentional. Microsoft's OLE architects do not believe that implementation inheritance is an appropriate relationship between independently developed software components that are distributed in binary form. This conclusion is largely based on the failure of standard object oriented programming languages to properly encapsulate the implementation detail of a base class from the subclasses that may be derived from it. Such information may include the size and internal structure of the base class from which the subclass is derived. Microsoft calls this the "fragile" base class problem.

Instead of offering a solution to the fragile base class problem Microsoft chose instead to constrain the semantics of their object model to preclude a mechanism that is widely regarded as one of the defining characteristics of object orientation. In 1987, Peter Wegner of Brown University introduced some order to the OO community by suggesting a subsequently well-accepted taxonomy for classifying object systems and programming languages based on the features and programming paradigms they support. In Wegner's terminology, systems that have classes and support implementation inheritance can be properly called "object-oriented," while those without implementation inheritance are characterized as "object-based." This offers a concise way of summarizing a significant difference between SOM and COM. SOM is "object-oriented" while COM is merely "object-based." Microsoft objects to this because they have improperly used the phrase "object-oriented" heavily in their COM/OLE marketing literature.

Of course, Microsoft's position is much less compelling when one considers that IBM started with a similar premise and chose instead to fix the problem by inventing SOM. The difference in approaches amounts to the fact that SOM permits class libraries to be developed using conventional object-oriented programming paradigms and to offer these same paradigms to their clients, while COM rejects the object-oriented notion of implementation inheritance in favor of totally different paradigms for client programming. Microsoft calls these new paradigms, containment and aggregation and offers them as an alternative approach for object reuse.

Aggregation and containment are essentially manual techniques for code reuse entirely implemented by developer supplied code. COM is not involved in aggregation or containment; it simply provides rules about what users must write.

Containment is used when one wants to change some aspect of the implementation of an object. It requires the programmer to encapsulate the object to be modified with another object whose interface includes the same functions as the encapsulated object. The programmer then supplies the new behavior for functions that are to be changed and provides redispatch stubs to call the corresponding function in the encapsulated object for functions that are not changed.

Aggregation is used when a programmer wishes to add functionality to an object but does not need to change any of the preexisting behavior to the object. Aggregation is really nothing more than an optimized form of containment in which the programmer is not required to write redispatch stubs for each function in the object's interface. However, the ability to support aggregation must be explicitly built into a COM object by the developer, so not all Windows objects can be used in this fashion.

With SOM, IBM has chosen to solve the fragile base class problem as opposed to constraining developers' ability to apply widely accepted object oriented techniques. Instead of redefining the commonly accepted parlance of object technology or making pronouncements on the correctness or incorrectness of prevalent object oriented techniques, SOM supports the familiar paradigm used by developers of object-oriented components. Developers of SOM objects can employ single inheritance, multiple inheritance or abstract (interface only) inheritance.

Summary

In summary, the System Object Model provides a rich, extensible, object model with complete runtime support for its semantics. SOM embodies features commonly associated with object oriented programming systems such as implementation inheritance, encapsulation and polymorphism as well as advanced capabilities including metaclasses, user intercept and control of method dispatch and dynamic class construction. SOM provides these capabilities in a language independent way that solves the fragile base class problem. SOM provides programmers with a familiar object-oriented paradigm for building their applications.

The Component Object Model on the other hand is largely a set of rules for building interoperable software components. COM places the burden of support for the protcol onto the developer who must duplicate this support in every COM object he or she implements. As a direct consequence of the lack of runtime support, implementing COM objects is a highly complex, time consuming process fraught with opportunities for misinterpretation and error. Because COM does not support many commonly accepted object-oriented concepts and mechanisms, developers are forced to adopt an unfamiliar programming paradigm and, in fact, typically have to switch between the paradigm provided by the implementation language (e.g. C++) and the paradigm enforced by COM. In comparison with SOM, COM embodies an impoverished set of functionality that places many constraints on developers.