SOMobjects Developer Toolkit

An Overview

An overview of IBM's SOMobjects Developer Toolkit — including the System Object Model and its accompanying frameworks

Version 2.0
June 1993

Second Edition (June 1993)

An Overview of the SOMobjects Developer Toolkit

Background

Object-oriented programming (or **OOP**) is an important new programming technology that offers expanded opportunities for software reuse and extensibility. Object-oriented programming shifts the emphasis of software development away from functional decomposition and toward the recognition of units (called *objects*) that encapsulate both code and data. As a result, programs become easier to maintain and enhance. Object-oriented programs are typically more impervious to the "ripple effects" of subsequent design changes than their non-object-oriented counterparts. This, in turn, leads to improvements in programmer productivity.

Despite its promise, penetration of object-oriented technology to major commercial software products has progressed slowly because of certain obstacles. This is particularly true of products that offer only a binary programming interface to their internal object classes *(i.e.,* products that do not allow access to source code).

The first obstacle that developers must confront is the choice of an object-oriented programming language.

So-called "pure" object-oriented languages (such as Smalltalk) presume a complete run-time environment (sometimes known as a virtual machine), because their semantics represent a major departure from traditional, procedure-oriented system architectures. So long as the developer works within the supplied environment, everything works smoothly and consistently. When the need arises to interact with foreign environments, however (for example, to make an external procedure call), the pure-object paradigm ends, and objects must be reduced to data structures for external manipulation. Unfortunately, data structures do not retain the advantages that objects offer with regard to encapsulation and code reuse.

"Hybrid" languages such as C++, on the other hand, require less run-time support, but sometimes result in tight bindings between programs that implement objects (called "class libraries") and their clients (the programs that use them). That is, implementation detail is often unavoidably compiled into the client programs. Tight binding between class libraries and their clients means that client programs often must be recompiled whenever simple changes are made in the library. Furthermore, no binary standard exists for C++ objects, so the C++ class libraries produced by one C++ compiler cannot (in general) be used from C++ programs built with a different C++ compiler.

The second obstacle developers of object-oriented software must confront is that, because different object-oriented languages and toolkits embrace incompatible models of what objects are and how they work, software developed using a particular language or toolkit is naturally limited in scope. Classes implemented in one language cannot be readily used from another. A C++ programmer, for example, cannot easily use classes developed in Smalltalk, nor can a Smalltalk programmer make effective use of C++ classes. Object-oriented language and toolkit boundaries become, in effect, barriers to interoperability.

Ironically, no such barrier exists for ordinary procedure libraries. Software developers routinely construct procedure libraries that can be shared across a variety of languages, by adhering to standard linkage conventions. Object-oriented class libraries are inherently different in that no binary standards or conventions exist to derive a new class from an existing one, or even to invoke a method in a standard way. Procedure libraries also enjoy the benefit that their implementations can be freely changed without requiring client programs to be recompiled, unlike the situation for C++ class libraries.

For developers who need to provide binary class libraries, these are serious obstacles. In an era of open systems and heterogeneous networking, a single-language solution is frequently not broad enough. Certainly, mandating a specific compiler from a specific vendor in order to use a class library might be grounds not to include the class library with an operating system or other general-purpose product.

The **System Object Model (SOM)** is IBM's solution to these problems.

Introducing SOM and the SOMobjects Toolkit

The System Object Model (SOM) is a new object-oriented programming technology for building, packaging, and manipulating binary *class libraries*.

- With SOM, class implementors describe the *interface* for a class of objects (names of the methods it supports, the return types, parameter types, and so forth) in a standard language called the **Interface Definition Language**, or **IDL**.
- They then *implement* methods in their preferred programming language (which may be either an object-oriented programming language or a procedural language such as C).

This means that programmers can begin using SOM quickly, and also extends the advantages of OOP to programmers who use non-object-oriented programming languages.

A principal benefit of using SOM is that SOM accommodates changes in implementation details and even in certain facets of a class's interface, without breaking the binary interface to a class library and without requiring recompilation of client programs. As a rule of thumb, <u>if changes to a SOM class do not require source-code changes in client programs, then those client programs will not need to be recompiled</u>. This is not true of many object-oriented languages, and it is one of the chief benefits of using SOM. For instance, SOM classes can undergo structural changes such as the following, yet retain full backward, binary compatibility:

- Adding new methods,
- Changing the size of an object by adding or deleting instance variables,
- Inserting new parent (base) classes above a class in the inheritance hierarchy, and
- Relocating methods upward in the class hierarchy.

In short, implementors can make the typical kinds of changes to an implementation and its interfaces that evolving software systems experience over time.

Unlike the object models found in formal object-oriented programming languages, SOM is language-neutral. It preserves the key OOP characteristics of encapsulation, inheritance, and polymorphism, without requiring that the user of a SOM class and the implementor of a SOM class use the same programming language. SOM is said to be *language-neutral* for four reasons:

1. All SOM interactions consist of <u>standard procedure calls</u>. On systems that have a standard linkage convention for system calls, SOM interactions conform to those conventions. Thus, most programming languages that can make external procedure calls can use SOM.
2. The form of the SOM Application Programming Interface, or <u>API</u> (the way that programmers invoke methods, create objects, and so on) <u>can vary widely from language to language</u>, as a benefit of the SOM binding*s*. *Bindings* are a set of macros and procedure calls that make implementing and using SOM classes more convenient by tailoring the interface to a particular programming language.
3. SOM supports several <u>mechanisms for method resolution</u> that can be readily mapped into the semantics of a wide range of object-oriented programming languages. Thus, SOM class libraries can be shared across object-oriented languages that have differing object models. A SOM object can potentially be accessed with three different forms of method resolution:

   - <u>Offset resolution</u>: roughly equivalent to the C++ "virtual function" concept. Offset resolution implies a static scheme for typing objects, with polymorphism based strictly on class derivation. It offers the best performance characteristics for SOM method resolution. Methods accessible through offset resolution are called *static* methods, because they are considered a fixed aspect of an object's interface.

- Name-lookup resolution: similar to that employed by Objective-C and Smalltalk. Name resolution supports untyped (sometimes called "dynamically" typed) access to objects, with polymorphism based on the actual protocols that objects honor. Name resolution offers the opportunity to write code to manipulate objects with little or no awareness of the type or shape of the object when the code is compiled.
- Dispatch-function resolution: a unique feature of SOM that permits method resolution based on arbitrary rules known only in the domain of the receiving object. Languages that require special entry or exit sequences or local objects that represent distributed object domains are good candidates for using dispatch-function resolution. This technique offers the highest degree of encapsulation for the implementation of an object, with some cost in performance.

4. SOM conforms fully with the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standards.† In particular,
   - Interfaces to SOM classes are described in CORBA's Interface Definition Language, IDL, and the entire SOMobjects Toolkit supports all CORBA-defined data types.
   - The SOM bindings for the C language are compatible with the C bindings prescribed by CORBA.
   - All information about the interface to a SOM class is available at run time through a CORBA-defined "Interface Repository."

SOM is not intended to replace existing object-oriented languages. Rather, it is intended to complement them so that application programs written in different programming languages can share common SOM class libraries. For example, SOM can be used with C++ to

- Provide upwardly compatible class libraries, so that when a new version of a SOM class is released, client code needn't be recompiled, so long as no changes to the client's source code are required.
- Allow other language users (and other C++ compiler users) to use SOM classes implemented in C++.
- Allow C++ programs to use SOM classes implemented using other languages.
- Allow other language users to implement SOM classes derived from SOM classes implemented in C++.
- Allow C++ programmers to implement SOM classes derived from SOM classes implemented using other languages.
- Allow encapsulation (implementation hiding) so that SOM class libraries can be shared without exposing private instance variables and methods.
- Allow dynamic (run-time) method resolution in addition to static (compile-time) method resolution (on SOM objects).
- Allow information about classes to be obtained and updated at run time. (C++ classes are compile-time structures that have no properties at run time.)

The SOM Compiler

The **SOMobjects Toolkit** contains a tool, called the **SOM Compiler**, that helps implementors build classes in which *interface* and *implementation* are decoupled. The SOM Compiler reads the IDL definition of a class interface and generates:

- an implementation skeleton for the class,
- bindings for implementors, and
- bindings for client programs.

†OMG is an industry consortium founded to advance the use of object technology in distributed, heterogeneous environments.

Bindings are language-specific macros and procedures that make implementing and using SOM classes more convenient. These bindings offer a convenient interface to SOM that is tailored to a particular programming language. For instance, C programmers can invoke methods in the same way they make ordinary procedure calls. The C++ bindings "wrap" SOM objects as C++ objects, so that C++ programmers can invoke methods on SOM objects in the same way they invoke methods on C++ objects. In addition, SOM objects receive full C++ typechecking, just as C++ objects do. Currently, the SOM Compiler can generate both C and C++ language bindings for a class. The C and C++ bindings will work with a variety of commercial products available from IBM and others. Vendors of other programming languages may also offer SOM bindings. Check with your language vendor about possible SOM support.

The SOM run-time library

In addition to the SOM Compiler, SOM includes a **run-time library**. This library provides, among other things, a set of classes, methods, and procedures used to create objects and invoke methods on them. The library allows any programming language to use SOM classes (classes developed using SOM) if that language can:

- Call external procedures,
- Store a pointer to a procedure and subsequently invoke that procedure, and
- Map IDL types onto the programming language's native types.

Thus, the user of a SOM class and the implementor of a SOM class needn't use the same programming language, and neither is required to use an object-oriented language. The independence of client language and implementation language also extends to subclassing: a SOM class can be derived from other SOM classes, and the subclass may or may not be implemented in the same language as the parent class(es). Moreover, SOM's run-time environment allows applications to access information about classes dynamically (at run time).

Frameworks provided in the SOMobjects Toolkit

In addition to SOM itself (the SOM Compiler and the SOM run-time library), the SOMobjects Developer Toolkit also provides a set of *frameworks* (class libraries) that can be used in developing object-oriented applications. These include Distributed SOM, the Interface Repository Framework, the Persistence Framework, the Replication Framework, and the Emitter Framework, described below.

Distributed SOM

Distributed SOM (or **DSOM**) allows application programs to access SOM objects across address spaces. That is, application programs can access objects in other processes, even on 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, and the client accesses the object as if were local. The current release of DSOM supports distribution of objects among processes within a workstation, and across a local area network consisting of OS/2 systems, AIX systems. or a mix of both. Future releases may support larger enterprise–wide networks.

Interface Repository Framework

The **Interface Repository** is a database, optionally created and maintained by the SOM Compiler, that holds all the information contained in the IDL description of a class of objects. The **Interface Repository Framework** consists of the 11 classes defined in the CORBA standard for accessing the Interface Repository. Thus, the Interface Repository Framework provides run-time access to all information contained in the IDL description of a class of objects. Type information is available as **TypeCode**s — a CORBA-defined way of encoding the complete description of any data type that can be constructed in IDL.

Persistence Framework

The **Persistence Framework** is a collection of SOM classes that provide methods for saving objects (either in a file or in a more specialized repository) and later restoring them. This means that 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 so forth. The Persistence Framework includes the following features:

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

Replication Framework

The **Replication Framework** is a collection of SOM classes that allows a replica (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 mutual consistency among the replicas. The Replication Framework includes these important features:

- Good response times for both readers and writers,
- Fault-tolerance against node failures and message loss,
- Simple coding rules (that can be automated) for building replicated objects,
- Graceful degradation under wide-area networks, and
- Minimal overhead when replication is not activated.

Emitter Framework

Finally, the **Emitter Framework** is a collection of SOM classes that allows programmers to write their own emitters. *Emitter* is a general term used to describe a back-end output component of the SOM Compiler. Each emitter takes as input 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). In addition, users may wish to 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/C++. The Emitter Framework is separately documented in the *SOMobjects Developer Toolkit: Emitter Framework Guide and Reference.*

Basic Concepts of the System Object Model (SOM)

The **System Object Model (SOM),** provided by the **SOMobjects Developer Toolkit**, is a set of libraries, utilities, and conventions used to create binary class libraries that can be used by application programs written in various object-oriented programming languages, such as C++ and Smalltalk, or in traditional procedural languages, such as C and Cobol. The following paragraphs introduce some of the basic terminology used when creating classes in SOM:

- An *object* is an OOP entity that has *behavior* (its *methods* or operations) and *state* (its data values). In SOM, an object is a run-time entity with a specific set of methods and instance variables. The methods are used by a client programmer to make the object exhibit behavior (that is, to do something), and the instance variables are used by the object to store its state. (The state of an object can change over time, which allows the object's behavior to change.) When a method is invoked on an object, the object is said to be the *receiver* or *target* of the method call.
- An object's *implementation* is determined by the procedures that execute its methods, and by the type and layout of its instance variables. The procedures and instance variables that implement an object are normally *encapsulated* (hidden from the caller), so a program can use the object's methods without knowing anything about how those methods are implemented. Instead, a user is given access to the object's methods through its *interface* (a description of the methods in terms of the data elements required as input and the type of value each method returns).
- An interface through which an object may be manipulated is represented by an *object type*. That is, by declaring a type for an object variable, a programmer specifies the interface that is intended to be used to access that object. *SOM IDL* (the **SOM Interface Definition Language**) is used to define object interfaces. The *interface names* used in these IDL definitions are also the type names used by programmers when typing SOM object variables.
- In SOM, as in most approaches to object-oriented programming, a *class* defines the implementation of objects. That is, the implementation of any SOM object (as well as its interface) is defined by some specific SOM class. A class definition begins with an IDL specification of the interface to its objects, and the name of this interface is used as the class name as well. Each object of a given class may also be called an *instance* of the class, or an *instantiation* of the class.
- *Inheritance,* or *class derivation*, is a technique for developing new classes from existing classes. The original class is called the *base* class, or the *parent* class, or sometimes the direct *ancestor* class. The derived class is called a *child* class or a *subclass*. The primary advantage of inheritance is that a derived class inherits all of its parent's methods and instance variables. Also through inheritance, a new class can *override* (or redefine) methods of its parent, in order to provide enhanced functionality as needed. In addition, a derived class can introduce new methods of its own. If a class results from several generations of successive class derivation, that class "knows" all of its ancestors's methods (whether overridden or not), and an object (or instance) of that class can execute any of those methods.
- SOM classes can also take advantage of *multiple inheritance,* which means that a new class is jointly derived from two or more parent classes. In this case, the derived class inherits methods from all of its parents (and all of its ancestors), giving it greatly expanded capabilities. In the event that different parents have methods of the same name that execute differently, SOM provides ways for avoiding conflicts.
- In the SOM run time, classes are themselves objects. That is, classes have their own methods and interfaces, and are themselves defined by other classes. For this reason, a class is often called a *class object*. Likewise, the terms *class methods* and *class variables*

are used to distinguish between the methods/variables of a class object vs. those of its instances. (Note that the type of an object is *not* the same as the type of its class, which as a “class object” has its own type.)

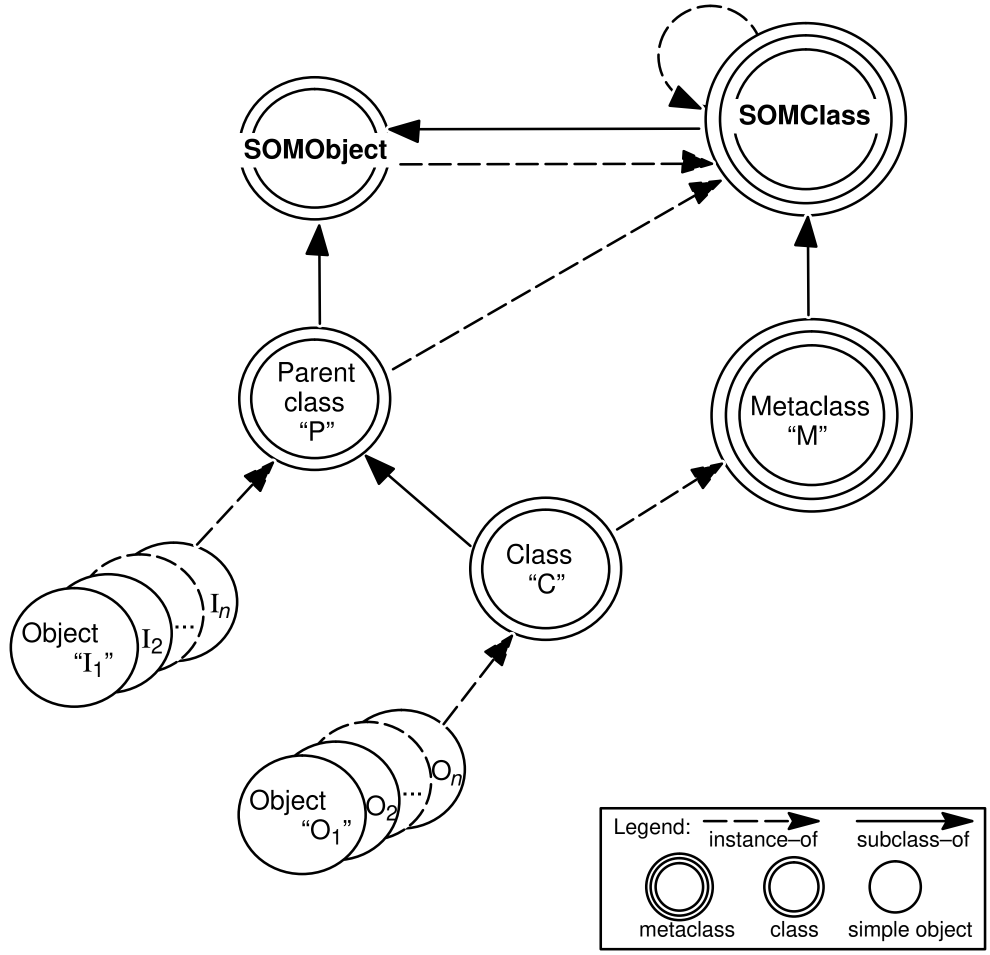
- A class that defines the implementation of class objects is called a *metaclass*. Just as an instance of a class is an object, so an instance of a metaclass is a class object. Moreover, just as an ordinary class defines methods that its objects respond to, so a metaclass defines methods that a class object responds to. For example, such methods might involve operations that execute when a class (that is, a class object) is creating an instance of itself (an object). Just as classes are derived from parent classes, so metaclasses can be derived from parent metaclasses, in order to define new functionality for class objects.

- The SOM system contains three primitive classes that are the basis for all subsequent classes:

  SOMObject — the root ancestor class for all SOM classes,
  SOMClass — the root ancestor class for all SOM metaclasses, and
  SOMClassMgr — the class of the SOMClassMgrObject, an object created automatically during SOM initialization, to maintain a registry of existing classes and to assist in dynamic class loading/unloading.

  SOMClass is defined as a subclass (or child) of **SOMObject** and inherits all generic object methods; this is why instances of a metaclass are class *objects* (rather than simply classes) in the SOM run time. The adjacent figure illustrates typical relationships of classes, metaclasses, and objects in the SOM run time. (This illustration does not include the SOMClassMgrObject.)

Typical class, metaclass, and object relationships



SOM classes are designed to be *language neutral*. That is, SOM classes can be implemented in one programming language and used in programs of another language. To achieve language neutrality, the *interface* for a class of objects must be defined separately from its *implementation*. That is, defining interface and implementation requires two completely separate steps (plus an intervening compile), as follows:

- An *interface* is the information that a program must know in order to use an object of a particular class. This interface is described in an interface definition (which is also the class definition), using a formal language whose syntax is independent of the programming language used to implement the class's methods. For SOM classes, this is the SOM Interface Definition Language (SOM IDL). The interface is defined in a file known as the *IDL source file* (or, using its extension, this is often called the *.idl file*).

  An interface definition is specified within the *interface declaration* (or *interface statement*) of the .idl file, which includes:

  (a) the interface name (or class name) and the name(s) of the class's parent(s), and
  (b) the names of the class's attributes and the signatures of its new methods. (Recall that the complete set of available methods also includes all inherited methods.)

  Each *method signature* includes the method name, and the type and order of its arguments, as well as the type of its return value (if any). *Attributes* are instance variables for which “set” and “get” methods will automatically be defined, for use by the application program. (By contrast, instance variables that are not attributes are hidden from the user.)

- Once the IDL source file is complete, the *SOM Compiler* is used to analyze the .idl file and create the *implementation template file*, within which the class implementation will be defined. Before issuing the SOM Compiler command, **sc**, the class implementor can set an environment variable that determines which *emitters* (output-generating programs) the SOM Compiler will call and, consequently, which programming language and operating system the resulting *binding files* will relate to. (Alternatively, this emitter information can be placed on the command line for **sc.**) In addition to the implementation template file itself, the binding files include two language-specific header files that will be #included in the implementation template file and in application program files. The header files define many useful SOM macros, functions, and procedures that can be invoked from the files that include the header files.
- The *implementation* of a class is done by the class implementor in the *implementation template file* (often called just the *implementation file* or the *template file*). As produced by the SOM Compiler, the template file contains *stub procedures* for each method of the class. These are incomplete method procedures that the class implementor uses as a basis for implementing the class by writing the corresponding code in the programming language of choice.

In summary, the process of *implementing a SOM class* includes using the SOM IDL syntax to create an IDL source file that specifies the interface to a class of objects — that is, the methods and attributes that a program can use to manipulate an object of that class. The SOM Compiler is then run to produce an implementation template file and two binding (header) files that are specific to the designated programming language and operating system. Finally, the class implementor writes language-specific code in the template file to implement the method procedures.

At this point, the next step is to write the application (or client) program(s) that use the objects and methods of the newly implemented class. (Observe, here, that a programmer could write an application program using a class implemented entirely by someone else.) If not done previously, the SOM compiler is run to generate usage bindings for the new class, as appropriate for the language used by the client program (which may be different from the language in which the class was implemented). After the client program is finished, the programmer compiles and links it using a language-specific compiler, and then executes the program. (Notice again, the client program can invoke methods on objects of the SOM class without knowing how those methods are implemented.)

What’s New in the SOMobjects Developer Toolkit

The SOMobjects Toolkit is a major step up from SOM Version1.0 in terms of functionality, usability, standardization, performance, and documentation. In particular, the SOMobjects Developer Toolkit Version 2.0 offers the following additions over SOM Version1.0:

- C++ bindings
- Multiple platforms (OS/2 and AIX)
- Multiple inheritance (when using IDL)
- Derived metaclasses
- CORBA compliance (including IDL)
- SOM Compiler that handles both IDL and OIDL interface descriptions
- Full binary compatibility with SOM 1.0
- Improved method resolution
- Improved memory management
- Improved error checking
- Faster emitters
- Smaller and faster binaries for class libraries
- Header files that automatically prevent name collisions
- Smaller implementation header files
- Ability to package multiple classes in a single file, as an IDL module
- Ability to define (in IDL) methods that return structures
- Ability to use a C/C++ preprocessor within .idl files
- Improved incremental update of implementation files
- With IDL, full type checking and full name scoping
- Class shadowing (described in the *Emitter Framework Guide and Reference*)
- DSOM, which enables distribution of SOM objects across processes (see Chapter 6)
- Automatic generation and revision of an Interface Repository (see Chapter 7)
- Persistence Framework, which simplifies creating persistent objects (see Chapter 8)
- Replication Framework, which enables construction of replicated objects (see Chapter 9)
- Utility metaclasses (see Chapter 10)
- Collection classes (see Chapter 11)
- An Event Manager (see Chapter 12)
- Tools for automatically converting .csc files to .idl files (see Appendix B)
- Emitter Framework, which allows developers to write their own emitters (documented in the *Emitter Framework Guide and Reference*)