Bull DPX/20

SOMobject Base Toolkit
User's Guide

AIX

Bull DPX/20

SOMobject Base Toolkit
User's Guide

AIX

Software

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

Trademarks and Acknowledgements

We acknowledge the right of proprietors of trademarks mentioned in this book.

AIX® is a registered trademark of International Business Machines Corporation, and is being used under licence.

UNIX is a registered trademark in the USA and other countries licensed exclusively through X/Open.

The information in this document is subject to change without notice. Groupe Bull will not be liable for errors contained herein, or for incidental or consequential damages in connection with the use of this material.

About This Book

This book accompanies the **SOMobjects Base Toolkit**, which consists of the base (or core) capabilities in the **System Object Model** (**SOM**) of the **SOMobjects Developer Toolkit.** The base system is a rich subset of the full-capability SOMobjects Developer Toolkit. Because the current manual is a subset of chapters from the full *SOMobjects Developer Toolkit Users Guide*, references herein to the "SOMobjects Developer Toolkit" should be interpreted as capabilities of the SOMobjects Base Toolkit.

This manual explains how programmers using C, C++, and other languages can:

- Implement class libraries that exploit the SOM library-packaging technology,

- Develop client programs that use class libraries that were built using SOM, and

- Develop applications that use the *frameworks* supplied with this core version of the SOMobjects Toolkit, class libraries that facilitate development of object-oriented applications.

In addition to this book, refer to the *SOMobjects Base Toolkit Programmer's Reference Manual* during application development for specific information about the classes, methods, functions, and macros of the SOMobjects Toolkit. Also, the *SOMobjects Base Toolkit Quick Reference Guide* shows the syntax of each method or function, along with a one-sentence description of its purpose, and gives syntax for the SOM Compiler commands and option flags.

For purchasers of the full-capability SOMobjects system, the *SOMobjects Developer Toolkit: Emitter Framework Guide and Reference* contains documentation of the Emitter Framework of the SOMobjects Developer Toolkit. In addition, the *SOMobjects Developer Toolkit: Collection Classes Reference Manual* describes the collection classes and methods provided with the SOMobjects Developer Toolkit.

How This Book Is Organized

This book contains nine chapters, three appendices, a glossary, and an index.

- The first part (chapters 1 and 2) introduces the reader to the SOMobjects Toolkit, gives an overview of the major elements of SOM, and provides a Tutorial containing several evolutionary examples.

- The second part (chapter 3) describes how to develop client programs in C, C++, or other languages to use classes that were implemented using SOM, including how to create instances of a class and call methods on them.

- The third part (chapters 4 and 5) describes the SOM run-time environment, gives the syntax of the SOM Interface Definition Language, provides directions for running the SOM Compiler, and contains advanced information about SOM's object model and how to customize SOM for particular applications.

- The fourth part (chapters 6 through 9) contains information about the frameworks composing the SOMobjects Toolkit: Distributed SOM (DSOM), the Interface Repository Framework, and the Event Management Framework. This part also describes the utility metaclasses provided with the SOMobjects Toolkit.

- The appendices describe customer support-procedures and error codes, and provide the grammar for SOM IDL. The final appendix describes how to subclass a "Sockets" class that facilitates inter-process communications required by the DSOM and Event Management Frameworks.

- The Glossary provides brief definitions of terminology related to SOM and the SOMobjects Toolkit. Finally, an extensive Index enables the reader to locate specific information quickly.

Who Should Read This Book

This book is for the professional programmer using C, C++, or another language who wishes to:

- Use SOM to build object-oriented class libraries, or

- Write application programs using class libraries that others have implemented using SOM, even if the programming language does not directly support object-oriented programming.

The discussions in this book are expressed in the commonly used terminology of object-oriented programming. A number of important terms are everyday English words that take on specialized meanings. These terms appear in the Glossary at the back of this book. You may find it worth consulting the Glossary if the unusual significance attached to an otherwise ordinary word puzzles you.

This book assumes that you are an experienced programmer and that you have a general familiarity with the basic notions of object-oriented programming. Practical experience using an object-oriented programming language is helpful, but not essential.

If you would like a good introduction to object-oriented programming or a general survey of the many aspects of the topic, you might enjoy reading one of the following books:

- Booch, G, *Object-Oriented Design with Applications*, Benjamin/Cummings 1991, ISBN 0-8053-0091-0.

- Budd, T, *An Introduction to Object-Oriented Programming*, Addison-Wesley 1991, ISBN 0-201-54709-0.

- Cox, B, and Novobilski, A, *Object-Oriented Programming, An Evolutionary Approach* 2nd Edition, Addison-Wesley 1991, ISBN 0-201-54834-8.

Contents

Chapter 1. Introduction to the SOMobjects Toolkit

1.1 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)** — the core of the SOMobjects Base Toolkit — provides a solution to these problems.

1.2 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, 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. 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 standard procedure calls. 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 API (the way that programmers invoke methods, create objects, and so on) can vary widely from language to language, 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 mechanisms for method resolution 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:

   - Offset resolution: 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.

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. 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 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, the Emitter Framework, and the Metaclass Framework, described below.

Note: **SOMobjects Base Toolkit**, the core version of SOMobjects shipped with this docu-mentation, contains *only* Distributed SOM. the Interface Repository, and the Metaclass frameworks. SOMobjects manuals are available individually.

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 a different machine. 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. DSOM in the core-capability SOMobjects Base Toolkit supports distribution of objects among processes within a single workstation. The full-capability version of DSOM also supports distribution across a local area network consisting of AIX systems, OS/2 systems, Windows systems, or some mix of these systems. 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.*

Metaclass Framework

Finally, the **Metaclass Framework** is a collection of SOM metaclasses that provide functionality that may be useful to SOM class designers for modifying the default semantics of method invocation and object creation. These metaclasses are described in Chapter 8, "The Metaclass Framework."

1.3 What's New in SOMobjects Version 2.1

Version 2.1 of the SOMobjects Developer Toolkit provides enhanced capabilities and improved performance for both SOM and DSOM. In addition, the Toolkit now includes support for DirectToSOM (DTS) C++ compilers. New metaclasses in the Metaclass Framework allow class implementors to define classes that automatically possess certain convenience facilities. Also, TCP/IP support is available for Windows users, and OS/2 users can choose 32–bit TCP/IP.

In particular, SOMobjects Version 2.1 offers the following additions over Version 2.0:

General enhancements

- C++ programmers can use DirectToSOM (DTS) C++ compilers (available from independent software vendors) as an alternative to the SOMobjects Toolkit's C++ bindings. (A DTS C++ compiler uses SOM classes to implement C++ objects.) The support provided by SOMobjects for DTS C++ consists of various enhancements to the SOM API (useful to SOM programmers in general), and a new emitter that produces DTS C++ header files corresponding to SOM IDL. DTS C++ programs #include these ".hh" header files, which are emitted as described under "Generating binding files" in Chapter 4, "SOM IDL and the SOM Compiler."

- With this release, TCP/IP support is now available for SOMobjects For Windows. Also, for OS/2 users only, SOMobjects now supports the 32–bit TCP/IP version 2.0, which offers greater performance over the 16–bit TCP/IP version 1.2.1. These are described in your *SOMobjects Installation/Configuration Guide*.

SOMobjects enhancements

- A new default process whereby SOMobjects initializes and destroys objects more efficiently (using the **somDefaultInit** and **somDestruct** methods). See "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM."

- A new kind of method, **nonstatic**, that is similar to a C++ nonstatic member function. It is normally invoked using offset resolution but can use any form of SOMobjects method resolution. See "The four kinds of SOM methods" in Chapter 5, "Implementing Classes in SOM" and the **nonstatic** modifier under "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."

- A new kind of data, **staticdata**, that is similar to C++ static data members. A **staticdata** variable is not stored in an object; rather, the ClassData structure of the implementing class contains a pointer to the variable. See the **staticdata** modifier under "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."

- Two new modifiers, **somallocate** and **somdeallocate**, used to indicate that a user-written procedure should be executed to allocate/deallocate memory for class instances when the **somAllocate** or **somDeallocate** method is invoked. See these modifiers under "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."

- The ability to "cast" objects. See the methods **somCastObj** and **somResetObj**.

- New support for loading/unloading class libraries. (This was introduced in SOMobjects For Windows and is now applicable on OS/2 as well.) See the **SOM_ClassLibrary** macro in the *SOMobjects Developer Toolkit Programmers Reference Manual*.

DSOM enhancements

- A new command-line tool, **dsom**, for managing DSOM servers (starting, restarting, stopping, querying, and so forth). See "The **dsom** server manager utility" in Chapter 6, "Distributed SOM." In addition, a corresponding programmatic interface is provided by methods of the new **SOMDServerMgr** class. See that class and its methods in the *SOMobjects Developer Toolkit Programmers Reference Manual*.

- A new command-line tool, **somdchk**, that performs environment validation. See "Verifying the DSOM environment with **somdchk**" in Chapter 6, "Distributed SOM."

- New SOM IDL modifiers for memory management of parameters: **memory_management = corba**, **caller_owns_parameters, caller_owns_result, object_owns_parameters,** and **object_owns_result.** The individual modifiers are described under "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler." See "Memory management" in Chapter 6, "Distributed SOM," for a complete discussion of the subject. This memory-management support also includes the new method **somdReleaseResources** and the functions **somdExceptionFree** and **SOMD_NoORBfree**, described in the *SOMobjects Developer Toolkit Programmers Reference Manual*.

- A graphical user interface to the DSOM 'regimpl' utility on OS/2, called **pregimpl**. See "Registration steps using 'pregimpl' or 'wregimpl'," in Chapter 6, "Distributed SOM."

- Support for the CORBA constant OBJECT_NIL. In addition, the **is_nil** method of **SOMDObject** can now be used for local objects as well as NULL pointers.

- Support for passing self-referential structs and unions (those valid in IDL) in remote method calls.

- Support for local/remote transparency in DSOM's object-destruction methods. See "Destroying remote objects."

- Ability for users to define customized base proxy classes. See "Customizing the default base proxy class."

- Ability for users to specify an upper limit on the number of threads that a server can spawn, via the SOMDNUMTHREADS environment variable. See this variable under "Preparing the environment."

- Improved error handling and performance improvements.

- More sample programs distributed with the SOMobjects Toolkit.

Metaclass Framework

- **SOMMBeforeAfter** — a metaclass that enables the programming of "before/after" metaclasses, whose instances execute a particular method before and after each method invocation. See Chapter 8, "The Metaclass Framework," for information about the new Metaclass Framework. Individual metaclasses, along with related classes and methods, are documented in the *SOMobjects Developer Toolkit Programmers Reference Manual*.

- **SOMMTraced** — a utility metaclass to do tracing.

New restrictions and deprecated methods

While implementing the Metaclass Framework, we learned that metaclasses must be programmed so that the capabilities they implement will be preserved when various metaclasses are combined (using multiple inheritance) into a SOM-derived metaclass. To assure this result, the Metaclass Framework metaclasses have been programmed using a "Cooperative Metaclass." However, we are not yet ready to include the Cooperative metaclass among the officially supported features of SOMobjects.

To prevent user-defined metaclasses from interfering with the operation of the Cooperative Metaclass and consequently with the Metaclass Framework, SOMobjects programmers are strongly urged to observe the following restriction when programming new metaclasses:

- User-defined metaclasses can introduce new class methods and class variables, but should *not* override any of the methods introduced by the **SOMClass** class*.*

SOMobjects users whose metaclass programming requirements cannot be met within the above restrictions will be given access to the Cooperative Metaclass and its documentation. Note, however, that metaclasses developed using the Cooperative Metaclass may require reprogramming when an officially supported Cooperative Metaclass is later introduced.

In addition, use of a number of (public) methods introduced by **SOMClass** is now deprecated because they are useful only from overridden **SOMClass** methods. These methods are listed under the heading "Deprecated methods" in the documentation for **SOMClass** within the *SOMobjects Developer Toolkit Programmers Reference Manual*, until such time as SOMobjects is ready to officially provide a framework within which their use will not interfere with the internal operation of SOMobjects itself.

1.4 Overview of this book

This book is a subset of the *SOMobjects Developer Toolkit Users Guide* for the full-capability SOMobjects Developer Toolkit. The omitted chapters describe capabilities not included with the SOMobjects Base Toolkit. Chapters that are included appear in their entirety, as written for the full-capability SOMobjects system, except for some references to chapters/capabilities not included in the SOMobjects Base Toolkit. Because the current manual is a subset, references herein to the "SOMobjects Developer Toolkit" should be interpreted as referencing capabilities of the SOMobjects Base Toolkit.

Chapter 2 contains a Tutorial with six examples that illustrate techniques for implementing classes in SOM. *All* readers should cover this tutorial.

Chapter 3 describes how an application program creates instances of a SOM class, how it invokes methods, and so on. For readers interested only in *using* SOM classes, rather than building them, chapters 2 and 3 may provide all of the information they need to begin using SOM classes.

Chapter 4 provides more comprehensive information about the SOM system itself, including operation of the SOM run-time environment, method resolution, and inheritance. This chapter also describes how to create language-neutral class libraries using SOM, including explanations of valid syntax components. All class implementors will need to reference this chapter.

Chapter 5 deals with advanced topics for customizing SOM to better suit the needs of a particular application.

Chapter 6 describes **Distributed SOM** and how to use it to access objects across address spaces, even on different machines. Chapter 6 also describes how to customize DSOM. Note that SOMobjects Base Toolkit supports only *Workstation DSOM* (distribution among processes on a single machine), whereas SOMobjects Developer Toolkit also supports *Workgroup DSOM* (distribution among a network of machines).

Chapter 7 describes the **Interface Repository Framework** of classes supplied with the SOMobjects Toolkit.

Chapter 8 describes the **Metaclass Framework** and some utility metaclasses that SOM provides to assist users in deriving new classes with special abilities to execute "before" and "after" operations when a method call occurs, as well as other capabilities for modifying the default semantics of method invocation and object creation.

Chapter 9 describes the **Event Management Framework**, which allows grouping of all application events and waiting on multiple events in one place. The Event Manager is used by both DSOM and the Replication Framework (the latter is not part of the SOMobjects Base Toolkit).

Appendix A describes service and technical support policies for the SOMobjects Toolkit, and contains lists of the error codes and messages that can be issued by the SOM kernel or by the included frameworks.

Appendix B contains the SOM IDL language grammar.

Appendix C describes how to subclass a "Sockets" class that facilitates inter-process communications required by the DSOM and Event Management Frameworks.

A Glossary and a thorough Index conclude this manual.

Chapter 2. Tutorial for Implementing SOM Classes

This tutorial contains five examples showing how SOM classes can be implemented to achieve various functionality. Obviously, for any person who wishes to become a class implementor, this tutorial is essential. However, even for those programmers who expect only to *use* SOM classes that were implemented by others, the tutorial is also necessary, as it presents several concepts that will help clarify the process of using SOM classes.

2.1 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 an interface that can 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
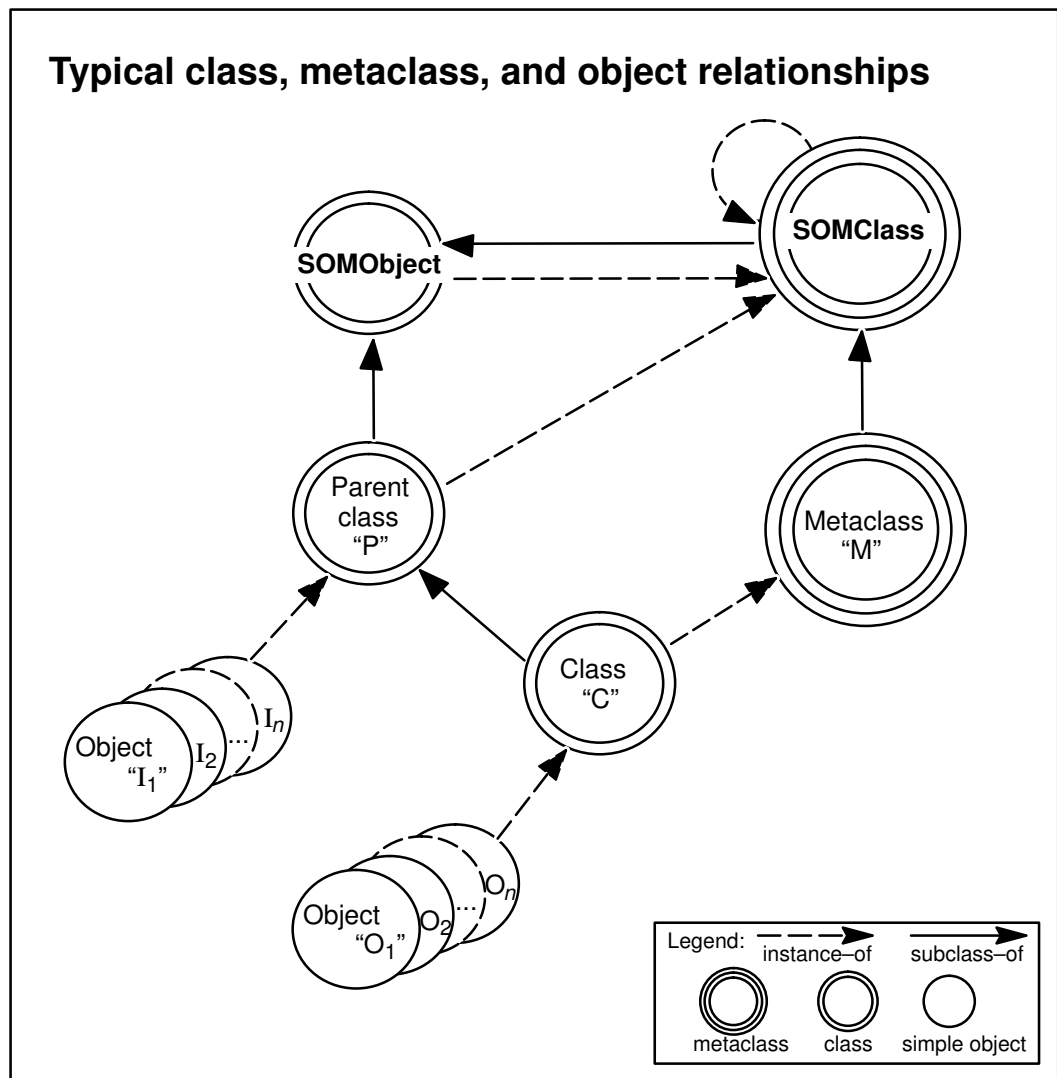
SOM classes are designed to be *language neutral*. That is, SOM classes can be implemented in one programming language and used in programs of another language. To achieve language neutrality, the *interface* for a class of objects must be defined separately from its *implementation*. That is, defining 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** (or **somc** on Windows), 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** or **somc.**)

  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.)

Development of the Tutorial examples

- **Example 1 — Implementing a simple class with one method**
  Prints a default message when the "sayHello" method is invoked on an object of the "Hello" class.

- **Example 2 — Adding an attribute to the Hello class**
  Defines a "msg" attribute for the "sayHello" method to use. The client program "sets" a message; then the "sayHello" method "gets" the message and prints it. (There is no defined message when an object of the "Hello" class is first created.)

- **Example 3 — Overriding an inherited method**
  Overrides the SOMobjects method **somPrintSelf** so that invoking this method on an object of the "Hello" class will not only display the class name and the object's location, but will also include the object's message attribute.

- **Example 4 — Initializing a SOM object**
  Overrides the default initialization method, **somDefaultInit**, to illustrate how an object's instance variables can be initialized when the object is created.

- **Example 5 — Using multiple inheritance**
  Extends the "Hello" class to provide it with multiple inheritance (from the "Disk" and "Printer" classes). The "Hello" interface defines an enum and an "output" attribute that takes its value from the enum (either "screen", "printer", or "disk"). The client program "sets" the form of "output" before invoking the "sayHello" method to send a "msg" (defined as in Example 4).

2.2 Basic Steps for Implementing SOM Classes

Implementing and using SOM classes in C or C++ involves the following steps, which are explicitly illustrated in the examples of this tutorial:

1. Define the interface to objects of the new class (that is, the interface declaration), by creating a **.idl** file.

2. Run the SOM Compiler on the .idl file (by issuing the **sc** command on AIX or OS/2, or by issuing the **somc** command on Windows) to produce the following binding files:

   - Template implementation file
     a **.c** file for C programmers, or
     a **.C** file (on AIX) or a **.cpp** file (on OS/2 or Windows) for C++ programmers;

   - Header file to be included in the implementation file
     a **.ih** file for C programmers, or
     a **.xih** file for C++ programmers; and

   - Header file to be included in client programs that use the class
     a **.h** file for C clients, or
     a **.xh** file for C++ clients.

   To specify whether the SOM Compiler should produce C or C++ bindings, set the value of the SMEMIT environment variable or use the "**–s**" option of the **sc** or **somc** command, as described in Section 4.3, "The SOM Compiler." By default, the SOM Compiler produces C bindings.

3. Customize the implementation, by adding code to the template implementation file.

4. Create a client program that uses the class.

5. Compile and link the client code with the class implementation, using a C or C++ compiler.

6. Execute the client program.

The following examples illustrate appropriate syntax for defining interface declarations in a .idl file, including designating the methods that the class's instances will perform. In addition, the example template implementation files contain typical code that the SOM Compiler produces. Explanations accompanying each example discuss topics that are significant to the particular example; full explanations of the SOM IDL syntax are contained in Chapter 4, "SOM IDL and the SOM Compiler." Customization of each implementation file (step 3) is illustrated in both C and C++.

Notes: (1) The Tutorial assumes you will work through the examples in order. If you do *not* do so, the code that the SOM Compiler generates from your revised .idl file may vary slightly from what you see in the Tutorial.
(2) When the SOMobjects Toolkit is installed, a choice is made between "somcorba" and "somstars" for the style of C bindings the SOM Compiler will generate. The Tutorial examples use the "somcorba" style, where an interface name used as a type indicates a *pointer* to an object, as required by strict CORBA bindings. Consequently, as the examples show, a "*****" does not explicitly appear for types that are pointers to objects. If your system was installed for "somstars" C bindings, you can set the environment variable SMADDSTAR=1 or use the SOM Compiler option "–maddstar" to request bindings that use explicit pointer stars. For more information, see "Declaring object variables" in Chapter 3,"Using SOM Classes in Client Pro-grams" and "Object types" in Chapter 4, "SOM IDL and the SOM Compiler."

Example 1 — Implementing a Simple Class with One Method

Example 1 defines a class "Hello" which introduces one new method, "sayHello". When invoked from a client program, the "sayHello" method will print the fixed string "Hello, World!" The example follows the six steps described in the preceding topic, "Basic Steps for Implementing SOM Classes."

1) Define the interface to class "Hello", which inherits methods from the root class **SOMObject** and introduces one new method, "sayHello". Define these IDL specifications in the file "hello.idl".

The "interface" statement introduces the <u>name of a new class</u> and any <u>parents</u> (base classes) it may have (here, the root class **SOMObject**). The body of the interface declaration introduces the method "sayHello." Observe that method declarations in IDL have syntax similar to C and C++ function prototypes:

```
#include <somobj.idl>  //# Get the parent class definition.

interface Hello : SOMObject
/*  This is a simple class that demonstrates how to define the
 *  interface to a new class of objects in SOM IDL.
 */
{
    void sayHello();
    // This method outputs the string "Hello, World!".
       /* On Windows, use:   string sayHello();
        * This method returns the string "Hello, World!". */
};
```

Note that the method "sayHello" has no (explicit) arguments and returns no value (except on Windows, which returns a string). The characters "//" start a line comment, which finishes at the end of the line. The characters "/*" start a block comment that finishes with "*/". Block comments do not nest. The two comment styles can be used interchangeably. Throw-away comments are also permitted in a .idl file; they are ignored by the SOM Compiler. Throw-away comments start with the characters "//#" and terminate at the end of the line.

Note: For simplicity, this IDL fragment does not include a **releaseorder** modifier; consequently, the SOM Compiler will issue a warning for the method "sayHello". For directions on using the **releaseorder** modifier to remove this warning, see the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler." (The warning does not prohibit continued use of the .idl file.)

2) Run the SOM Compiler to produce binding files and an implementation template. That is, issue the **sc** command on AIX or OS/2, as follows:

```
> sc -s"c;h;ih" hello.idl       (for C bindings on AIX or OS/2)

> sc -s"xc;xh;xih" hello.idl   (for C++ bindings on AIX or OS/2)
```

On Windows, issue the **somc** command (started from the Run option on the File menu):

```
> somc -sc;h;ih hello.idl       (for C bindings on Windows)

> somc -sxc;xh;xih hello.idl   (for C++ bindings on Windows)
```

When set to generate C binding files, the SOM Compiler generates the following template implementation file, named "hello.c". The <u>template implementation file contains *stub procedures* for each new method</u>; these are procedures whose bodies are largely vacuous, to be filled in by the implementor. (Unimportant details have been removed for this tutorial.)

Under AIX or **OS/2**:

```
#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");
}
```

Under Windows:

```
#include <hello.ih>

/*
 *  This method returns the string "Hello, World!".
 */

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

The terms "SOM_Scope" and "SOMLINK" appear in the prototype for all stub procedures, but they are defined by SOM and are not of interest to the developer. In the method procedure for the "sayHello" method, "somSelf" is a pointer to the target object (here, an instance of the class "Hello") that will respond to the method. A "somSelf" parameter appears in the procedure prototype for every method, since SOM requires every method to act on some object.

The *target object* is always the first parameter of a method's procedure, although it should *not* be included in the method's IDL specification. The second parameter (which also is not included in the method's IDL specification) is the parameter **(Environment *ev)**. This parameter can be used by the method to return exception information if the method encounters an error. (Contrast the prototype for the "sayHello" method in steps 1 and 2 above.)

The remaining lines of the template above are not pertinent at this point. (For those interested, they are discussed in section 5.4 of Chapter 5, "Implementing Classes in SOM.") The file is now ready for customization with the C code needed to implement method "sayHello".

When set to generate C++ binding files, the SOM Compiler generates an implementation template file, "hello.C" (on AIX) or "hello.cpp (on OS/2 or Windows), similar to the one above. (Chapter 5 discusses the implementation template in more detail.)

Recall that, in addition to generating a template implementation file as shown above, the SOM Compiler also generates implementation bindings (in a header file to be included in the implementation file) and usage bindings (in a header file to be included in client programs). These files are named "hello.ih" and "hello.h" for C bindings, and are "hello.xih" and "hello.xh" for C++ bindings. Notice that the "hello.c" file shown above includes the "hello.ih" implementation binding file.

3) Customize the implementation, by adding code to the template implementation file.

Modify the body of the "sayHello" method procedure in the "hello.c" (or, for C++, "hello.C" on AIX, "hello.cpp" on OS/2) implementation file so that the "sayHello" method prints "Hello, World!":

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

    printf("Hello, World!\n");
        /*  On Windows, use:  return ("Hello, World!");   */
}
```

4) Create a client program that uses the class.

Write a program "main" that creates an instance (object) of the "Hello" class and invokes the method "sayHello" on that object.

Under AIX or OS/2:
A C programmer would write the following program in "main.c", which uses the bindings defined in the "hello.h" header file:

```
#include <hello.h>

int main(int argc, char *argv[])
{
    /* Declare a variable to point to an instance of Hello */
    Hello obj;

    /* Create an instance of the Hello class */
    obj = HelloNew();

    /* Execute the "sayHello" method */
    _sayHello(obj, somGetGlobalEnvironment());

    /* Free the instance: */
    _somFree(obj);
    return (0);
}
```

Notice the statement `obj = HelloNew();` The "hello.h" header file automatically contains the SOM-defined macro *<className>***New()**, which is used to create an instance of the *<className>* class (here, the "Hello" class).

Also notice that, in C, a method is invoked on an object by using the form:

_*<methodName>***(**_*<objectName>*, *<environment_arg>*, *<other_method_args>***)**

as used above in the statement `_sayHello(obj, somGetGlobalEnvironment())`. As shown in this example, the SOM-provided **somGetGlobalEnvironment** function can be used to supply the (**Environment** *) argument of the method.

Finally, the code uses the method **somFree**, which SOM also provides, to free the object created by `HelloNew()`. Notice that **somFree** does not require an (**Environment** *) argument. This is because the method procedures for some of the classes in the SOMobjects Toolkit (including **SOMObject**, **SOMClass**, and **SOMClassMgr**) do not have an Environment parameter, to ensure compatibility with the first release of SOM. The documentation for each SOM-kernel method in the *SOMobjects Developer Toolkit Programmers Reference Manual* indicates whether an Environment parameter is used.

A C++ programmer would write the following program in "main.C" (on AIX) or "main.cpp" (on OS/2), using the bindings defined in the "hello.xh" header file:

```
#include <hello.xh>

int main(int argc, char *argv[])
{
    /* Declare a variable to point to an instance of Hello */
    Hello *obj;

    /* Create an instance of the Hello class */
    obj = new Hello;

    /* Execute the "sayHello" method */
    obj->sayHello(somGetGlobalEnvironment());

    obj->somFree();
    return (0);
}
```

Notice that the only argument passed to the "sayHello" method by a C++ client program is the **Environment** pointer. (Contrast this with the invocation of "sayHello" in the C client program, above.)

Under Windows:
The subsequent "main.c" file fragment uses bindings defined in the "hello.h" and "windows.h" header files, and contains C code illustrating the WinMain and WndProc procedures that are pertinent to Step 4. The macro **SOM_MainProgram** is recommended as the first line of executed code in each executable Windows program, such as WinMain (note: it does not work from a DLL). **SOM_MainProgram** combines execution of the **somMainProgram** function with scheduling of the **somEnvironmentEnd** function, which frees classes introduced by your application when they are no longer needed.

```
#include <windows.h>
#include <hello.h>

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
    {
    < variable declarations here >

    SOM_MainProgram();
    /*  Registers an exit procedure to ensure cleanup upon
     *  normal program termination. */

    < rest of WinMain here >
    }
```

Notice that the following WndProc code handles three messages: WM_CREATE, WM_PAINT, and WM_DESTROY. It lets the Windows default procedure handle the rest. The "Hello" object is created in response to the Window creation message (WM_CREATE) and is freed in response to the Window destroy message (WM_DESTROY). The "sayHello" method's greeting is displayed each time the window receives a WM_PAINT message. Notice that "obj" is declared static so that it persists between calls.

```
long FAR PASCAL _export WndProc (HWND hwnd, UINT message,
                                 UINT wParam, LONG lParam)
    {
    HDC         hdc ;
    PAINTSTRUCT ps ;
    RECT        rect ;
    char sBuf [80];
    static Hello obj;

    switch (message)
        {
        case WM_CREATE:
             obj = HelloNew();
             return 0;

        case WM_PAINT:
             hdc = BeginPaint (hwnd, &ps) ;

             GetClientRect (hwnd, &rect) ;

             strcpy(sBuf, _sayHello(obj,
                                somGetGlobalEnvironment()));
             DrawText (hdc, sBuf, -1, &rect,
                       DT_SINGLELINE | DT_CENTER | DT_VCENTER);

             EndPaint (hwnd, &ps) ;
             return 0 ;
```

```
                  case WM_DESTROY:
                      _somFree(obj);
                      PostQuitMessage (0) ;
                      return 0 ;
              }
         return DefWindowProc (hwnd, message, wParam, lParam) ;
         }
```

A C++ programmer on Windows would write a "main.cpp" file that uses the bindings defined in the "hello.xh" and "windows.xh" header files, with corresponding WinMain and WndProc code.

5) Compile and link the client code with the class implementation.

Note: On AIX or OS/2, the environment variable SOMBASE represents the directory in which SOM has been installed.

Under AIX, for C programmers:

```
> xlc -I. -I$SOMBASE/include -o hello main.c hello.c \
      -L$SOMBASE/lib -lsomtk
```

Under AIX, for C++ programmers:

```
> xlC -I. -I$SOMBASE/include -o hello main.C hello.C \
      -L$SOMBASE/lib -lsomtk
```

Under OS/2, for C programmers:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include -Fe hello main.c hello.c \
      somtk.lib
```

Under OS/2, for C++ programmers:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I\%SOMBASE%\include -Fe hello main.cpp hello.cpp \
      somtk.lib
```

Under Windows, for C programmers:

```
> cl -AL -Zp -D_WIN16 -F 5 -I. -I%SOMBASE%\include \
     -Fehello main.c hello.c llibcew.lib libw.lib somtk.lib
```

where %SOMBASE% is replaced by the directory in which SOM was installed (the default is c:\som). In makefiles, the expression $(SOMBASE) can be used. Note that the –F option to set the stack size is unnecessary if STACKSIZE is specified in a .def file.

Important: In this manual, compile and link commands are demonstrated for Windows using the Visual C++ compiler. Sample Makefiles shipped with SOMobjects For Windows also show how to build with other compilers.

Under Windows, for C++ programmers:

```
> cl -ALu -Zp -D_WIN16 -F 5 -I. -I%SOMBASE%\include \
     -Fehello main.cpp hello.cpp llibcew.lib libw.lib somtk.lib
```

where %SOMBASE% is replaced by the directory in which SOM was installed (the default is c:\som). In makefiles, the expression $(SOMBASE) can be used. Note that the –F option to set the stack size is unnecessary if STACKSIZE is specified in a .def file.

6) Execute the client program.

```
> hello
Hello, World!
```

Example 2 will extend the "Hello" class to introduce an "attribute".

File extensions for SOM files

- **IDL source file:**
  .idl   for all users

- **Implementation template file:**
  .c   for C, all systems
  .C   for C++, on AIX
  .cpp   for C++, on OS/2
                          or Windows

- **Header file for implementation file:**
  .ih   for C
  .xih   for C++

- **Header file for program file:**
  .h   for C
  .xh   for C++

Example 2 — Adding an Attribute to the Hello class

Example 1 introduced a class "Hello" which has a method "sayHello" that prints the fixed string "Hello, World!" Example 2 extends the "Hello" class so that clients can customize the output from the method "sayHello".

1) Modify the interface declaration for the class definition in "hello.idl."

Class "Hello" is extended by adding an attribute that we call "msg". Declaring an *attribute* is equivalent to defining "get" and "set" methods. For example, specifying:

```
attribute string msg;
```

is equivalent to defining the two methods:

```
string _get_msg();
void  _set_msg(in string msg);
```

Thus, for convenience, an attribute can be used (rather than an instance variable) in order to use the automatically defined "get" and "set" methods <u>without having to write their method procedures</u>. The new interface specification for "Hello" that results from adding attribute "msg" to the "Hello" class is as follows (with some comment lines omitted):

```
#include <somobj.idl>

interface Hello : SOMObject
{
  void  sayHello();

  attribute string msg;
   //# This is equivalent to defining the methods:
   //#     string _get_msg();
   //#      void  _set_msg(string msg);
};
```

2) Re-run the SOM Compiler on the updated .idl file, as in example 1. This produces new header files and updates the existing implementation file, if needed, to reflect changes made to the .idl file. In this example, the implementation file is not modified by the SOM Compiler.

3) Customize the implementation.

Customize the implementation file by modifying the print statement in the "sayHello" method procedure. This example prints the contents of the "msg" attribute (which must be initialized in the client program) by invoking the "_get_msg" method. Notice that, because the "_get_msg" method name begins with an underscore, the method is invoked with *two* leading underscores (for C only).

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

    printf("%s\n", __get_msg(somSelf, ev));

    /* for C++, use  somSelf->_get_msg(ev);  */
}
```

This implementation assumes that "_set_msg" has been invoked to initialize the "msg" attribute before the "_get_msg" method is invoked by the "sayHello" method. This initialization can be done within the client program.

4) Update the client program.

Modify the client program so that the "_set_msg" method is invoked to initialize the "msg" attribute before the "sayHello" method is invoked. Notice that, because the "_set_msg" method name begins with an underscore, the C client program invokes the method with *two* leading underscores.

For C programmers:

```
#include <hello.h>

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

    obj = HelloNew();

    /* Set the msg text */
    __set_msg(obj, somGetGlobalEnvironment(), "Hello World Again");

    /* Execute the "sayHello" method */
    _sayHello(obj, somGetGlobalEnvironment());

    _somFree(obj);
    return (0);
}
```

For C++ programmers:

```
#include <hello.xh>

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

    /* Set the msg text */
    obj->_set_msg(somGetGlobalEnvironment(), "Hello World Again");

    /* Execute the "sayHello" method */
    obj->sayHello(somGetGlobalEnvironment());

    obj->somFree();
    return (0);
}
```

5) Compile and link the client program, as before.

6) Execute the client program:

```
> hello
Hello World Again
```

The next example extends the "Hello" class to override (redefine) one of the methods it inherits from its parent class, **SOMObject**.

Attributes vs instance variables

As an alternative to defining "msg" as an attribute, an instance variable "message" could be introduced, with "set_msg" and "get_msg" methods defined for setting and retrieving its value. Instance variables are declared in an **implementation** statement, as shown below:

```
interface Hello
{
  string get_msg() ;
  void set_msg(in string msg);

#ifdef __SOMIDL__
implementation
{
    string message;
};
#endif
};
```

As demonstrated in this example, one disadvantage to using an instance variable is that the "get_msg" and "set_msg" methods must be defined in the implementation file by the class implementor. For attributes, by contrast, default implementations of the "get" and "set" methods are generated automatically by the SOM Compiler in the .ih and .xih header files.

Note: For some attributes (particularly those involving structures, strings, and pointers) the default implementation generated by the SOM Compiler for the "set" method may not be suitable. This happens because the SOM Compiler only performs a "shallow copy," which typically is not useful for distributed objects with these types of attributes. In such cases, it is possible to write your own implementations, as you do for any other method, by specifying the "noset/noget" modifiers for the attribute. (See the subtopic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler.")

Regardless of whether you let the SOM Compiler generate your implementations or not, if access to instance data is required, either from a subclass or a client program, then this access should be facilitated by using an attribute. Otherwise, instance data can be defined in the "implementation" statement as above (using the same syntax as used to declare variables in C or C++), with appropriate methods defined to access it. For more information about "implementation" statements, see the topic "Implementation statements" in Chapter 4.

As an example where instance variables would be used (rather than attributes), consider a class "Date" that provides a method for returning the current date. Suppose the date is represented by three instance variables — "mm", "dd", and "yy". Rather than making "mm", "dd", and "yy" attributes (and allowing clients to access them directly), "Date" defines "mm", "dd", and "yy" as instance variables in the "implementation" statement, and defines a method "get_date" that converts "mm", "dd", and "yy" into a string of the form "mm/dd/yy":

```
interface Date
{
    string get_date() ;

#ifdef __SOMIDL__
implementation
{
    long mm,dd,yy;
};
#endif
};
```

To access instance variables that a class introduces from within the class implementation file, two forms of notation are available:

somThis–>variableName

or

_variableName

For example, the implementation for "get_date" would likely

access the "mm" instance variable as `somThis->mm` or `_mm`,
access "dd" as `somThis->dd` or `_dd`, and
access "yy" as `somThis->yy` or `_yy`.

In C++ programs, the *_variableName* form is available only if the programmer first defines the macro VARIABLE_MACROS (that is, enter `#define VARIABLE_MACROS`) in the implementation file prior to including the .xih file for the class.

Example 3 — Overriding an Inherited Method

An important aspect of OOP programming is the ability of a subclass to replace an inherited method implementation with a new implementation especially appropriate to its instances. This is called *overriding* a method. Sometimes, a class may introduce methods that every descendant class is expected to override. For example, **SOMObject** introduces the **somPrintSelf** method, and a good SOM programmer will generally override this method when implementing a new class.

The purpose of **somPrintSelf** is to print a brief description of an object. The method can be useful when debugging an application that deals with a number of objects of the same class — assuming the class designer has overridden **somPrintSelf** with a message that is useful in distinguishing different objects of the class. For example, the implementation of **somPrintSelf** provided by **SOMObject** simply prints the class of the object and its address in memory. **SOMClass** overrides this method so that, when **somPrintSelf** is invoked on a class object, the name of the class will print.

This example illustrates how **somPrintSelf** might be overridden for the "Hello" class. An important identifying characteristic of "Hello" objects is the message they hold; thus, the following steps illustrate how **somPrintSelf** could be overridden in "Hello" to provide this information.

1) Modify the interface declaration in "hello.idl."

To override the **somPrintSelf** method in "Hello", additional information must be provided in "hello.idl" in the form of an **implementation** statement, which gives extra <u>information about the class, its methods and attributes, and any instance variables</u>. (The previous examples omitted the optional "implementation" statement, because it was not needed.)

In the current example, the "implementation" statement introduces the *modifiers* for the "Hello" class, as follows.

```
#include <somobj.idl>

interface Hello : SOMObject
{
  void sayHello();

  attribute string msg;

#ifdef __SOMIDL__
implementation
{
    //# Method Modifiers:
    somPrintSelf: override;
    // Override the inherited implementation of somPrintSelf.
};
#endif

};
```

Here, "somPrintSelf:" introduces a list of *modifiers* for the (inherited) **somPrintSelf** method in the class "Hello". Modifiers are like C/C++ #pragma commands and give specific implementation details to the compiler. This example uses only one modifier, "override". Because of the "override" modifier, when **somPrintSelf** is invoked on an instance of class "Hello", Hello's implementation of **somPrintSelf** (defined in the implementation file) will be called, instead of the implementation inherited from the parent class, **SOMObject**.

The "#ifdef _ _SOMIDL_ _" and "#endif" are standard C and C++ preprocessor commands that cause the "implementation" statement to be read only when using the SOM IDL compiler (and not some other IDL compiler).

2) Re-run the SOM Compiler on the updated .idl file, as before. The SOM Compiler extends the existing implementation file from Example 2 to include new stub procedures as needed (in this case, for **somPrintSelf**). Below is a shortened version of the C language implementation file as updated by the SOM Compiler; C++ implementation files are similarly revised. Notice that the code previously added to the "sayHello" method is not disturbed when the SOM Compiler updates the implementation file.

```
#include <hello.ih>

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

    printf("%s\n", __get_msg(somSelf, ev));
}

SOM_Scope void   SOMLINK somPrintSelf(Hello somSelf)
{
    HelloData *somThis = HelloGetData(somSelf);
    HelloMethodDebug("hello","somPrintSelf");

    Hello_parent_SOMObject_somPrintSelf(somSelf);
}
```

Note that the SOM Compiler added code allowing the "Hello" class to redefine **somPrintSelf**. The SOM Compiler provides a default implementation for overriding the **somPrintSelf** method. This default implementation simply calls the "parent method" [the procedure that the parent class of "Hello" (**SOMObject**) uses to implement the **somPrintSelf** method]. This parent method call is accomplished by the macro `Hello_parent_SOMObject_somPrintSelf`, defined in "hello.ih."

Notice that the stub procedure for overriding the **somPrintSelf** method does not include an Environment parameter. This is because **somPrintSelf** is introduced by **SOMObject**, which does not include the Environment parameter in any of its methods (to ensure backward compatibility). The signature for a method cannot change after it has been introduced.

3) Customize the implementation.

Within the new **somPrintSelf** method procedure, display a brief description of the object, appropriate to "Hello" objects. Note that the parent method call is not needed, so it has been deleted. Also, direct access to instance data introduced by the "Hello" class is not required, so the assignment to "somThis" has been commented out (see the first line of the procedure).

```
SOM_Scope void   SOMLINK somPrintSelf(Hello somSelf)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","somPrintSelf");

    somPrintf("-- a %s object at location %X with msg: %s\n",
              _somGetClassName(somSelf),
              somSelf,
              __get_msg(somSelf,0));
}
```

4) Update the client program to illustrate the change (also notice the new message text):

For C programmers:

```
#include <hello.h>

int main(int argc, char *argv[])
{
    Hello obj;
    Environment *ev = somGetGlobalEnvironment();

    obj = HelloNew();

    /* Set the msg text */
    __set_msg(obj, ev, "Hi There");

    /* Execute the "somPrintSelf" method */
    _somPrintSelf(obj);

    _somFree(obj);
    return (0);
}
```

For C++ programmers:

```
#include <hello.xh>

int main(int argc, char *argv[])
{
    Hello *obj;
    Environment *ev = somGetGlobalEnvironment();

    obj = new Hello;

    /* Set the msg text */
    __set_msg(obj, ev, "Hi There");

    /* Execute the "somPrintSelf" method */
     obj->somPrintSelf();

    obj->somFree();
    return (0);
}
```

5) Compile and link the client program, as before.

6) Execute the client program, which now outputs the message:

```
> hello
 -- a Hello object at location 20062838 with msg: Hi There
```

Example 4 — Initializing a SOM Object

The previous example showed how to override the method **somPrintSelf**, introduced by **SOMObject**. As mentioned in that example, **somPrintSelf** should generally be overridden when implementing a new class. Another method introduced by **SOMObject** that should generally be overridden is **somDefaultInit**. The purpose of **somDefaultInit** is to provide a "default" *initializer* for the instance variables introduced by a class.

This example shows how to override **somDefaultInit** to give each "Hello" object's message an initial value when the object is first created. To learn more about initializers than shown in this example (including how to introduce new initializers that take arbitrary arguments, and how to explicitly invoke initializers) read Section 5.5, "Initializing and Uninitializing Objects," in Chapter 5, "Implementing Classes in SOM."

The overall process of overriding **somDefaultInit** is similar to that of the previous example. First, the IDL for "Hello" is modified. In addition to an **override** modifier, an **init** modifier is used to indicate that a stub procedure for an initialization method is desired (the stub procedures for initializers are different from those of normal methods).

1) Modify the interface declaration in "hello.idl."

```
#include <somobj.idl>

interface Hello : SOMObject
{
  void sayHello();

  attribute string msg;

#ifdef __SOMIDL__
implementation
{
    //# Method Modifiers:
    somPrintSelf: override;
    somDefaultInit: override, init;
};
#endif

};
```

2) Re-run the SOM Compiler on the updated hello.idl file, as before. The SOM Compiler extends the existing implementation file. Below is the initializer stub procedure that the SOM Compiler adds to the C language implementation file; C++ implementation files would be similarly revised:

```
SOM_Scope void SOMLINK
                somDefaultInit(Hello somSelf, somInitCtrl *ctrl)
{
  HelloData *somThis; /* set by BeginInitializer */
  somInitCtrl globalCtrl;
  somBooleanVector myMask;
  HelloMethodDebug("Hello", "somDefaultInit");
  Hello_BeginInitializer_somDefaultInit;

  Hello_Init_SOMObject_somDefaultInit(somSelf, ctrl);
  /*
   * local Hello initialization code added by programmer
   */
}
```

3) Customize the implementation.

Here, the "msg" instance variable is set in the implementation template (rather than in the client program, as before). Thus, the "msg" is defined as part of the "Hello" object's initialization.

```
SOM_Scope void SOMLINK
                 somDefaultInit(Hello somSelf, somInitCtrl *ctrl)
{
  HelloData *somThis; /* set by BeginInitializer */
  somInitCtrl globalCtrl;
  somBooleanVector myMask;
  HelloMethodDebug("Hello", "somDefaultInit");
  Hello_BeginInitializer_somDefaultInit;

  Hello_Init_SOMObject_somDefaultInit(somSelf, ctrl);
  /*
   * local Hello initialization code added by programmer
   */
  __set_msg(somSelf, "Initial Message");
}
```

4) Update the client program to illustrate default initialization.

```
#include <hello.h>
main()
{
    Hello h = HelloNew();

    /* Execute the "somPrintSelf" method */
    _somPrintSelf(h);
}
```

5) Compile and link the client program, as before.

6) Execute the client program:

```
> hello
-- a Hello object at 200633A8 with msg: Initial Message
```

Example 5 — Using Multiple Inheritance

The "Hello" class is useful for writing messages to the screen. So that clients can also write messages to printers and disk files, this example references two additional classes: "Printer" and "Disk". The "Printer" class will manage messages to a printer, and the "Disk" class will manage messages sent to files. These classes can be defined as follows:

```
#include <somobj.idl>

interface Printer : SOMObject
{
    void stringToPrinter(in string s) ;
    // This method writes a string to a printer.
};

#include <somobj.idl>

interface Disk : SOMObject
{
    void stringToDisk(in string s) ;
    // This method writes a string to disk.

};
```

This example assumes the "Printer" and "Disk" classes are defined separately (in "print.idl" and "disk.idl", for example), are implemented in separate files, and are linked with the other example code. Given the implementations of the "Printer" and "Disk" interfaces, the "Hello" class can use them by inheriting from them, as illustrated next.

1) Modify the interface declaration in "hello.idl".

```
#include <somcls.idl>
#include <disk.idl>
#include <printer.idl>

interface Hello : Disk, Printer
{
    void sayHello();

    attribute string msg;

    enum outputTypes {screen, printer, disk};
    // Declare an enumeration for the different forms of output

    attribute outputTypes output;
    // The current form of output

#ifdef __SOMIDL__

   implementation {
        somDefaultInit: override, init;
   };
#endif //#  __SOMIDL__
};
```

Notice that **SOMObject** is not listed as a parent of "Hello" above, because **SOMObject** is a parent of "Disk" (and of "Printer").

The IDL specification above declares an enumeration "outputTypes" for the different forms of output, and an attribute "output" whose value will depend on where the client wants the output of the "sayHello" method to go.

Note: SOM IDL allows the use of structures, unions (though with a syntax different from C or C++), enumerations, constants, and typedefs, both inside and outside the body of an interface statement. Declarations that appear inside an interface body will be emitted in the header file

(that is, in "hello.h" or "hello.xh"). Declarations that appear outside of an interface body do not appear in the header file (unless required by a special #pragma directive; see the SOM Compiler options in Chapter 4).

SOM IDL also supports all of the C and C++ preprocessor directives, including conditional compilation, macro processing, and file inclusion.

2) Re-run the SOM Compiler on the updated .idl file.

Unfortunately, when this is done, the implementation for **somDefaultInit** is not correctly updated to reflect the addition of two new parents to "Hello." This is because the implementation-file emitter never changes the bodies of existing method procedures. As a result, method procedures for initializer methods are *not* given new parent calls when the parents of a class are changed. One way to deal with this (when the parents of a class are changed) is to temporarily rename the method procedures for initializer methods, and then run the implementation emitter. Once this is done, the code in the renamed methods can be merged into the new templates, which will include all the appropriate parent method calls. When this is done here, the new implementation for **somDefaultInit** would appear as:

```
SOM_Scope void SOMLINK
                somDefaultInit(Hello somSelf, somInitCtrl *ctrl)
{
  HelloData *somThis; /* set by BeginInitializer */
  somInitCtrl globalCtrl;
  somBooleanVector myMask;
  HelloMethodDebug("Hello", "somDefaultInit");
  Hello_BeginInitializer_somDefaultInit;

  Hello_Init_Disk_somDefaultInit(somSelf, ctrl);
  Hello_Init_Printer_somDefaultInit(somSelf, ctrl);
  /*
   * local Hello initialization code added by programmer
   */
  __set_msg(somSelf, "Initial Message");
}
```

3) Continue to customize the implementation file, hello.c, as follows. Notice that the "sayHello" method (last discussed in Example 2) is now modified to allow alternate ways of outputting a "msg".

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

    switch (__get_output(somSelf, ev) ) {
            /*  for C++, use:  somSelf->_get_output(ev) */
    case Hello_screen:
        printf("%s\n", __get_msg(somSelf, ev) );
            /*  for C++, use:  somSelf->_get_msg(ev) */
        break;
    case Hello_printer:
        _stringToPrinter(somSelf, ev, __get_msg(somSelf, ev) );
            /* for C++, use:
             * somSelf->stringToPrinter(ev, somSelf->_get_msg(ev) );
             */
        break;
    case Hello_disk:
        _stringToDisk(somSelf, ev, __get_msg(somSelf, ev) );
            /*  for C++, use:
             *   somSelf->stringToDisk(ev, somSelf->_get_msg(ev) );
             */
        break;
    }
}
```

The "switch" statement invokes the appropriate method depending on the value of the "output" attribute. Notice how the "case" statements utilize the enumeration values of "outputTypes" declared in "hello.idl" by prefacing the enumeration names with the class name (Hello_screen, Hello_printer, and Hello_disk).

4) Update the client program, as illustrated next.

```
#include <hello.h>
        /* for C++, use "hello.xh" and <stdio.h> */

int main(int argc, char *argv[])
{
    Hello a = HelloNew();
    Environment *ev = somGetGlobalEnvironment();

    /* Invoke "sayHello" on an object and use each output */
    _sayHello(a, ev) ;           /* for C++, use:  a->sayHello(ev);  */
    __set_output(a, ev, Hello_printer) ;
                    /* C++:   a->_set_output(ev, Hello_printer); */
    _sayHello(a, ev) ;
    __set_output(a, ev, Hello_disk) ;
                    /* C++:   a->_set_output(ev, Hello_disk);  */
    _sayHello(a, ev) ;

    _somFree(a) ;  /* for C++, use: a->somFree(); */
    return (0);
}
```

5) Compile and link the client program as before, except also include the implementation files for the "Printer" and "Disk" classes in the compilation.

6) Execute the client program. Observe that the message that prints is the "msg" defined in Example 4 as part of the **somDefaultInit** initialization of the "Hello" object.

```
Initial Message
Initial Message            − goes to a Printer
Initial Message            − goes to Disk
```

This tutorial has described features of SOM IDL that will be useful to C and C++ programmers. SOM IDL also provides features such as full type checking, constructs for declaring private methods, and constructs for defining methods that receive and return pointers to structures. Chapter 4, "SOM IDL and the SOM Compiler," gives a complete description of the SOM IDL syntax and also describes how to use the SOM Compiler. In addition, Chapter 5, "Implementing Classes in SOM," provides helpful information for completing the implementation template, for using initializers (**somDefaultInit** or user-defined initialization methods), for defining SOM class libraries, and for customizing various aspects of SOMobjects execution.

Chapter 3. Using SOM Classes in Client Programs

This chapter discusses how to use SOM classes that have already been fully implemented. That is, these topics describe the steps that a programmer uses to instantiate an object and invoke some method(s) on it from within an application program.

Who should read this chapter?

- Programmers who wish to use SOM classes that were originally developed by someone else will need to know the information in this chapter. These programmers often may not need the information from any subsequent chapters.

- By contrast, class implementors who are creating their own SOM classes should continue with Chapter 4, "SOM IDL and the SOM Compiler," and Chapter 5, "Implementing Classes in SOM," for complete information on the SOM Interface Definition Language (SOM IDL) syntax and other details of class implementation.

Programs that use a class are referred to as *client programs.* A client program can be written in C, in C++, or in another language. As noted, this chapter describes how client programs can use SOM classes (classes defined using SOM, as described in Chapter 2, "Tutorial for Implementing SOM Classes," in Chapter 4, "SOM IDL and the SOM Compiler," and in Chapter 5, "Implementing Classes in SOM"). Using a SOM class involves creating instances of a class, invoking methods on objects, and so forth. All of the methods, functions, and macros described here can also be used by class implementors within the implementation file for a class.

Note: "Using a SOM class," as described in this chapter, does *not* include subclassing the class in a client program. In particular, the C++ compatible SOM classes made available in the .xh binding file *cannot* be subclassed in C++ to create new C++ or SOM classes.

Some of the macros and functions described here are supplied as part of SOM's C and C++ *usage bindings*. These bindings are functions and macros defined in header files to be included in client programs. The usage bindings make it more convenient for C and C++ programmers to create and use instances of SOM classes. SOM classes can be also used without the C or C++ bindings, however. For example, users of other programming languages can use SOM classes, and C and C++ programmers can use a SOM class without using its language bindings. The language bindings simply offer a more convenient programmer's interface to SOM. Vendors of other languages may also offer SOM bindings; check with your language vendor for possible SOM support.

To use the C or C++ bindings for a class, a client program must include a header file for the class (using the **#include** preprocessor directive). For a C language client program, the file *<classFileStem>*.h must be included. For a C++ language client program, the file *<classFileStem>*.xh must be included. The SOM Compiler generates these header files from an IDL interface definition. The header files contain definitions of the macros and functions that make up the C or C++ bindings for the class. Whether the header files include bindings for the class's private methods and attributes (in addition to the public methods and attributes) depends on the IDL interface definition available to the user, and on how the SOM Compiler was invoked when generating bindings.

Usage binding headers automatically include any other bindings upon which they may rely. Client programs not using the C or C++ bindings for any particular class of SOM object (for example, a client program that does not know at compile time what classes it will be using) should simply include the SOM-supplied bindings for **SOMObject**, provided in the header file "somobj.h" (for C programs) or "somobj.xh" (for C++ programs).

For each task that a user of a SOM class might want to perform, this chapter shows how the task would be accomplished by:

- a C programmer using the C bindings,
- a C++ programmer using the C++ bindings, or
- a programmer not using SOM's C or C++ language bindings.

If neither of the first two approaches is applicable, the third approach can always be used.

3.1 An Example Client Program

Following is a C program that uses the class "Hello" (as defined in the Tutorial in Chapter 2). The "Hello" class provides one attribute, "msg", of type string, and one method, "sayHello". The "sayHello" method simply displays the value of the "msg" attribute of the object on which the method is invoked.

```
#include <hello.h>  /* include the header file for Hello */

int main(int argc, char *argv[])
{
    /* declare a variable (obj) that is a
     * pointer to an instance of the Hello class: */
    Hello obj;

    /* create an instance of the Hello class
     * and store a pointer to it in obj: */
    obj = HelloNew();

    /* invoke method _set_msg on obj with the argument
     * "Hello World Again". This method sets the value of
     * obj's 'msg' attribute to the specified string.
     */
    __set_msg(obj, somGetGlobalEnvironment(), "Hello World Again");

    /* invoke method sayHello on obj. This method prints
     * the value of obj's 'msg' attribute.  */
    _sayHello(obj, somGetGlobalEnvironment());

    return(0);
}
```

The C++ version of the foregoing client program is shown below:

```
#include <hello.xh>  /* include the header file for Hello */

int main(int argc, char *argv[])
{
    /* declare a variable (obj) that is a
     * pointer to an instance of the Hello class: */
    Hello *obj;

    /* create an instance of the Hello class
     * and store a pointer to it in obj: */
    obj = new Hello;

    /* invoke method _set_msg on obj with the argument
     * "Hello World Again". This method sets the value of
     * obj's 'msg' attribute to the specified string.  */
    obj->_set_msg(somGetGlobalEnvironment(), "Hello World Again");

    /* invoke method sayHello on obj. This method prints
     * the value of obj's 'msg' attribute.  */
    obj->sayHello(somGetGlobalEnvironment());

    return(0);
}
```

These client programs both produce the output:

```
Hello World Again
```

3.2 Using SOM Classes — the Basics

This section describes the basic information needed to use SOM classes in a client program.

Declaring object variables

To declare an object variable, the name of an *object interface* defined in IDL is used as the *type* of the variable. The exact syntax is slightly different for C vs. C++ programmers. Specifically,

<interfaceName> obj ;   in C programs or
<interfaceName> *obj ;   in C++ programs

declares "obj" to be a pointer to an object that has type *<interfaceName>*. In SOM, objects of this type are instances of the SOM *class* named *<interfaceName>*, or of any SOM class derived from this class. Thus, for example,

```
Animal obj;          in C programs or
Animal *obj;         in C++ programs
```

declares "obj" as pointer to an object of type "Animal" that can be used to reference an instance of the SOM class "Animal" or any SOM class derived from "Animal". Note that the type of an object need not be the same as its class; an object of type "Animal" might not be an instance of the "Animal" class (rather, it might be an instance of some subclass of "Animal" — the "Cat" class, perhaps).

All SOM objects are of type **SOMObject**, even though they may not be instances of the **SOMObject** class. Thus, if it is not known at compile time what type of object the variable will point to, the following declaration can be used:

```
SOMObject obj;         in C programs or
SOMObject *obj;        in C++ programs.
```

Because the sizes of SOM objects are not known at compile time, instances of SOM classes must always be dynamically allocated. Thus, a variable declaration must always define a pointer to an object.

Note: in the C usage bindings, as within an IDL specification, an interface name used as a type *implicitly* indicates a pointer to an object that has that interface (this is required by the CORBA specification). The C usage bindings for SOM classes therefore hide the pointer with a C typedef for *<interfaceName>*. But this is not appropriate in the C++ usage bindings, which define a C++ class for *<interfaceName>*. Thus, it is not correct in C++ to use a declaration of the form:

<interfaceName> obj ;   *not valid* in C++ programs

Note: If a C programmer also prefers to use explicit pointers to *<interfaceName>* types, then the SOM Compiler option **–maddstar** can be used when the C binding files are generated, and the explicit "∗" will then be required in declarations of object variables. (This option is required for compatibility with existing SOM OIDL code. For information on using the **–maddstar** option, see "Running the SOM Compiler" in Chapter 4, "SOM IDL and the SOM Compiler.")

Users of other programming languages must also define object variables to be pointers to the data structure used to represent SOM objects. The way this is done is programming-language dependent. The header file "somtypes.h" defines the structure of SOM objects for the C language.

Creating instances of a class

For C programmers with usage bindings, SOM provides the *<className>***New** and the *<className>***Renew** macros for creating instances of a class.

These macros are illustrated with the following two examples, each of which creates a single instance of class "Hello":

```
obj = HelloNew();
obj = HelloRenew(buffer);
```

Using *<className>*New

After verifying that the *<className>* class object exists, the *<className>***New** macro invokes the **somNew** method on the class object. This allocates enough space for a new instance of *<className>*, creates a new instance of the class, initializes this new object by invoking **somDefaultInit** on it, and then returns a pointer to it. The *<className>***New** macro automatically creates the class object for *<className>*, as well as its ancestor classes and metaclass, if these objects have not already been created.

After a client program has finished using an object created using the *<className>***New** macro, the object should be freed by invoking the method **somFree** on it :

```
_somFree(obj);
```

After uninitializing the object by invoking **somDestruct** on it, **somFree** calls the class object for storage deallocation. This is important because storage for an object created using the *<className>***New** macro is allocated by the class of the object. Thus, only the class of the object can know how to reclaim the object's storage.

Using *<className>*Renew

After verifying that the *<className>* class object exists, the *<className>***Renew** macro invokes the **somRenew** method on the class object. *<className>***Renew** is only used when the space for the object has been allocated previously. (Perhaps the space holds an old, uninitialized object that is not needed anymore.) This macro converts the given space into a new, initialized instance of *<className>* and returns a pointer to it. The programmer is responsible for ensuring that the argument of *<className>***Renew** points 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. Like *<className>***New,** the *<className>***Renew** macro automatically creates any required class objects that have not already been created.

Hint: When creating a large number of class instances, it may be more efficient to allocate at once enough memory to hold all the instances, and then invoke *<className>***Renew** once for each object to be created, rather than performing separate memory allocations.

Using *<className>*NewClass

The C and C++ usage bindings for a SOM class also provide static linkage to a *<className>***NewClass** function that can be used to create the class object. This can be useful if the class object is needed before its instances are created.

For example, the following C code uses the function **HelloNewClass** to create the "Hello" class object. The arguments to this function are defined by the usage bindings, and indicate the version of the class implementation that is assumed by the bindings. (For more detail on creation of classes, see the later topic, "Creating a 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 uses the **HelloRenew** macro to create ten instances of the "Hello" class:

```
#include <hello.h>
main()
{
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:
 * (somGetInstanceSize is a method provided by SOM.) */
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);
...
...
/* Uninitialize the objects and free them */
for (i=0; i<10; i++)
  _somDestruct(objA[i],0,0);
SOMFree(buffer);
}
```

When an object created with the *<className>***Renew** macro is no longer needed, its storage must be freed using the dual to whatever method was originally used to allocate the storage. Two method pairs are typical:

- For example, if an object was originally initialized using the *<className>***New** macro, then, as discussed previously, the client should use the **somFree** method on it.

- On the other hand, if the program uses the **SOMMalloc** function to allocate memory, as illustrated in the example above, then the **SOMFree** function must be called to free the objects' storage (because **SOMFree** is the dual to **SOMMalloc**). Before this is done, however, the objects in the region to be freed should be deinitialized by invoking the **somDestruct** method on them. This allows each object to free any memory that it may have been allocated without the programmer's knowledge. (The **somFree** method also calls the **somDestruct** method.)

    Note: In the **somDestruct** method call above, the first zero indicates that memory should not be freed by the class of the object (that is, the programmer will do it explicitly). The second zero indicates that the class of the object is responsible for overall control of object uninitialization. For further discussion, see Section 5.5, "Initializing and Uninitializing Objects," in Chapter 5, "Implementing Classes in SOM."

For C++ programmers with usage bindings, instances of a class *<className>* can be created with a **new** operator provided by the usage bindings of each SOM class. The **new** operator automatically creates the class object for *<className>*, as well as its ancestor classes and metaclass, if they do not yet exist. After verifying the existence of the desired class object, the **new** operator then invokes the **somNewNoInit** method on the class. This allocates memory and creates a new instance of the class, but it does *not* initialize the new object.

Initialization of the new object is then performed using one of the C++ constructors defined by the usage bindings. (For further discussion, see Section 5.5, "Initializing and Uninitializing Objects," in Chapter 5, "Implementing Classes in SOM.") Two variations of the **new** operator require no arguments. When either is used, the C++ usage bindings provide a default constructor that invokes the **somDefaultInit** method on the new object. Thus, a new object initialized by **somDefaultInit** would be created using either of the forms:

> **new** *<className>*

> **new** *<className>*()

For example:

```
obj = new Hello;

obj1 = new Hello();
```

SOM objects created using the **new** operator should be freed using the **delete** operator, just as for normal C++ objects:

```
delete obj;
```

When previously allocated space will be used to hold a new object, C++ programmers should use the **somRenew** method, described below. C++ bindings do not provide a macro for this purpose.

somNew and somRenew: C and C++ programmers, as well as programmers using other languages, can create instances of a class using the SOM methods **somNew** and **somRenew**, invoked on the class object. As discussed and illustrated above for the C bindings, the class object must first be created using the *<className>***NewClass** procedure (or, perhaps, using the **somFindClass** method — see the section "Using class objects" later in this chapter).

The **somNew** method invoked on the class object creates a new instance of the class, initializes the object using **somDefaultInit**, and then returns a pointer to the new object. For instance, the following C example creates a new object of the "Hello" class.

```
#include <hello.h>
main()
{
  SOMClass helloCls;    /* a pointer to the Hello class */
  Hello obj;            /* a pointer to a Hello instance */

  /* create the Hello class  */
  helloCls = HelloNewClass(Hello_MajorVersion, Hello_MinorVersion);
  obj = _somNew(helloCls); /* create the Hello instance */
}
```

An object created using the **somNew** method should be freed by invoking the **somFree** method on it after the client program is finished using the object.

The **somRenew** method invoked on the class object creates a new instance of a class using the given space, rather than allocating new space for the object. The method converts the given space into an instance of the class, initializes the new object using **somDefaultInit**, and then returns a pointer to the new object. The argument to **somRenew** must point to a block of storage large enough to hold the new instance. The method **somGetInstanceSize** can be used to determine the amount of memory required. For example, the following C++ code creates ten instances of the "Hello" class:

```
#include <hello.xh>
#include <somcls.xh>
main()
{
  SOMClass *helloCls; // a pointer to the Hello class
  Hello *objA[10];    //  an array of Hello instance pointers
  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 = helloCls->somGetInstanceSize();
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 objects
for (i=0; i<10; i++)
   objA[i] = helloCls->somRenew(buffer+i*size);

 ...

// Uninitialize the objects and free them
 for (i=0; i<10; i++)
   objA[i]->somDestruct(0,0);
 SOMFree(buffer);
}
```

The **somNew** and **somRenew** methods are useful for creating instances of a class when the header file for the class is not included in the client program at compile time. (The name of the class might be specified by user input, for example.) However, the <*className*>**New** macro (for C) and the **new** operator (for C++) can *only* be used for classes whose header file is included in the client program at compile time.

Objects created using the **somRenew** method should be freed by the client program that allocated it, using the dual to whatever allocation approach was initially used. If the **somFree** method is not appropriate (because the **somNew** method was not initially used), then, before memory is freed, the object should be explicitly deinitialized by invoking the **somDestruct** method on it. (The **somFree** method calls the **somDestruct** method. Refer to the previous C example for **Renew** for an explanation of the arguments to **somDestruct**.)

Invoking methods on objects

This topic describes the general way to invoke methods in C/C++ and in other languages, and then presents subtopics for more specialized situations.

Making typical method calls

For C programmers with usage bindings: To invoke a method in C, use the macro:

  _<*methodName*> (*receiver, args*)

(that is, an underscore followed by the method name). Arguments to the macro are the receiver of the method followed by all of the arguments to the method. For example:

```
_foo(obj, somGetGlobalEnvironment(), x, y);
```

This invokes method "foo" on "obj" (the remaining arguments are arguments to the method "foo"). This expression can be used anywhere that a standard function call can be used in C.

Required arguments

In C, calls to methods defined using IDL require at least two arguments — a pointer to the *receiving object* (the object responding to the method) and a value of type (**Environment ***). The **Environment** data structure is specified by CORBA, and is used to pass environmental information between a caller and a called method. For example, it is used to return exceptions. (For more information on how to supply and use the **Environment** structure, see the section entitled "Exceptions and error handling" later in this chapter.)

In the IDL definition of a method, by contrast, the receiver and the **Environment** pointer are *not* listed as parameters to the method. (Unlike the receiver, the **Environment** pointer is considered a method parameter, even though it is never explicitly specified in IDL. For this reason, it is called an *implicit* method parameter.) For example, if a method is defined in a .idl file with two parameters, as in:

```
int foo (in char c, in float f);
```

then, with the C usage bindings, the method would be invoked with four arguments, as in:

```
intvar = _foo(obj, somGetGlobalEnvironment(), x, y);
```

where "obj" is the object responding to the method and "x" and "y" are the arguments corresponding to "c" and "f", above.

If the IDL specification of the method includes a *context* specification, then the method has an additional (implicit) **context** parameter. Thus, 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 prescribed by the CORBA standard.

If the IDL specification of the class that introduces the method includes the **callstyle=oidl** modifier, then 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 (if any). Some of the classes supplied in the SOMobjects Toolkit (including **SOMObject, SOMClass**, and **SOMClassMgr**) are defined in this way, to ensure compatibility with the previous release of SOM. The *SOMobjects Developer Toolkit Programmers Reference Manual* specifies for each method whether these arguments are used.

If you use a C expression to compute the first argument to a method call (the receiver), you must use an expression without side effects, 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** can *not* be used as the first argument to a C method call, because doing so would create two new class instances rather than one.

Following the initial, required arguments to a method (the receiving object, the **Environment**, if any, and the **context**, if any), you enter any additional arguments required by that method, as specified in IDL. For a discussion of how IDL **in/out/inout** argument types map to C/C++ data types, see the topic "Parameter list" in Chapter 4, "SOM IDL and the SOM Compiler."

Short form vs long form

If a client program uses the bindings for two different classes that introduce or inherit two different methods of the same name, then the _<*methodName*> macro described above (called the *short form*) will not be provided by the bindings, because the macro would be ambiguous in that circumstance. The following *long form* macro, however, is always provided by the usage bindings for each class that supports the method:

<*className*>_<*methodName*> (*receiver, args*)

For example, method "foo" supported by class "Bar" can be invoked as:

```
Bar_foo(obj, somGetGlobalEnvironment(), x, y)        (in C)
```

where "obj" has type "Bar" and "x" and "y" are the arguments to method "foo".

In most cases (where there is no ambiguity, and where the method is not a **va_list** method, as described in the subsequent subtopic "Using 'va_list' methods"), a C programmer may use either the short or the long form of a method invocation macro interchangeably. However, _only the long form complies with the CORBA standard for C usage bindings_. If you wish to write code that can be easily ported to other vendor platforms that support the CORBA standard, use the long form exclusively. The long form is always available for every method that a class supports. The short form is provided both as a programming convenience and for source code compatibility with release 1 of SOM.

In order to use the long form, a programmer will usually know what type an object is expected to have. If this is not known, but the different methods have the same signature, the method can be invoked using name-lookup resolution, as described in a following subtopic of this section.

For C++ programmers with usage bindings: To invoke a method, use the standard C++ form shown below:

obj–>_<methodName>_ (_args_)

where _args_ are the arguments to the method. For instance, the following example invokes method "foo" on "obj":

```
obj–>foo(somGetGlobalEnvironment(), x, y)
```

Required arguments

All methods introduced by classes declared using IDL (except those having the SOM IDL **callstyle=oidl** modifier) have at least one parameter — a value of type (**Environment ***). The **Environment** data structure is used to pass environmental information between a caller and a called method. For example, it is used to return exceptions. For more information on how to supply and use the **Environment** structure, see the section entitled "Exceptions and error handling" later in this chapter.

The **Environment** pointer is an implicit parameter. That is, in the IDL definition of a method, the **Environment** pointer is _not_ explicitly listed as a parameter to the method. For example, if a method is defined in IDL with two explicit parameters, as in:

```
int foo (in char c, in float f);
```

then the method would be invoked from C++ bindings with three arguments, as in:

```
intvar = obj–>foo(somGetGlobalEnvironment(), x, y);
```

where "obj" is the object responding to the method and "x" and "y" are the arguments corresponding to "c" and "f", above.

If the IDL specification of the method includes a _context_ specification, then the method has a second implicit parameter, of type **context**, and the method must be invoked with an additional **context** argument. This argument must follow immediately after the **Environment** pointer argument. (No SOM-supplied methods require **context** arguments.) The **Environment** and **context** method parameters are prescribed by the CORBA standard.

If the IDL specification of the class that introduces the method includes the **callstyle=oidl** modifier, then the (**Environment ***) and **context** arguments should not be supplied when the method is invoked. Some of the classes supplied in the SOMobjects Toolkit (including **SOMObject, SOMClass**, and **SOMClassMgr**) are defined in this way, to ensure compatibility with the previous release of SOM. The _SOMobjects Developer Toolkit Programmers Reference Manual_ specifies for each method whether these arguments are used.

Following the initial, required arguments to a method (the receiving object, the **Environment**, if any, and the **context**, if any), you enter any additional arguments required by that method, as specified in IDL. For a discussion of how IDL **in/out/inout** argument types map to C/C++ data types, see the topic "Parameter list" in Chapter 4, "SOM IDL and the SOM Compiler."

For non-C/C++ programmers: To invoke a *static method* (that is, a method declared when defining an OIDL or IDL object interface) without using the C or C++ usage bindings, a programmer can use the **somResolve** procedure. The **somResolve** procedure takes as arguments a pointer to the object on which the method is to be invoked and a *method token* for the desired method. It returns a pointer to the method's procedure (or raises a fatal error if the object does not support the method). Depending on the language and system, it may be necessary to cast this procedure pointer to the appropriate type; the way this is done is language-specific.

The method is then invoked by calling the procedure returned by **somResolve**, passing the method's receiver, the **Environment** pointer (if necessary), the **context** argument (if necessary) and the remainder of the method's arguments, if any. Note that the means for calling a procedure, given a pointer returned by **somResolve**, is language-specific. (See the section above for C programmers. The arguments to a method procedure are the same as the arguments passed using the long form of the C-language method-invocation macro for that method.)

Using **somResolve** requires the programmer to know where to find the *method token* for the desired method. Method tokens are available from class objects that support the method (via the method **somGetMethodToken**), or from a global data structure, called the *ClassData structure,* corresponding to the class that introduces the method. In C and C++ programs with access to the definitions for ClassData structures provided by usage bindings, the method token for method *methodName* introduced by class *className* may be accessed by the following expression:

> *<className>***ClassData.***<methodName>*

For example, the method token for method "sayHello" introduced by class "Hello" is stored at location `HelloClassData.sayHello`, for C and C++ programmers. The way method tokens are accessed in other languages is language-specific.

As an example of using offset resolution to invoke methods from a programming language other than C/C++, one would do the following to create an instance of a SOM Class *X* in Smalltalk:

1. Initialize the SOM run-time environment, if it has not previously been initialized, using the **somEnvironmentNew** function.

2. If the class object for class *X* has not yet been created, use **somResolve** with arguments **SOMClassMgrObject** (returned by **somEnvironmentNew** in step 1) and the method token for the **somFindClass** method*,* to obtain a method procedure pointer for the **somFindClass** method. Use the method procedure for **somFindClass** to create the class object for class *X*: Call the procedure with arguments **SOMClassMgrObject**, the result of calling the **somIdFromString** function with argument *"X",* and the major and minor version numbers for class *X* (or zero). The procedure returns the class object for class *X*.

3. Use **somResolve** with arguments representing the class object for *X* (returned by **somFindClass** in step 2) and the method token for the **somNew** method, to obtain a method procedure pointer for method **somNew.** (The **somNew** method is used to create instances of class *X*.)

4. Call the method procedure for **somNew** (using the method procedure pointer obtained in step 3) with the class object for *X* (returned by **somFindClass** in step 3) as the argument. The procedure returns a new instance of class *X*.

In addition to **somResolve**, SOM also supplies the **somClassResolve** procedure. Instead of an object, the **somClassResolve** procedure takes a class as its first argument, and then selects a method procedure from the instance method table of the passed class. (The **somResolve** procedure, by contrast, selects a method procedure from the instance method table of the class of which the passed object is an instance.) The **somClassResolve** procedure therefore supports *casted* method resolution. See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on **somResolve** and **somClassResolve**.

If the programmer does not know at compile time which class introduces the method to be invoked, or if the programmer cannot directly access method tokens, then the procedure **somResolveByName** can be used to obtain a method procedure using name-lookup resolution, as described in the next section.

If the signature of the method to be invoked is not known at compile time, but can be discovered at run time, use **somResolve** or **somResolveByName** to get a pointer to the **somDispatch** method procedure, then use it to invoke the specific method, as described below under "Method name or signature not known at compile time."

Accessing attributes

In addition to methods, SOM objects can also have attributes. An *attribute* is an IDL shorthand for declaring methods, and 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. (Attributes that are declared as "readonly" have no "set" method, however.)

The "get" and "set" methods are invoked in the same way as other methods. For example, given class "Hello" with attribute "msg" of type **string**, the following code segments set and get the value of the "msg" attribute:

For C:

```
#include <hello.h>
Hello obj;
Environment *ev = somGetGlobalEnvironment();

obj = HelloNew();
__set_msg(obj, ev, "Good Morning");/*note:two leading underscores */
printf("%s\n", __get_msg(obj, ev));
```

For C++:

```
#include <hello.xh>
#include <stdio.h>
Hello *obj;
Environment *ev = somGetGlobalEnvironment();

obj = new Hello;
obj->_set_msg(ev, "Good Morning");
printf("%s\n", obj->_get_msg(ev));
```

Attributes available with each class, if any, are described in the documentation of the class itself in the *SOMobjects Developer Toolkit Programmers Reference Manual.*

Using 'va_list' methods

SOM supports methods whose final argument is a **va_list**. A va_list is a datatype whose representation depends on the operating system platform. On AIX, OS/2 and Windows, a va_list is simply a pointer to a block of memory that contains a number of arguments, sequentially laid out in memory. In the future, SOMobjects may provide special facilities for creating a va_list, but on the above systems the C/C++ **va_arg** macro can be used for this purpose. This is illustrated below.

As a convenience, methods whose final argument is a va_list, such as:

```
void setMany(in short start, in short numArgs, in va_list ap);
```

can be invoked from C and C++ by specifying a variable number of arguments, as follows:

For C:

```
_setMany(aVector, somGetGlobalEnvironment(), 2, 4, 20, 12, 32, 41);
```

For C++:

```
aVector->setMany(somGetGlobalEnvironment(), 2, 4, 20, 12, 32, 41);
```

C programmers must be aware that the short form of the invocation macro used above to pass a variable number of arguments to a va_list method is only available in the absence of ambiguity. The long-form macro (which is always available) requires an explicit va_list argument, as described next.

Note: As mentioned above, the short form may not be available due to ambiguity in the bindings. If a variable-argument interface is desired in such cases, however, you can inspect the usage bindings to find the short form expansion and make use of this directly.

As an alternative to the preceding example, both C and C++ programmers can use an explicit va_list argument, as suggested by the method definition. That is, to use a va_list argument, a method must be invoked as *<className>_<methodName>*, where *<className>* is the name of a class that supports the method.

For example, assume that class "Vector" supports the "setMany" method above. The following code first constructs a variable of type va_list, uses the **va_arg** macro to store the arguments to "setMany" in the va_list, and then invokes the "setMany" method with this variable:

For C:

```
va_list start_ap, ap;
Vector aVector = VectorNew();
...
start_ap = ap = (va_list) 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);
```

For C++:

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

Using name-lookup method resolution

For C/C++ programmers: Offset resolution is the most efficient way to select the method procedure appropriate to a given method call. Client programs can, however, invoke a method using "name-lookup" resolution instead of offset resolution. The C and C++ bindings for method invocation use offset resolution by default, but methods defined with the **namelookup** SOM IDL modifier result in C bindings in which the short form invocation macro uses name-lookup resolution instead. Also, for both C and C++ bindings, a special *lookup_<methodName>* macro is defined.

Name-lookup resolution is appropriate in the case where a programmer knows at compile time which arguments will be expected by a method (that is, its *signature*), but does not know the *type* of the object on which the method will be invoked. For example, name-lookup resolution can be used when two different classes introduce different methods of the same name and signature, and it is not known which method should be invoked (because the type of the object is not known at compile time).

Name-lookup resolution is also used to invoke *dynamic methods* (that is, methods that have been added to a class's interface at run time rather than being specified in the class's IDL specification). For more information on name-lookup method resolution, see the topic "Method Resolution" in Chapter 5, "Implementing Classes in SOM."

For C: To invoke a method using name-lookup resolution, <u>when using the C bindings for a method that has been implemented with the **namelookup** modifier</u>, use either of the following macros:

   _<*methodName*> (*receiver, args*)

   lookup_<*methodName*> (*receiver, args*)

Thus, the short-form method invocation macro results in name-lookup resolution (rather than offset resolution), when the method has been defined as a **namelookup** method. (The long form of the macro for offset resolution is still available in the C usage bindings.) If the method takes a variable number of arguments, then the first form shown above is used when supplying a variable number of arguments, and the second form is used when supplying a **va_list** argument in place of the variable number of arguments.

For C++: To invoke a method using name-lookup resolution, <u>when using the C++ bindings for a method that has been defined with the **namelookup** modifier</u>, use either of the following macros:

   lookup_<*methodName*> (*receiver, args*)

   <className>**_lookup_**<*methodName*> (*receiver, args*)

If the method takes a variable number of arguments, then the first form shown above is used when supplying a variable number of arguments, and the second form is used when supplying a **va_list** argument in place of the variable number of arguments. Note that the offset-resolution forms for invoking methods using the C++ bindings are also still available, even if the method has been defined as a **namelookup** method.

For C/C++: To invoke a method using name-lookup resolution, <u>when the method has *not* been defined as a **namelookup** method</u>:

• Use the **somResolveByName** procedure (described in the following section)**,** or any of the methods **somLookupMethod, somFindMethod** or **somFindMethodOk** to <u>obtain a pointer to the procedure</u> that implements the desired method.

• Then, invoke the desired method by <u>calling that procedure</u>, passing the method's intended receiver, the **Environment** pointer (if needed), the **context** argument (if needed), and the remainder of the method's arguments, if any.

The **somLookupMethod, somFindMethod** and **somFindMethodOK** methods are invoked on a class object (the class of the method receiver should be used), and take as an argument the **somId** for the desired method (which can be obtained from the method's name using the **somIdFromString** function). For more information on these methods, see the *SOMobjects Developer Toolkit Programmers Reference Manual*.

Important Note: SOM provides many ways for a SOM user to acquire a pointer to a method procedure. Once this is done, it becomes the user's responsibility to make appropriate use of this procedure.

• First, the procedure should only be used on objects for which this is appropriate. Otherwise, run-time errors are likely to result.

• Second, when the procedure is used, it is essential that the compiler be given correct information concerning the signature of the method and the linkage required by the method. (On many systems, there are different ways to pass method arguments, and linkage information tells a compiler how to pass the arguments indicated by a method's signature).

SOM method procedures on OS/2 must be called with "system" linkage. On Windows, method procedures use the C linkage convention. On AIX, there is only one linkage convention for procedure calls. While C and C++ provide standard ways to indicate a method signature, the way to indicate linkage information depends on the specific compiler and system. For each method declared using OIDL or IDL, the C and C++ usage bindings therefore use conditional macros and a typedef to name a type that has the correct linkage convention. This type name can then be used by programmers with access to the usage bindings for the class that introduces the method whose procedure pointer is used. The type is named **somTD_**<*className*>_<*methodName*>. This is illustrated in the following example, and further details are provided in the section below, entitled "Obtaining a method's procedure pointer."

A name-lookup example

The following example shows the use of name-lookup by a SOM client programmer. Name-lookup resolution is appropriate when a programmer knows that an object will respond to a method of some given name, but does not know enough about the type of the object to use offset method resolution. How can this happen? It normally happens when a programmer wants to write generic code, using methods of the same name and signature that are applicable to different classes of objects, and yet these classes have no common ancestor that introduces the method. This can easily occur in single-inheritance systems (such as Smalltalk and SOM release 1) and can also happen in multiple-inheritance systems such as SOM release 2 — when class hierarchies designed by different people are brought together for clients' use.

If multiple inheritance is available, it is always possible to create a common class ancestor into which methods of this kind can be migrated. A refactoring of this kind often implements a semantically pleasing generalization that unifies common features of two previously unrelated class hierarchies. This step is most practical, however, when it does not require the redefinition or recompilation of current applications that use offset resolution. SOM is unique in that it allows this.

However, such refactoring must redefine the classes that originally introduced the common methods (so the methods can be inherited from the new "unifying" class instead). A client programmer who simply wants to create an application may not control the implementations of the classes. Thus, the use of name-lookup method resolution seems the best alternative for programmers who do not want to define new classes, but simply to make use of available ones.

For example, assume the existence of two different SOM classes, "classX" and "classY", whose only common ancestor is **SOMObject**, and who both introduce a method named "reduce" that accepts a *string* as an argument and returns a *long*. We assume that the classes were not designed in conjunction with each other. As a result, it is unlikely that the "reduce" method was defined with a **namelookup** modifier. The following figure illustrates the class hierarchy for this example.

Denotes "is a subclass of"
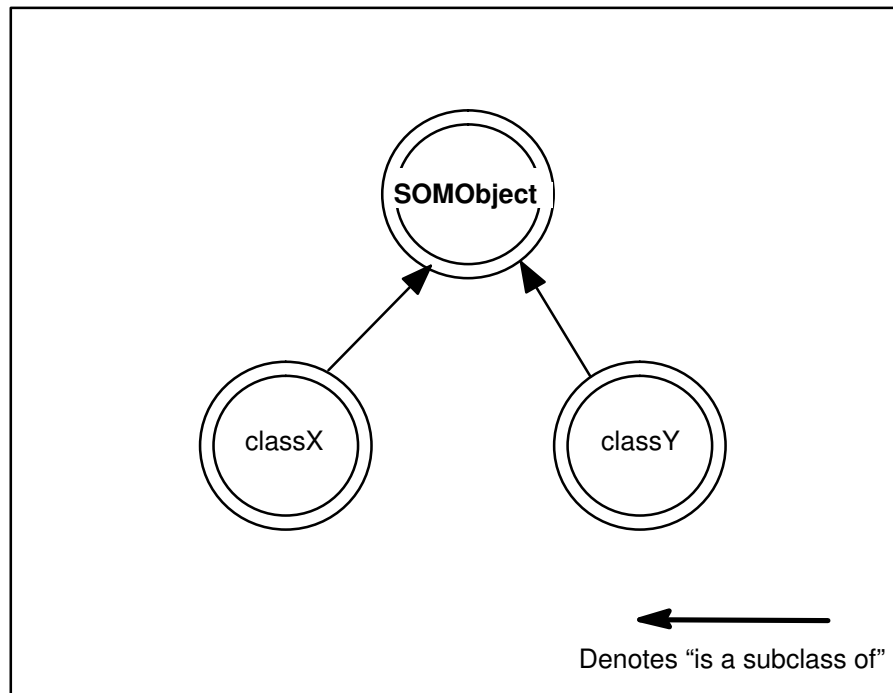
Following is a C++ generic procedure that uses name-lookup method resolution to invoke the "reduce" method on its argument, which may be either of type "classX" or "classY". Note that there is no reason to include classY's usage bindings, since the typedef provided for the "reduce" method procedure in "classX" is sufficient for invoking the method procedure, independently of whether the target object is of type "classX" or "classY".

```
#include <classX.xh> // use classX's method proc typedef

// this procedure can be invoked on a target of type
// classX or classY.

long generic_reduce1(SOMObject *target, string arg)
{
    somTD_classX_reduce reduceProc = (somTD_classX_reduce)
        somResolveByName(target, "reduce");
    return reduceProc(target, arg);
}
```

On the other hand, if the classes were designed in conjunction with each other, and the class designer felt that programmers might want to write generic code appropriate to either class of object, the **namelookup** modifier might have been used. This is a possibility, even with multiple inheritance. However, it is much more likely that the class designer would use multiple inheritance to introduce the "reduce" method in a separate class, and then use this other class as a parent for both "classX" and "classY" (thereby allowing the use of offset resolution).

In any case, if the "reduce" method in "classX" were defined as a **namelookup** method, the following code would be appropriate. Note that the name-lookup support provided by "classX" usage bindings is still appropriate for use on targets that do not have type "classX". As a result, the "reduce" method introduced by "classY" need not have been defined as a **namelookup** method.

```
#include <classX.xh> // use classX's name-lookup support

// this procedure can be invoked on a target of type
// classX or classY.

long generic_reduce2(SOMObject *target, string arg)
{
    return lookup_reduce(target, arg);
}
```

For non-C/C++ programmers: Name-lookup resolution is useful for non-C/C++ programmers when the type of an object on which a method must be invoked is not known at compile time or when method tokens cannot be directly accessed by the programmer. To invoke a method using name-lookup resolution when not using the C or C++ usage bindings, use the **somResolveByName** procedure to acquire a procedure pointer. How the programmer indicates the method arguments and the linkage convention in this case is compiler specific.

The **somResolveByName** procedure takes as arguments a pointer to the object on which the method is to be invoked and the name of the method, as a string. It returns a pointer to the method's procedure (or NULL if the method is not supported by the object). The method can then be invoked by calling the method procedure, passing the method's receiver, the **Environment** pointer (if necessary), the **context** argument (if necessary), and the rest of the method's arguments, if any. (See the section above for C programmers; the arguments to a method procedure are the same as the arguments passed to the long-form C-language method-invocation macro for that method.)

As an example of invoking methods using name-lookup resolution using the procedure **somResolveByName**, the following steps are used to create an instance of a SOM Class *X* in Smalltalk:

1. Initialize the SOM run-time environment (if it is not already initialized) using the **somEnvironmentNew** function.

2. If the class object for class *X* has not yet been created, use **somResolveByName** with the arguments **SOMClassMgrObject** (returned by **somEnvironmentNew** in step 1) and the string *"somFindClass",* to obtain a method procedure pointer for the **somFindClass** method. Use the method procedure for **somFindClass** to create the class object for class *X*: Call the method procedure with these four arguments: **SOMClassMgrObject**; the variable holding class *X*'s **somId** (the result of calling the **somIdFromString** function with argument *"X"*); and the major and minor version numbers for class *X* (or zero). The result is the class object for class *X*.

3. Use **somResolveByName** with arguments the class object for *X* (returned by **somFindClass** in step 2) and the string *"somNew"*, to obtain a method procedure pointer for method **somNew.** (This **somNew** method is used to create instances of a class.)

4. Call the method procedure for **somNew** (using the method procedure pointer obtained in step 3) with the class object for *X* (returned by **somFindClass** in step 3) as the argument. The result is a new instance of class *X.* How the programmer indicates the method arguments and the linkage convention is compiler-specific.

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 run time. The method is then invoked subsequently by calling that procedure, passing the method's intended receiver, the **Environment** pointer (if needed), the **context** argument (if needed), 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 is achieved 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. (See the topic "Method Resolution" in Chapter 5, "Implementing Classes in SOM," for more information on offset and name-lookup method resolution.)

Offset resolution

To obtain a pointer to a procedure using **offset** resolution, 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 in the two macros is that the **SOM_Resolve** macro performs consistency checking on its arguments, but the macro **SOM_ResolveNoCheck**, which is faster, does not. Both macros require the same arguments:

SOM_Resolve(<*receiver*>, <*className*>, <*methodName*>)
SOM_ResolveNoCheck(<*receiver*>, <*className*>, <*methodName*>)

where the arguments are as follows:

receiver	— The object to which the method will apply. It should be specified as an expression without side effects.
className	— The name of the class that introduces the method.
methodName	— The name of the desired method.

These two names (*className* and *methodName*) must be given as tokens, rather than strings or expressions. (For example, as Animal rather than "Animal".) If the symbol **SOM_TestOn** is defined and the symbol **SOM_NoTest** is not defined in the current compilation unit, then **SOM_Resolve** verifies that *receiver* is an instance of *className* or some class derived from *className*. If this test fails, an error message is output and execution is terminated.

The **SOM_Resolve** and **SOM_ResolveNoCheck** macros use the procedure **somResolve** to obtain the entry-point address of the desired method procedure (or raise a fatal error if *methodName* is not introduced by *className)*. This result can be directly applied to the method arguments, or stored in a variable of generic procedure type (for example, **somMethodPtr**) and retained for later method use. This second possibility would result in a loss of information, however, for the reasons now given.

The **SOM_Resolve** or **SOM_ResolveNoCheck** macros are especially useful because they cast the method procedure they obtain to the right type to allow the C or C++ compiler to call this procedure with *system linkage* and with the appropriate arguments. This is why the result of **SOM_Resolve** is immediately useful for calling the method procedure, and why storing the result of **SOM_Resolve** in a variable of some "generic" procedure type results in a loss of information. The correct type information can be regained, however, because the type used by **SOM_Resolve** for casting the result of **somResolve** is available from C/C++ usage bindings using the typedef name **somTD_**<*className*>_<*methodName*>. This type name describes a pointer to a method procedure for *methodName* introduced by class *className*. If the final argument of the method is a **va_list**, then the method procedure returned by **SOM_Resolve** or **SOM_ResolveNoCheck** must be called with a **va_list** argument, and not a variable number of arguments.

Below is a C example of using **SOM_Resolve** to obtain a method procedure pointer for method "sayHello", introduced by class "Hello", and using it to invoke the method on "obj." (Assume that the only argument required by the "sayHello" method is the **Environment** pointer.)

```
somMethodProc *p;
SOMObject obj = HelloNew();
p = SOM_Resolve(obj, Hello, sayHello);
((somTD_Hello_sayHello)p) (obj, somGetGlobalEnvironment());
```

SOM_Resolve and **SOM_ResolveNoCheck** can only be used to obtain method procedures for *static methods* (methods that have been declared in an IDL specification for a class) and not methods that are added to a class at run time. See the *SOMobjects Programmers Reference Manual* for more information and examples on **SOM_Resolve** and **SOM_ResolveNoCheck**.

Name-lookup method resolution

To obtain a pointer to a method's procedure using **name-lookup** resolution, use the **somResolveByName** procedure (described in the following section)**,** or any of the **somLookupMethod, somFindMethod** and **somFindMethodOK** methods. These methods are invoked on a class object that supports the desired method, and they take an argument specifying the a **somId** for the desired method (which can be obtained from the method's name using the **somIdFromString** function). For more information on these methods and for examples of their use, see the *SOMobjects Developer Toolkit Programmers Reference Manual*.

Method name or signature not known at compile time

If the programmer does not know a method's name at compile time (for example, it might be specified by user input), then the method can be invoked in one of two ways, depending upon whether its signature is known:

- Suppose the signature of the method is known at compile time (even though the method name is not). In that case, when the name of the method becomes available at run time, the **somLookupMethod, somFindMethod** or **somFindMethodOk** methods or the **somResolveByName** procedure can be used to obtain a pointer to the method's procedure using name-lookup method resolution, as described in the preceding topics. That method procedure can then be invoked, passing the method's intended receiver, the **Environment** pointer (if needed), the **context** argument (if needed), and the remainder of the method's arguments.

- If the method's signature is unknown until run time, then dispatch-function resolution is indicated, as described in the next topic.

Dispatch-function method resolution

If the *signature* of the method is *not* known at compile time (and hence the method's argument list cannot be constructed until run time), then the method can be invoked at run time by (a) placing the arguments in a variable of type **va_list** at run time and (b) either using the **somGetMethodData** method followed by use of the **somApply** function, or by invoking the **somDispatch** or **somClassDispatch** method. Using **somApply** is more efficient, since this is what the **somDispatch** method does, but it requires two steps instead of one. In either case, the result invokes a "stub" procedure called an *apply stub*, whose purpose is to remove the method arguments from the **va_list**, and then pass them to the appropriate method procedure in the way expected by that procedure. For more information on these methods and for examples of their use, see the **somApply** function, and the **somGetMethodData, somDispatch,** and **somClassDispatch** methods in the *SOMobjects Programmers Reference Manual*.

Using class objects

Using a class object encompasses three aspects: getting the class of an object, creating a new class object, or simply referring to a class object through the use of a pointer.

Getting the class of an object

To get the class that an object is an instance of, 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. For example, the following statements store in "myClass" the class object of which "obj" is an instance.

```
myClass = _somGetClass(obj);    (for C)

myClass = obj->somGetClass();    (for C++)
```

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. The section below entitled "Getting information about a class" describes the methods that can be invoked on a class object after it is obtained using **somGetClass**.

The **somGetClass** method can be overridden by a class 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 in order 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 (if **som-GetClass** has been overridden). This difference may be limited to side effects, but it is possible for their results to differ as well. The **SOM_GetClass** macro should only be used when absolutely necessary.

Creating a class object

A class object is created automatically the first time the <*className*>**New** macro (for C) or the **new** operator (C++) is invoked to create an instance of that class. In other situations, however, it may be necessary to create a class object explicitly, as this section describes.

Using <*className*>Renew or somRenew

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. Getting this information requires creating the class object (see the example under "Creating instances of a class" early in this chapter). 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**, and these methods require that the class object be created in advance.

Use the <*className*>**NewClass** procedure to create a class object :

- When using the C/C++ language bindings for the class, and

- When the name of the class is known at compile time.

Using <*className*>NewClass

The <*className*>**NewClass** procedure initializes the SOM run-time environment, if necessary, creates the class object (unless it already exists), 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 its creation, the class object can be referenced in client code using the macro

_<*className*>   (for C and C++ programs)

or the expression

<*className*>**ClassData.classObject**   (for C and C++ programs).

The <*className*>**NewClass** procedure takes two arguments, the major version number and 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 a higher minor version number. If the class is not compatible, an error is raised. Major version numbers usually only change 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. Zero can be used in place of version numbers to bypass version number checking.

When using SOM bindings for a class, these bindings define constants representing the major and minor version numbers of the class at the time the bindings were generated. These constants are named *<className>*_MajorVersion and *<className>*_MinorVersion. For example, the following procedure call:

```
AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
```

creates the class object for class "Animal". Thereafter, `_Animal` can be used to reference the "Animal" class object.

The preceding technique for checking version numbers is not failsafe. For performance reasons, the version numbers for a class are only checked when the class object is created, and not when the class object or its instances are used. Thus, run-time errors may result when usage bindings for a particular version of a class are used to invoke methods on objects created by an earlier version of the class.

Using somFindClass or somFindClsInFile

To create a class object <u>when *not* using the C/C++ language bindings for the class</u>, or <u>when the class name is *not* known at compile time</u>:

- First, initialize the SOM run-time environment by calling the **somEnvironmentNew** function (unless it is known that the SOM run-time environment has already been initialized).

- Then, use the **somFindClass** or **somFindClsInFile** method to create the class object. (The class must already be defined in a dynamically linked library, or DLL.)

The **somEnvironmentNew** function initializes the SOM run-time environment. That is, it creates the four primitive SOM objects (**SOMClass, SOMObject, SOMClassMgr,** and the **SOMClassMgrObject**), and it initializes SOM global variables. The function takes no arguments and returns a pointer to the **SOMClassMgrObject**.

Note: Although **somEnvironmentNew** must be called before using other SOM functions and methods, explicitly calling **somEnvironmentNew** is usually not necessary when using the C/C++ bindings, because the macros for *<className>***NewClass**, *<className>***New**, and *<className>***Renew** call it automatically, as does the **new** operator for C++. Calling **somEnvironmentNew** repeatedly does no harm.

After the SOM run-time environment has been initialized, the methods **somFindClass** and **somFindClsInFile** can be used to create a class object. These methods must be <u>invoked on the class manager</u>, which is pointed to by the global variable **SOMClassMgrObject.** (It is also returned as the result of **somEnvironmentNew**.)

The **somFindClass** method takes the following arguments:

classId	— A **somId** identifying the name of the class to be created. The **somIdFromString** function returns a *classId* given the name of the class.
major version number	— The expected major version number of the class.
minor version number	— The expected minor version number of the class.

The version 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 **somFindClass** method dynamically loads the DLL containing the class's implementation, if needed, creates the class object (unless it already exists) by invoking its *<className>***NewClass** procedure, and returns a pointer to it. If the class could not be created, **somFindClass** returns NULL. For example, the following C code fragment creates the class "Hello" and stores a pointer to it in "myClass":

```
SOMClassMgr cm = somEnvironmentNew();
somId classId = somIdFromString("Hello");
SOMClass myClass = _somFindClass(SOMClassMgrObject, classId,
                    Hello_MajorVersion, Hello_MinorVersion);
. . .
SOMFree(classId);
```

The **somFindClass** method uses **somLocateClassFile** to get the name of the library file containing the class. If the class was defined with a "dllname" class modifier, then **somLocate-ClassFile** returns that file name; otherwise, it assumes that the class name is the name of the library file. The **somFindClsInFile** method is similar to **somFindClass,** except that it takes an additional (final) argument — the name of the library file containing the class. The **somFindCl-sInFile** method is useful when a class is packaged in a DLL along with other classes and the "dllname" class modifier has not been given in the class's IDL specification.

Warning: On AIX and Windows, the **somFindClass** and **somFindClsInFile** methods should *not* be used to create a class whose implementation is statically linked with the client program. Instead, the class object should be created using the *<className>***NewClass** procedure provided by the class's .h/.xh header file. Static linkage is not created by simply including usage bindings in a program, but by use of the offset-resolution method-invocation macros.

Referring to class objects

Saving a pointer as the class object is created: The *<className>***NewClass** macro and the **somFindClass** method, used to create class objects, both return a pointer to the newly created class object. Hence, one way to obtain a pointer to a class object is to save the value returned by *<className>***NewClass** or **somFindClass** when the class object is created.

Getting a pointer after the class object is created: After a class object has been created, client programs can also get a pointer to the class object from the class name. When the class name is known at compile time and the client program is using the C or C++ language bindings, the macro

    _*<className>*

can be used to refer to the class object for *<className>*. Also, when the class name is known at compile time and the client program is using the C or C++ language bindings, the expression

    *<className>***ClassData.classObject**

refers to the class object for *<className>*. For example, `_Hello` refers to the class object for class "Hello" in C or C++ programs, and `HelloClassData.classObject` refers to the class object for class "Hello." in C or C++ programs.

Getting a pointer to the class object from an instance: If any instances of the class are known to exist, a pointer to the class object can also be obtained by invoking the **somGetClass** method on such an instance. (See "Getting the class of an object," above.)

Getting a pointer in other situations: If the class name is *not* known until run time, or if the client program is *not* using the C or C++ language bindings, and *no* instances of the class are known to exist, then the **somClassFromId** method can be used to obtain a pointer to a class object after the class object has been created. The **somClassFromId** method should be invoked on the class manager, which is pointed to by the global variable **SOMClassMgrObject.** The only argument to the method is a **somId** for the class name, which can be obtained using the **somIdFromString** function. The method **somClassFromId** returns a pointer to the class object of the specified class. For example, the following C code stores in "myClass" a pointer to the class object for class "Hello" (or NULL, if the class cannot be located):

```
SOMClassMgr cm    =   somEnvironmentNew();
somId classId     =   somIdFromString("Hello");
SOMClass myClass = _somClassFromId(SOMClassMgrObject, classId,
                       Hello_MajorVersion, Hello_MinorVersion);
SOMFree(classId);
```

Compiling and linking

This section describes how to compile and link C and C++ client programs. Compiling and linking a client program with a SOM class is done in one of two ways, depending upon whether or not the class is packaged as a library, as described below.

If the class is _not_ packaged as a library (that is, the client program has the implementation source code for the class, as in the examples given in the SOM IDL tutorial), then the client program can be compiled together with the class implementation file as follows. (This assumes that the client program and the class are both implemented in the same language, C or C++. If this is not the case, then each module must be compiled separately to produce an object file and the resulting object files linked together to form an executable.)

In the following examples, the environment variable SOMBASE refers to the directory in which SOM has been installed. The examples also assume that the header files and the import library for the "Hello" class reside in the "include" and "lib" directories where SOM has been installed. If this is not the case, additional path information should be supplied for these files. For client program "main" and class "Hello":

Under AIX, for **C** programmers:
```
> xlc -I. -I$SOMBASE/include main.c hello.c \
      -L$SOMBASE/lib -lsomtk -o main
```

Under AIX, for **C++** programmers:
```
> xlC -I. -I$SOMBASE/include main.C hello.C \
      -L$SOMBASE/lib -lsomtk -o main
```

Under OS/2, for **C** programmers:
```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include main.c hello.c somtk.lib
```

Under OS/2, for **C++** programmers:
```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include main.cpp hello.cpp somtk.lib
```

Under Windows, for **C** programmers:
```
> cl -AL -Zp1 -I. -I%SOMBASE%\include \
     main.c hello.c llibcew.lib libw.lib somtk.lib main.def
```

Under Windows, for **C++** programmers:
```
> cl -AL -Zp1 -I. -I%SOMBASE%\include \
     main.cpp hello.cpp llibcew.lib libw.lib somtk.lib main.def
```

Windows compiler notes: The SOM Compiler and runtime libraries under Windows only provide support for Large Memory Model systems. The "–Oi" compile flag should be used whenever a SOM Windows program contains or calls any functions that return floats or doubles. The "–F" option can be used to set the stack size if STACKSIZE is not specified in a .def file. Additionally, for this manual, compile and link commands are demonstrated using the Visual C++ compiler. Sample Makefiles shipped with SOMobjects For Windows also show how to build with other compilers.

If the class is packaged as a class library, then the client program, "main", is compiled as above, except that the class implementation file is not part of the compilation. Instead, the "import library" provided with the class library is used to resolve the symbolic references that appear in "main". For example, to compile the C client program "main.c" that uses class "Hello":

Under AIX:
```
> xlc -I. -I$SOMBASE/include main.c -lc -L$SOMBASE/lib -lsomtk \
      -lhello -o main
```

Under OS/2:
```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include main.c somtk.lib hello.lib
```

Under Windows:
```
> cl -AL -Zp1 -I. -I%SOMBASE%\include \
     main.c llibcew.lib libw.lib somtk.lib hello.lib main.def
```

Windows users: See also "Windows Compiler Notes" above.

3.3 Language-neutral Methods and Functions

This section describes methods, functions, and macros that client programs can use regardless of the programming language in which they are written. In other words, these functions and methods are *not* part of the C or C++ bindings.

Generating output

The following functions and methods are used to generate output, including descriptions of SOM objects. They all produce their output using the character-output procedure held by the global variable **SOMOutCharRoutine**. The default procedure for character output simply writes the character to *stdout*, but it can be replaced to change the output destination of the methods and functions below. (See Chapter 5 for more information on customizing SOM.)

somDumpSelf	— (method) Writes a detailed description of an object, including its class, its location, and its instance data. The receiver of the method is the object to be dumped. An additional argument is the "nesting level" for the description. [All lines in the description will be indented by (2 * level) spaces.]
somPrintSelf	— (method) Writes a brief description of an object, including its class and location in memory. The receiver of the method is the object to be printed.
somPrintf	— (function) SOM's version of the C "printf" function. It generates character stream output via **SOMOutCharRoutine**. It has the same interface as the C "printf" function.
somVprintf	— (function) Represents the "vprint" form of **somPrintf**. Its arguments are a formatting string and a **va_list** holding the remaining arguments.
somPrefixLevel	— (function) Generates (via **somPrintf**) spaces to prefix a line at the indicated level. The return type is **void**. The argument is an integer specifying the level. The number of spaces generated is (2 * level).
somLPrintf	— (function) Combines **somPrefixLevel** and **somPrintf.** The first argument is the level of the description (as for **somPrefixLevel**) and the remaining arguments are as for **somPrintf** (or for the C "printf" function).

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific function or method.

Getting information about a class

The following methods are used to obtain information about a class or to locate a particular class object:

somCheckVersion	— Checks a class for compatibility with the specified major and minor version numbers. The receiver of the method is the SOM class about which information is needed. Additional arguments are values of the major and minor version numbers. The method returns TRUE if the class is compatible, or FALSE otherwise.
somClassFromId	— Finds the class object of an existing class when given its **somId**, but without loading the class. The receiver of the method is the class manager (pointed to by the global variable **SOMClassMgrObject**). The additional argument is the class's **somId**. The method returns a pointer to the class (or NULL if the class does not exist).

somDescendedFrom	— Tests whether one class is derived from another. The receiver of the method is the class to be tested, and the potential ancestor class is the argument. The method returns TRUE if the relationships exists, or FALSE otherwise.
somFindClass	— Finds or creates the class object for a class, given the class's **somId** and its major and minor version numbers. The receiver of the method is the class manager (pointed to by the global variable **SOMClassMgrObject**). Additional arguments are the class's **somId** and the major and minor version numbers. The method returns a pointer to the class object, or NULL if the class could not be created.
somFindClsInFile	— Finds or creates the class object for a class. This method is similar to **somFindClass**, except the user also provides the name of a file to be used for dynamic loading, if needed. The receiver of the method is the class manager (pointed to by the global variable **SOMClassMgrObject**). Additional arguments are the class's **somId**, the major and minor version numbers, and the file name. The method returns a pointer to the class object, or NULL if the class could not be created.
somGetInstancePartSize	— Obtains the size of the instance variables introduced by a class. The receiver of the method is the class object. The method returns the amount of space, in bytes, needed for the instance variables.
somGetInstanceSize	— Obtains the total size requirements for an instance of a class. The receiver of the method is the class object. The method returns the amount of space, in bytes, required for the instance variables introduced by the class itself and by all of its ancestor classes.
somGetName	— Obtains the name of a class. The receiver of the method is the class object. The method returns the class name.
somGetNumMethods	— Obtains the number of methods available for a class. The receiver of the method is the class object. The method returns the total number of currently available methods (static or otherwise, including inherited methods).
somGetNumStaticMethods	— Obtains the number of static methods available for a class. (A static method is one declared in the class's interface specification [.idl] file.) The receiver of the method is the class object. The method returns the total number of available static methods, including inherited ones.
somGetParents	— Obtains a sequence of the parent (base) classes of a specified class. The receiver of the method is the class object. The method returns a pointer to a linked list of the parent (base) classes (unless the receiver is **SOMObject**, for which it returns NULL).
somGetVersionNumbers	— Obtains the major and minor version numbers of a class. The receiver of the method is the class object. The return type is void, and the two arguments are pointers to locations in memory where the method can store the major and minor version numbers (of type **long**).

| **somSupportsMethod** | — Indicates whether instances of a given class support a given method. The receiver of the **somSupportsMethod** method is the class object. The argument is the **somId** for the method in question. The **somSupportsMethod** method returns TRUE if the method is supported, or FALSE otherwise. |

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific method.

Getting information about an object

The following methods and functions are used to obtain information about an object (instance) or to determine whether a variable holds a valid SOM object.

Methods

| **somGetClass** | — Gets the class object of a specified object. The receiver of the method is the object whose class is desired. The method returns a pointer to the object's corresponding class object. |

| **somGetClassName** | — Obtains the class name of an object. The receiver of the method is the object whose class name is desired. The method returns a pointer to the name of the class of which the specified object is an instance. |

| **somGetSize** | — Obtains the size of an object. The receiver of the method is the object. The method returns the amount of contiguous space, in bytes, that is needed to hold the object itself (not including any additional space that the object may be using or managing outside of this area). |

| **somIsA** | — Determines whether an object is an instance of a given class or of one of its descendant classes. The receiver of the method is the object to be tested. An additional argument is the name of the class to which the object will be compared. This method returns TRUE if the object is an instance of the specified class or if (unlike **somIsInstanceOf**) it is an instance of any descendant class of the given class; otherwise, the method returns FALSE. |

| **somIsInstanceOf** | — Determines whether an object is an instance of a specific class (but not of any descendant class). The receiver of the method is the object. The argument is the name of the class to which the object will be compared. The method returns TRUE if the object is an instance of the specified class, or FALSE otherwise. |

| **somRespondsTo** | — Determines whether an object supports a given method. The receiver of the method is the object. The argument is the **somId** for the method in question. (A **somId** can be obtained from a string by using the **somIdFromString** function.) The **somRespondsTo** method returns TRUE if the object supports the method, or FALSE otherwise. |

Functions

| **somIsObj** | — Takes as its only argument an address (which may not be valid). The function returns TRUE (1) if the address contains a valid SOM object, or FALSE (0) otherwise. This function is designed to be failsafe. |

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific method or function.

Debugging

The following macros are used to conditionally generate output for debugging. All output generated by these macros is written using the replaceable character-output procedure pointed to by the global variable **SOMOutCharRoutine**. The default procedure simply writes the character to *stdout*, but it can be replaced to change the output destination of the methods and functions below. (See Chapter 5, "Implementing Classes in SOM," for more information on customizing SOM.)

Debugging output is produced or suppressed based on the settings of three global variables, **SOM_TraceLevel, SOM_WarnLevel,** and **SOM_AssertLevel:**

- **SOM_TraceLevel** controls the behavior of the *<className>***MethodDebug** macro;
- **SOM_WarnLevel** controls the behavior of the macros **SOM_WarnMsg**, **SOM_TestC,** and **SOM_Expect;** and
- **SOM_AssertLevel** controls the behavior of the **SOM_Assert** macro.

Available macros for generating debugging output are as follows:

*<className>***MethodDebug**

— (macro for C and C++ programmers using the SOM language bindings for *<className>*)
The arguments to this macro are a class name and a method name. If the **SOM_TraceLevel** global variable has a nonzero value, the <className>**MethodDebug** macro produces a message each time the specified method (as defined by the specified class) is executed. This macro is typically used within the procedure that implements the specified method. (The SOM Compiler automatically generates calls to the *<className>***MethodDebug** macro within the implementation template files it produces.) To suppress method tracing for all methods of a class, put the following statement in the implementation file after including the header file for the class:

```
#define <className>MethodDebug(c,m) \
                    SOM_NoTrace(c,m)
```

This can yield a slight performance improvement. The **SOMMTraced** metaclass, discussed below, provides a more extensive tracing facility that includes method parameters and returned values.

SOM_TestC
— The **SOM_TestC** macro takes as an argument a boolean expression. If the boolean expression is TRUE (nonzero) and **SOM_AssertLevel** is greater than zero, then an informational message is output. If the expression is FALSE (zero) and **SOM_WarnLevel** is greater than zero, a warning message is produced.

SOM_WarnMsg
— The **SOM_WarnMsg** macro takes as an argument a character string. If the value of **SOM_WarnLevel** is greater than zero, the specified message is output.

SOM_Assert
— The **SOM_Assert** macro takes as arguments a boolean expression and an error code (an integer). If the boolean expression is TRUE (nonzero) and **SOM_AssertLevel** is greater than zero, then an informational message is output. If the expression is FALSE (zero), and the error code indicates a warning-level error and **SOM_WarnLevel** is greater than zero, then a warning message is output. If the expression is FALSE and the error code indicates a fatal error, then an error message is produced and the process is terminated.

SOM_Expect	— The **SOM_Expect** macro takes as an argument a boolean expression. If the boolean expression is FALSE (zero) and **SOM_WarnLevel** is set to be greater than zero, then a warning message is output. If *condition* is TRUE and **SOM_AssertLevel** is set to be greater than zero, then an informational message is output.

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific macro.

The **somDumpSelf** and **somPrintSelf** methods can be useful in testing and debugging. The **somPrintSelf** method produces a brief description of an object, and the **somDumpSelf** method produces a more detailed description. See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information.

Checking the validity of method calls

The C and C++ language bindings include code to check the validity of method calls at run time. If a validity check fails, the **SOM_Error** macro ends the process. (**SOM_Error** is described below.) To enable method-call validity checking, place the following directive in the client program prior to any *#include* directives for SOM header files:

```
#define SOM_TestOn
```

Alternatively, the **–DSOM_TestOn** option can be used when compiling the client program to enable method-call validity checking.

Exceptions and error handling

In the classes provided in the SOM run-time library (that is, **SOMClass, SOMObject,** and **SOMClassMgr**), error handling is performed by a user-replaceable procedure, pointed to by the global variable **SOMError**, that produces an error message and an error code and, if appropriate, ends the process where the error occurred. (Chapter 5 describes how to customize the error handling procedure.)

Each error is assigned a unique integer error code. Errors are grouped into three categories, based on the last digit of the error code:

SOM_Ignore	— This category of error represents an informational event. The event is considered normal and can be ignored or logged at the user's discretion. Error codes having a last digit of 2 belong to this category.
SOM_Warn	— This category of error represents an unusual condition that is not a normal event, but is not severe enough to require program termination. Error codes having a last digit of 1 belong to this category.
SOM_Fatal	— This category of error represents a condition that should not occur or that would result in loss of system integrity if processing were allowed to continue. In the default error handling procedure, these errors cause the termination of the process in which they occur. Error codes having a last digit of 9 belong to this category.

The various codes for all errors detected by SOM are listed in Appendix A, "Customer Support and Error Codes."

When errors are encountered in client programs or user defined-classes, the following two macros can be used to invoke the error-handling procedure:

SOM_Error — The **SOM_Error** macro takes an error code as its only argument and invokes the SOM error handling procedure (pointed to by the global variable **SOMError**) to handle the error. The default error handling procedure prints a message that includes the error code, the name of the source file, and the line number where the macro was invoked. If the last digit of the error code indicates a serious error (of category **SOM_Fatal**), the process causing the error is terminated. (Chapter 5 describes how to customize the error handling procedure.)

SOM_Test — The **SOM_Test** macro takes a boolean expression as an argument. If the expression is TRUE (nonzero) and the **SOM_AssertLevel** is greater than zero, then an informational message is output. If the expression is FALSE (zero), an error message is produced and the program is terminated.

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific macro.

Other classes provided by the SOMobjects Toolkit (including those in the Persistence, Replication, DSOM, and Interface Repository frameworks, and the utility classes and metaclasses) handle errors differently. Rather than invoking **SOMError** with an error code, their methods return *exceptions* via the (**Environment ***) inout parameter required by these methods. The following sections describe the exception declarations, the standard exceptions, and how to set and get exception information in an **Environment** structure.

Exception declarations

As discussed in Chapter 4 in Section 4.2 entitled "SOM Interface Definition Language," a method may be declared to return zero or more **exceptions**. IDL exceptions are implemented by simply passing back error information after a method call, as opposed to the "catch/throw" model where an exception is implemented by a long jump or signal. Associated with each type of exception is a name and, optionally, a struct-like data structure for holding error information. A method declares the types of exceptions it may return in a **raises** expression.

Below is an example IDL declaration of a "BAD_FLAG" exception, which may be "raised" by a "checkFlag" method, as part of a "MyObject" interface:

```
interface MyObject {
    exception BAD_FLAG { long ErrCode; char Reason[80]; };

    void checkFlag(in unsigned long flag) raises(BAD_FLAG);
};
```

An exception structure contains whatever information is necessary to help the caller understand the nature of the error. The exception declaration can be treated like a **struct** definition: that is, whatever you can access in an IDL **struct**, you can access in an **exception** declaration. Alternatively, the structure can be *empty*, whereby the exception is just identified by its name.

The SOM Compiler will map the exception declaration in the above example to the following C language constructs:

```
typedef struct BAD_FLAG {
    long ErrCode;
    char Reason[80];
} BAD_FLAG;

#define ex_BAD_FLAG "MyObject::BAD_FLAG"
```

When an exception is detected, the "checkFlag" method must call **SOMMalloc** to allocate a "BAD_FLAG" structure, initialize it with the appropriate error information, and make a call to **somSetException** (see "Setting an exception value," below) to record the exception value in the **Environment** structure passed in the method call. The caller, after invoking "checkFlag", can check the **Environment** structure that was passed to the method to see if there was an exception, and if so, extract the exception value from the **Environment** (see "Getting an exception value," below.)

Standard exceptions

In addition to user-defined exceptions (those defined explicitly in an IDL file), there are several predefined exceptions for system run-time errors. A *system exception* can be returned on any method call. (That is, they are implicitly declared for every method whose class uses IDL call style, and they do not appear in any **raises** expressions.) The standard exceptions are listed in Table 2 of Section 4.5, "SOM Interface Definition Language". Most of the predefined system exceptions pertain to Object Request Broker errors. Consequently, these types of exceptions are most likely to occur in DSOM applications (Chapter 6).

Each of the standard exceptions has the same structure: an error code (to designate the subcategory of the exception) and a completion status code. For example, the NO_MEMORY standard exception has the following definition:

```
enum completion_status {YES, NO, MAYBE};
exception NO_MEMORY { unsigned long minor;
                      completion_status completed; };
```

The completion status value indicates whether the method was never initiated (NO), completed execution prior to the exception (YES), or the completion status is indeterminate (MAYBE).

Since all the standard exceptions have the same structure, file "somcorba.h" (included by "som.h") defines a generic **StExcep** typedef which can be used instead of the specific typedefs:

```
typedef struct StExcep {
     unsigned long minor;
     completion_status completed;
} StExcep;
```

The standard exceptions are defined in an IDL module called **StExcep**, in the file named "stexcep.idl", and the C definitions can be found in "stexcep.h".

The Environment

The **Environment** is a data structure that contains environmental information that can be passed between a caller and a called object when a method is executed. For example, it is used to pass information about the user id of a client, to return exception data to the client following a method call, and so on.

A pointer to an **Environment** variable is passed as an argument to method calls (unless the method's class has the **callstyle=oidl** SOM IDL modifier). The **Environment** typedef is defined in "som.h", and an instance of the structure is allocated by the caller in any reasonable way: on the stack (by declaring a local variable and initializing it using the macro **SOM_InitEnvironment**), dynamically (using the **SOM_CreateLocalEnvironment** macro), or by calling the **somGetGlobalEnvironment** function to allocate an **Environment** structure to be shared by objects running in the same thread.

For class libraries that use **callstyle=oidl**, there is no explicit **Environment** parameter. For these libraries, exception information may be passed using the per-thread **Environment** structure returned by the **somGetGlobalEnvironment** procedure.

Setting an exception value

To set an exception value in the caller's **Environment** structure, a method implementation makes a call to the **somSetException** procedure:

void somSetException (Environment *ev*,
                            exception_type *major*,
                            string *exception_name*,
                            void **params***);**

where "ev" is a pointer to the **Environment** structure passed to the method, "major" is an **exception_type,** "exception_name" is the string name of the exception (usually the constant defined by the IDL compiler, for example, `ex_BAD_FLAG`), and "params" is a pointer to an (initialized) exception structure which <u>must be allocated by **SOMMalloc:**</u>

```
typedef enum exception_type {
   NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION
} exception_type;
```

It is important to reiterate that **somSetException** expects the *params* argument to be a pointer to a structure that was allocated using **SOMMalloc**. When **somSetException** is called, the client *passes ownership* of the exception structure to the SOM run-time environment. The SOM run-time environment will free the structure when the exception is reset (that is, upon the next call to **somSetException**), or when the **somExceptionFree** function is called.

Note that **somSetException** simply sets the exception value; it performs no exit processing. If there are multiple calls to **somSetException** before the method returns, the caller will only see the last exception value.

Getting an exception value

After a method returns, the calling client program can look at the **Environment** structure to see if there was an exception. The **Environment** struct is mostly opaque, except for an exception type field named **_major**:

```
typedef struct Environment {
   exception_type   _major;
   ...
} Environment;
```

If **ev._major != NO_EXCEPTION**, there was an exception returned by the call. The caller can retrieve the exception name and value (passed as parameters in the **somSetException** call) from an **Environment** struct via the following functions:

string somExceptionId (Environment *ev***);**
somToken somExceptionValue (Environment *ev***);**

The **somExceptionId** function returns the exception name, if any, as a string. The function **somExceptionValue** returns a pointer to the value of the exception, if any, contained in the exception structure. If NULL is passed as the **Environment** pointer in either of the above calls, an implicit call is made to **somGetGlobalEnvironment**.

The **somExceptionFree** function will free any memory in the **Environment** associated with the last exception:

void somExceptionFree (Environment *ev***);**

If preferred, developers can alternatively use the CORBA "exception_free" API to free the memory in an **Environment** structure.

Note**:** File "somcorba.h" (included by "som.h") provides the following aliases for strict compliance with CORBA programming interfaces:

```
#ifdef CORBA_FUNCTION_NAMES
#define exception_id     somExceptionId
#define exception_value  somExceptionValue
#define exception_free   somExceptionFree
#endif /* CORBA_FUNCTION_NAMES */
```

The following IDL interface for a "MyObject" object (in a file called "myobject.idl") declares a "BAD_FLAG″ exception, which can be raised by the "checkFlag" method,:

```
interface MyObject {
    exception BAD_FLAG { long ErrCode; char Reason[80]; };

    void checkFlag(in unsigned long flag) raises(BAD_FLAG);
};
```

The SOM IDL compiler will map the exception to the following C language constructs, in "myobject.h":

```
typedef struct BAD_FLAG {
    long ErrCode;
    char Reason[80];
} BAD_FLAG;

#define ex_BAD_FLAG "MyObject::BAD_FLAG"
```

A client program that invokes the "checkFlag" method might contain the following error handling code. (Note: The error checking code below lies in the user-written procedure, "ErrorCheck," so the code need not be replicated through the program.)

```
#include "som.h"
#include "myobject.h"

boolean ErrorCheck(Environment *ev);   /* prototype */

main()
{
    unsigned long flag;
    Environment ev;
    MyObject myobj;
    char      *exId;
    BAD_FLAG *badFlag;
    StExcep  *stExValue;

    myobj = MyObjectNew();
    flag  = 0x01L;
    SOM_InitEnvironment(&ev);

     /* invoke the checkFlag method, passing the Environment param */
    _checkFlag(myobj, &ev, flag);

    /* check for exception */
    if (ErrorCheck(&ev))
    {
       /* ... */
       somExceptionFree(&ev);   /* free the exception memory */
    }

    /* ... */
}
```

```
/* error checking procedure */

boolean ErrorCheck(Environment *ev)
{
   switch (ev._major)
   {
   case SYSTEM_EXCEPTION:
      /* get system exception id and value */
      exId      = somExceptionId(ev);
      stExValue = somExceptionValue(ev);
      /* ... */
      return(TRUE);

   case USER_EXCEPTION:
      /* get user-defined exception id and value */
      exId = somExceptionId(ev);
      if (strcmp(exId, ex_BAD_FLAG) == 0)
      {
         badFlag = (BAD_FLAG *) somExceptionValue(ev);
         /* ... */
      }
      /* ... */
      return(TRUE);

   case NO_EXCEPTION:
      return(FALSE);
   }
}
```

The implementation of the "checkFlag" method might contain the following error-handling code:

```
#include "som.h"
#include "myobject.h"

void checkFlag(MyObject somSelf, Environment *ev,
               unsigned long flag)
{
   BAD_FLAG *badFlag;
   /* ... */

   if ( /* flag is invalid */ )
   {
      badFlag = (BAD_FLAG *) SOMMalloc(sizeof(BAD_FLAG));
      badFlag->ErrCode = /* bad flag code */;
      strcpy(badFlag->Reason, "bad flag was passed");
      somSetException(ev, USER_EXCEPTION,
                      ex_BAD_FLAG, (void *)badFlag);
      return;
   }
   /* ... */
}
```

Memory management

The SOMobjects Toolkit provides several functions that can be used for memory management.

Using SOM equivalents to ANSI C functions

The memory management functions used by SOM are a subset of those supplied in the ANSI C standard library. They have the same calling interface and the same return types as their ANSI C equivalents, but include supplemental error checking. Errors detected by these functions are passed to **SOMError** (described in the previous section). The correspondence between SOM memory management functions and their ANSI C standard library equivalents is shown below:

SOM Function	Equivalent ANSI C Library Routine
SOMMalloc	**malloc**
SOMCalloc	**calloc**
SOMRealloc	**realloc**
SOMFree	**free**

SOMMalloc, SOMCalloc, SOMRealloc, and **SOMFree** are actually *global variables* that point to the SOM memory management functions (rather than being the names of the functions themselves), so that users can replace them with their own memory management functions if desired. (See Chapter 5 for information on replacing the SOM memory management functions.)

Clearing memory for objects

The memory associated with objects initialized by a client program must also be freed by the client. The SOM-provided method **somFree** is used to release the storage containing the receiver object:

```
#include "origcls.h"

main ()
{
   OrigCls myObject;
   myObject = OrigClsNew ();

    /*  Code to use myObject  */

   _somFree (myObject);
}
```

Clearing memory for the Environment

Any memory associated with an exception in an **Environment** structure is typically freed using the **somExceptionFree** function. (Or, the CORBA "exception_free" API can be used.) The **somExceptionFree** function takes the following form (also see "Example" in the previous topic for an application example):

> **void somExceptionFree (Environment ****ev***);**

Note: For information on managing the memory, objects, and exceptions used by DSOM applications, see "Memory management for DSOM" in Chapter 6, "Distributed SOM (DSOM)."

SOM manipulations using somId's

A **somId** is similar to a number that represents a zero-terminated **string**. A **somId** is used in SOM to identify method names, class names, and so forth. For example, many of the SOM methods that take a method or class name as a parameter require a value of type **somId** rather than **string**. All SOM manipulations using **somId**s are case insensitive, although the original case of the **string** is preserved.

During its first use with any of the following functions, a **somId** is automatically converted to an internal representation (registered). Because the representation of a **somId** changes, a special SOM type (**somId**) is provided for this purpose. Names and the corresponding **somId** can be declared at compile time, as follows:

```
string example = "exampleMethodName";
somId exampleId = &example;
```

or a **somId** can be generated at run time, as follows:

```
somId myMethodId;
myMethodId = somIdFromString("exampleMethodName");
```

SOM provides the following functions that generate or use a **somId**:

somIdFromString	— Finds the **somId** that corresponds to a **string**. The method takes a **string** as its argument, and returns a value of type **somId** that represents the string. The returned **somId** must later be freed using **SOMFree.**
somStringFromId	— Obtains the **string** that corresponds to a **somId**. The function takes a **somId** as its argument and returns the **string** that the **somId** represents.
somCompareIds	— Determines whether two **somId** values are the same (that is, represent the same **string**). This function takes two **somId** values as arguments. It returns TRUE (1) if the **somId**s represent the same **string**, or FALSE (0) otherwise.
somCheckId	— Determines whether SOM already knows a **somId**. The function takes a **somId** as its argument. It verifies whether the **somId** is registered and in normal form, registers it if necessary, and returns the input **somId**.
somRegisterId	— The same as **somCheckId,** except it returns TRUE (1) if this is the first time the **somId** has been registered, or FALSE (0) otherwise.
somUniqueKey	— Finds the unique key for a **somId.** The function takes a **somId** identifier as its argument, and returns the unique key for the **somId** — a number that uniquely represents the **string** that the **somId** represents. This key is the same as the key for another **somId** if and only if the other **somId** refers to the same **string** as the input **somId**.

somTotalRegIds — Finds the total number of **somId**s that have been registered, as an **unsigned long**. This function is used to determine an appropriate argument to **somSetExpectedIds**, below, in later executions of the program. The function takes no input arguments.

somSetExpectedIds — Indicates how many unique **somId**s SOM can expect to use during program execution, which, if accurate, can improve the space and time utilization of the program slightly. This routine must be called before the SOM run-time environment is initialized (that is, before the function **somEnvironmentNew** is invoked and before any objects are created). This is the only SOM function that can be invoked before the SOM run-time environment is initialized. The input argument is an **unsigned long**. The function has no return value.

somBeginPersistentIds and **somEndPersistentIds**
— Delimit a time interval for the current thread during which it is guaranteed that (a) any new **somId**s that are created will refer only to static strings and (b) these strings will not be subsequently modified or freed. These functions are useful because **somId**s that are registered within a "persistent ID interval" can be handled more efficiently.

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific function.

Using SOM Classes in Client Programs   **3-35**

Chapter 4. SOM IDL and the SOM Compiler

This chapter first discusses how to define SOM classes and then describes the SOM Compiler. To allow a class of objects to be implemented in one programming language and used in another (that is, to allow a SOM class to be language neutral), the interface to objects of this class must be specified separately from the objects' implementation.

To summarize: As a first step, a file known as the .idl file is used to declare classes and their methods, using SOM's language-neutral Interface Definition Language (IDL). Next, the SOM Compiler is run on the .idl file to produce a template implementation file that contains stub method procedures for the new and overridden methods; this preliminary code corresponds to the computer language that will implement the class. Then, the class implementor fills in the stub procedures with code that implements the methods (or redefines overridden methods) and sets instance data. (This implementation process is the subject of Chapter 5, "Implementing Classes in SOM.") At this point, the implementation file can be compiled and linked with a client program that uses it (as described in Chapter 3, "Using SOM Classes in Client Programs").

Syntax for SOM IDL and the SOM Compiler are presented in this chapter, along with along with helpful information for using them correctly.

4.1 Interface vs Implementation

The *interface* to a class of objects contains the information that a client must know to use an object — namely, the names of its <u>attributes</u> and the signatures of its <u>methods</u>. The interface is described in a formal language independent of the programming language used to implement the object's methods. In SOM, the formal language used to define object interfaces is the **Interface Definition Language** (IDL), standardized by CORBA.

The *implementation* of a class of objects (that is, the procedures that implement methods and the variables used to store an object's state) is written in the implementor's preferred programming language. This language can be object-oriented (for instance, C++) or procedural (for instance, C).

A completely implemented class definition, then, consists of two main files:

- An IDL specification of the <u>interface</u> to instances of the class — the *interface definition file* (or .idl file) and

- <u>Method procedures</u> written in the implementor's language of choice — the *implementation file*.

The interface definition file has a .idl extension, as noted. The implementation file, however, has an extension specific to the language in which it is written. For example, implementations written in C have a .c extension, and implementations written in C++ have a .C (for AIX) or .cpp (for OS/2 or Windows) extension.

To assist users in implementing SOM classes, the SOMobjects Toolkit provides a SOM Compiler. The SOM Compiler takes as input an object interface definition file (the .idl file) and produces a set of *binding files* that make it convenient to implement and use a SOM class whose instances are objects that support the defined interface. The binding files and their purposes are as follows:

- An *implementation template* that serves as a guide for how the implementation file for the class should look. The class implementor fills in this template file with language-specific code to implement the methods that are available on the class' instances.

- *Header files* to be included (a) in the class's implementation file and (b) in client programs that use the class.

These binding files produced by the SOM Compiler bridge the gap between SOM and the object model used in object-oriented languages (such as C++), and they allow SOM to be used with non-object-oriented languages (such as C). The SOM Compiler currently produces binding files for the C and C++ programming languages. SOM can also be used with other programming languages; the bindings simply offer a more convenient programmer's interface to SOM. Vendors of other languages may also offer SOM bindings; check with your language vendor for possible SOM support.

The subsequent sections of this chapter provide full syntax for SOM IDL and the SOM Compiler.

4.2 SOM Interface Definition Language

This section describes the syntax of SOM's **Interface Definition Language (SOM IDL)**. SOM IDL complies with CORBA's standard for IDL; it also adds constructs specific to SOM. (For more information on the CORBA standard for IDL, see *The Common Object Request Broker: Architecture and Specification*, published by Object Management Group and x/Open.) The full grammar for SOM IDL is given in Appendix C. Instructions for converting existing OIDL-syntax files to IDL are given in Appendix B. The current section describes the syntax and semantics of SOM IDL using the following conventions:

Constants (words to be used literally, such as keywords) appear in **bold**.
User-supplied elements appear in *italics*.
{ } Groups related items together as a single item.
[ ] Encloses an optional item.
* Indicates zero or more repetitions of the preceding item.
+ Indicates one or more repetitions of the preceding item.
| Separates alternatives.
_ Within a set of alternatives, an underscore indicates the default, if defined.

IDL is a formal language used to describe object interfaces. Because, in SOM, objects are implemented as instances of classes, an IDL object interface definition specifies for a class of objects what methods (operations) are available, their return types, and their parameter types. For this reason, we often speak of an IDL specification for a class (as opposed to simply an object interface). Constructs specific to SOM discussed below further strengthen this connection between SOM classes, and the IDL language.

IDL generally follows the same lexical rules as C and C++, with some exceptions. In particular:

- IDL uses the ISO Latin-1 (8859.1) character set (as per the CORBA standard).
- White space is ignored except as token delimiters.
- C and C++ comment styles are supported.
- IDL supports standard C/C++ preprocessing, including macro substitution, conditional compilation, and source file inclusion.
- Identifiers (user-defined names for methods, attributes, instance variables, and so on) are composed of alphanumeric and underscore characters (with the first character alphabetic) and can be of arbitrary length, up to an operating-system limit of about 250 characters.
- Identifiers must be spelled consistently with respect to case throughout a specification.
- Identifiers that differ only in case yield a compilation error.
- There is a single name space for identifiers (thus, using the same identifier for a constant and a class name within the same naming scope, for example, yields a compilation error).
- Integer, floating point, character, and string literals are defined just as in C and C++.

The terms listed in Table 1 on the following page are reserved keywords and may not be used otherwise. Keywords must be spelled using upper- and lower-case characters exactly as shown in the table. For example, "void" is correct, but "Void" yields a compilation error.

A typical IDL specification for a single class, residing in a single .idl file, has the following form. (Also see the later section, "Defining multiple interfaces in a .idl file.") The order is unimportant, except that names must be declared (or forward referenced) before they are referenced. The subsequent topics of this section describe the requirements for these specifications:

Include directives   (optional)
Type declarations   (optional)
Constant declarations   (optional)
Exception declarations   (optional)
Interface declaration   (optional)
Module declaration   (optional)

```
Table 1.  KEYWORDS FOR SOM IDL

any                FALSE              readonly
attribute          float              sequence
boolean            implementation     short
case               in                 string
char               inout              struct
class              interface          switch
const              long               TRUE
context            module             TypeCode
default            octet              typedef
double             oneway             unsigned
enum               out                union
exception          raises             void
```

Include directives

The IDL specification for a class normally contains **#include** statements that tell the SOM Compiler where to find the interface definitions (the .idl files) for:

- Each of the class's parent (direct base) classes, and
- The class's metaclass (if specified).

The **#include** statements must appear in the above order. For example,
if class "C" has parents "foo" and "bar" and metaclass "meta",
then file "C.idl" must begin with the following **#include** statements:

```
#include <foo.idl>
#include <bar.idl>
#include <meta.idl>
```

As in C and C++, if a filename is enclosed in angle brackets (< >), the search for the file will begin in system-specific locations. If the filename appears in double quotation marks (""), the search for the file will begin in the current working directory, then move to the system-specific locations.

Type and constant declarations

IDL specifications may include type declarations and constant declarations as in C and C++, with the restrictions/extensions described below. [Note: Readers unfamiliar with C might wish to consult *The C Programming Language* (2nd edition, 1988, Prentice Hall) by Brian W. Kernighan and Dennis M. Ritchie. See pages 36–40 for a discussion of type and constant declarations.]

IDL supports the following basic types (these basic types are also defined for C and C++ client and implementation programs, using the SOM bindings):

Integral types

IDL supports only the integral types **short**, **long**, **unsigned short**, and **unsigned long**, which represent the following value ranges:

short	$-2^{15} \mathrel{..} 2^{15}-1$
long	$-2^{31} \mathrel{..} 2^{31}-1$
unsigned short	$0 \mathrel{..} 2^{16}-1$
unsigned long	$0 \mathrel{..} 2^{32}-1$

Floating point types

IDL supports the **float** and **double** floating-point types. The **float** type represents the IEEE single-precision floating-point numbers; **double** represents the IEEE double-precision floating-point numbers. Note for Windows: Since returning floats and doubles by value may not be compatible across Windows compilers, client programs should return floats and doubles by reference.

Character type

IDL supports a **char** type, which represents an 8-bit quantity. The ISO Latin-1 (8859.1) character set defines the meaning and representation of graphic characters. The meaning and representation of null and formatting characters is the numerical value of the character as defined in the ASCII (ISO 646) standard. Unlike C/C++, type **char** cannot be qualified as signed or unsigned. (The **octet** type, below, can be used in place of unsigned char.)

Boolean type

IDL supports a **boolean** type for data items that can take only the values TRUE and FALSE.

Octet type

IDL supports an **octet** type, an 8-bit quantity guaranteed not to undergo conversion when transmitted by the communication system. The octet type can be used in place of the unsigned char type.

Any type

IDL supports an **any** type, which permits the specification of values of any IDL type. In the SOM C and C++ bindings, the **any** type is mapped onto the following **struct**:

```
typedef struct any {
    TypeCode _type;
    void *_value;
} any;
```

The "_value" member for an **any** type is a pointer to the actual value. The "_type" member is a pointer to an instance of a **TypeCode** that represents the type of the value. The **TypeCode** provides functions for obtaining information about an IDL type. Chapter 7, "The Interface Repository Framework," describes **TypeCode**s and their associated functions.

Constructed types

In addition to the above basic types, IDL also supports three **constructed** types: **struct, union**, and **enum.** The structure and enumeration types are specified in IDL the same as they are in C and C++ [Kernighan–Ritchie references: struct, p. 128; union, p. 147; enum, p. 39], with the following restrictions:

Unlike C/C++, recursive type specifications are allowed only through the use of the **sequence** template type (see below).

Unlike C/C++, structures, discriminated unions, and enumerations in IDL must be tagged. For example, "struct { int a; ... }" is an invalid type specification. The tag introduces a new type name.

In IDL, constructed type definitions need not be part of a **typedef** statement; furthermore, if they are part of a typedef statement, the tag of the struct must differ from the type name being defined by the typedef. For example, the following are valid IDL **struct** and **enum** definitions:

```
struct myStruct {
    long x;
    double y;
};                              /* defines type name myStruct */

enum colors { red, white, blue };  /* defines type name colors */
```

By contrast, the following IDL definitions are *not* valid:

```
typedef struct myStruct {       /*  NOT VALID  */
    long x;
    double y;
} myStruct;                     /* myStruct has been redefined */

typedef enum colors { red, white, blue } colors;  /* NOT VALID */
```

The valid IDL **struct** and **enum** definitions shown above are translated by the SOM Compiler into the following definitions in the C and C++ bindings, assuming they were declared within the scope of interface "Hello":

```
typedef struct Hello_myStruct {  /* C/C++ bindings for IDL struct */
   long x;
   double y;
} Hello_myStruct;

typedef unsigned long Hello_colors; /* C/C++ bindings for IDL enum
*/
#define Hello_red 1UL
#define Hello_white 2UL
#define Hello_blue 3UL
```

When an enumeration is defined within an interface statement for a class, then within C/C++ programs, the enumeration names must be referenced by prefixing the class name. For example, if the *colors* enum, above, were defined within the interface statement for class *Hello*, then the enumeration names would be referenced as *Hello_red, Hello_white*, and *Hello_blue*. Notice the first identifier in an enumeration is assigned the value 1.

Note for Windows: Since returning structs may not be compatible across all compilers, client programs should, in general, return an object that contains a struct. For more information, see "General guidelines for class library designers" in Chapter 5, "Implementing Classes in SOM."

All types and constants generated by the SOM Compiler are *fully qualified*. That is, prepended to them is the fully qualified name of the interface or module in which they appear. For example, consider the following fragment of IDL:

```
module M {
    typedef long long_t;
    module  N {
        typedef long long_t;
        interface I {
            typedef long long_t;
        };
    };
};
```

That specification would generate the following three types:

```
typedef long  M_long_t;
typedef long  M_N_long_t;
typedef long  M_N_I_long_t;
```

For programmer convenience, the SOM Compiler also generates shorter bindings, without the interface qualification. Consider the next IDL fragment:

```
module M {
    typedef long long_t;
    module  N {
        typedef short short_t;
        interface I {
            typedef char char_t;
        };
    };
};
```

In the C/C++ bindings of the preceding fragment, you can refer to "M_long_t" as "long_t", to "M_N_short_t" as "short_t", and to "M_N_I_char_t" as "char_t". However, these shorter forms are available *only* when their interpretation is not ambiguous. Thus, in the first example the shorthand for "M_N_I_long_t" would not be allowed, since it clashes with "M_long_t" and "M_N_long_t". If these shorter forms are not required, they can be ignored by setting `#define SOM_DONT_USE_SHORT_NAMES` before including the public header files, or by using the SOM Compiler option `-mnouseshort` so that they are not generated in the header files.

In the SOM documentation and samples, both long and short forms are illustrated, for both type names and method calls. It is the responsibility of each user to adopt a style according to personal preference. It should be noted, however, that CORBA specifies that only the long forms must be present.

Union type

IDL also supports a **union** type, which is a cross between the C *union* and *switch* statements. The syntax of a **union** type declaration is as follows:

> **union** *identifier* **switch (** *switch–type* **)**
>   **{** *case+* **}**

The "identifier" following the **union** keyword defines a new legal type. (**Union** types may also be named using a **typedef** declaration.) The "switch–type" specifies an integral, character, boolean, or enumeration type, or the name of a previously defined integral, boolean, character, or enumeration type. Each "case" of the **union** is specified with the following syntax:

> *case–label+   type–spec   declarator* ;

where "type-spec" is any valid type specification; "declarator" is an identifier, an array declarator (such as, `foo[3][5]`), or a pointer declarator (such as, `*foo`); and each "case-label" has one of the following forms:

> **case** *const–expr***:**
> **default:**

The "const-expr" is a constant expression that must match or be automatically castable to the "switch-type". A **default** case can appear no more than once.

Unions are mapped onto C/C++ **structs**. For example, the following IDL declaration:

```
union Foo switch (long) {
  case 1: long x;
  case 2: float y;
  default: char z;
};
```

is mapped onto the following C struct:

```
typedef Hello_struct {
  long _d;
  union {
    long x;
    float y;
    char z;
  } _u;
} Hello_foo;
```

The discriminator is referred to as "_d", and the union in the struct is referred to as "_u". Hence, elements of the union are referenced just as in C:

```
Foo v;

/* get a pointer to Foo in v: */
switch(v->_d) {
  case 1: printf("x = %ld\n", v->_u.x); break;
  case 2: printf("y = %f\n", v->_u.y); break;
  default: printf("z = %c\n", v->_u.z); break;
}
```

Note: This example is from *The Common Object Request Broker: Architecture and Specification,* revision 1.1, page 90.

Template types (sequences and strings)

IDL defines two template types not found in C and C++: **sequences** and **strings**. A **sequence** is a one-dimensional array with two characteristics: a maximum size (specified at compile time) and a length (determined at run time). **Sequences** permit passing unbounded arrays between objects. **Sequences** are specified as follows:

> **sequence <** *simple–type* [**,** *positive–integer–const*] **>**

where "simple-type" specifies any valid IDL type, and the optional "positive-integer-const" is a constant expression that specifies the maximum size of the **sequence** (as a positive integer).

Note: The "simple–type" cannot have a '*****' directly in the sequence statement. Instead, a typedef for the pointer type must be used. For example, instead of:

```
typedef sequence<long *> seq_longptr;   // Error: '*' not allowed.
```

use:

```
typedef long * longptr;
typedef sequence<longptr> seq_longptr;  // Ok.
```

In SOM's C and C++ bindings, **sequences** are mapped onto **structs** with the following members:

> **unsigned long _maximum;**
> **unsigned long _length;**
> *simple–type* ***_buffer;**

where "simple-type" is the specified type of the **sequence**. For example, the IDL declaration

```
typedef sequence<long, 10> vec10;
```

results in the following C **struct**:

```
#ifndef _IDL_SEQUENCE_long_defined
#define _IDL_SEQUENCE_long_defined
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    long *_buffer;
} _IDL_SEQUENCE_long;
#endif /* _IDL_SEQUENCE_long_defined */
typedef _IDL_SEQUENCE_long vec10;
```

and an instance of this type is declared as follows:

```
vec10 v = {10L, 0L, (long *)NULL};
```

The "_maximum" member designates the actual size of storage allocated for the **sequence**, and the "_length" member designates the number of values contained in the "_buffer" member. For bounded **sequences**, it is an error to set the "_length" or "_maximum" member to a value larger than the specified bound of the **sequence**.

Before a **sequence** is passed as the value of an "in" or "inout" method parameter, the "_buffer" member must point to an array of elements of the appropriate type, and the "_length" member must contain the number of elements to be passed. (If the parameter is "inout" and the **sequence** is unbounded, the "_maximum" member must also be set to the actual size of the array. Upon return, "_length" will contain the number of values copied into "_buffer", which must be less than "_maximum".) When a **sequence** is passed as an "out" method parameter or received as the return value, the method procedure allocates storage for "_buffer" as needed, the "_length" member contains the number of elements returned, and the "_maximum" member contains the number of elements allocated. (The client is responsible for subsequently freeing the memory pointed to by "_buffer".)

C and C++ programs using SOM's language bindings can refer to **sequence** types as:

> **_IDL_SEQUENCE_***type*

where "type" is the effective type of the **sequence** members. For example, the IDL type `sequence<long,10>` is referred to in a C/C++ program by the type name `_IDL_SEQUENCE_long`. If `longint` is defined via a typedef to be type `long`, then the IDL type `sequence<longint,10>` is also referred to by the type name `_IDL_SEQUENCE_long`.

If the typedef is for a pointer type, then the effective type is the name of the pointer type. For example, the following statements define a C/C++ type `_IDL_SEQUENCE_longptr` and *not* `_IDL_SEQUENCE_long`:

```
typedef long * longptr;
typedef sequence<longptr> seq_longptr;
```

A **string** is similar to a **sequence** of type **char.** It can contain all possible 8-bit quantities except NULL. **Strings** are specified as follows:

> **string [** < *positive–integer–const* > **]**

where the optional "positive-integer-const" is a constant expression that specifies the maximum size of the **string** (as a positive integer, which does not include the extra byte to hold a NULL as required in C/C++). In SOM's C and C++ bindings, **strings** are mapped onto zero-byte terminated character arrays. The length of the string is encoded by the position of the zero-byte. For example, the following IDL declaration:

```
typedef string<10> foo;
```

is converted to the following C/C++ **typedef**:

```
typedef char *foo;
```

A variable of this type is then declared as follows:

```
foo s = (char *) NULL;
```

C and C++ programs using SOM's language bindings can refer to **string** types by the type name *string*.

When an unbounded **string** is passed as the value of an "inout" method parameter, the returned value is constrained to be no longer than the input value. Hence, using unbounded **string**s as "inout" parameters is not advised.

Arrays

Multidimensional, fixed-size arrays can be declared in IDL as follows:

> *identifier* { **[** *positive–integer–const* **]** }+

where the "positive-integer-const" is a constant expression that specifies the array size, in each dimension, as a positive integer. The array size is fixed at compile time.

Pointers

Although the CORBA standard for IDL does not include them, SOM IDL offers pointer types. Declarators of a pointer type are specified as in C and C++:

> *type* *declarator

where "type" is a valid IDL type specification and "declarator" is an identifier or an array declarator.

Object types

The name of the interface to a class of objects can be used as a type. For example, if an IDL specification includes an **interface** declaration (described below) for a class (of objects) "C1", then "C1" can be used as a type name within that IDL specification. When used as a type, an interface name indicates a pointer to an object that supports that interface. An interface name can be used as the type of a method argument, as a method return type, or as the type of a member of a constructed type (a **struct, union**, or **enum**). In all cases, the use of an interface name implicitly indicates a pointer to an object that supports that interface.

As explained in Chapter 3, SOM's C usage bindings for SOM classes also follow this convention. However, within SOM's C++ bindings, the pointer is made explicit, and the use of an interface name as a type refers to a class instance itself, rather than a pointer to a class instance. For example, to declare a variable "myobj" that is a pointer to an instance of class "Foo" in an IDL specification and in a C program, the following declaration is required:

```
Foo myobj;
```

However, in a C++ program, the following declaration is required:

```
Foo *myobj;
```

If a C programmer uses the SOM Compiler option **–maddstar,** then the bindings generated for C will also require an explicit '*' in declarations. Thus,

```
Foo myobj;        in IDL requires
```

```
Foo *myobj;       in both C and C++ programs.
```

This style of bindings for C is permitted for two reasons:

- It more closely resembles the bindings for C++, thus making it easier to change to the C++ bindings at a later date; and

- It is required for compatibility with existing SOM OIDL code.

Note: The same C and C++ binding emitters should *not* be run in the same SOM Compiler command. For example,

```
sc "-sh;xh" cls.idl    // Not valid.
```

If you wish to generate both C and C++ bindings, you should issue the commands separately:

```
sc -sh cls.idl
sc -sxh cls.idl
```

Exception declarations

IDL specifications may include **exception** declarations, which define data structures to be returned when an exception occurs during the execution of a method. (IDL exceptions are implemented by simply passing back error information after a method call, as opposed to the "catch/throw" model where an exception is implemented by a long jump or signal.) Associated with each type of exception is a name and, optionally, a struct-like data structure for holding error information. Exceptions are declared as follows:

> **exception** *identifier* **{** *member* **};**

The "identifier" is the name of the exception, and each "member" has the following form:

> *type–spec declarators ;*

where "type-spec" is a valid IDL type specification and "declarators" is a list of identifiers, array declarators, or pointer declarators, delimited by commas. The members of an exception structure should contain information to help the caller understand the nature of the error. The exception declaration can be treated like a **struct** definition; that is, whatever you can access in an IDL **struct**, you can access in an **exception** declaration. Alternatively, the structure can be *empty*, whereby the exception is just identified by its name.

If an **exception** is returned as the outcome of a method, the exception "identifier" indicates which exception occurred. The values of the members of the exception provide additional information specific to the exception. The topic "Method declarations" below describes how to indicate that a particular method may raise a particular exception, and Chapter 3, "Using SOM Classes in Client Programs," describes how exceptions are handled, in the section entitled "Exceptions and error handling."

Following is an example declaration of a "BAD_FLAG" exception:

```
exception BAD_FLAG { long ErrCode; char Reason[80]; };
```

The SOM Compiler will map the above exception declaration to the following C language constructs:

```
#define ex_BAD_FLAG "::BAD_FLAG"
typedef struct BAD_FLAG {
    long  ErrCode;
    char  Reason[80];
} BAD_FLAG;
```

Thus, the `ex_BAD_FLAG` symbol can be used as a shorthand for naming the exception.

An exception declaration within an interface "Hello", such as this:

```
interface Hello {
    exception LOCAL_EXCEPTION { long ErrCode; };
};
```

would map onto:

```
#define ex_Hello_LOCAL_EXCEPTION "::Hello::LOCAL_EXCEPTION"
typedef struct Hello_LOCAL_EXCEPTION {
    long  ErrCode;
} Hello_LOCAL_EXCEPTION;
#define ex_LOCAL_EXCEPTION ex_Hello_LOCAL_EXCEPTION
```

In addition to user-defined exceptions, there are several predefined exceptions for system run-time errors. The standard exceptions as prescribed by CORBA are shown in Table 2. These exceptions correspond to standard run-time errors that may occur during the execution of any method (regardless of the list of exceptions listed in its IDL specification).

Each of the standard exceptions has the same structure: an error code (to designate the subcategory of the exception) and a completion status code. For example, the `NO_MEMORY` standard exception has the following definition:

```
enum completion_status {YES, NO, MAYBE};
exception NO_MEMORY { unsigned long minor;
                      completion_status completed; };
```

The "completion_status" value indicates whether the method was never initiated (NO), completed its execution prior to the exception (YES), or the completion status is indeterminate (MAYBE).

Since all the standard exceptions have the same structure, **somcorba.h** (included by **som.h**) defines a generic **StExcep** typedef which can be used instead of the specific typedefs:

```
typedef struct StExcep {
    unsigned long minor;
    completion_status completed;
} StExcep;
```

The standard exceptions shown in Table 2 are defined in an IDL module called **StExcep**, in the file called **stexcep.idl**, and the C definitions can be found in **stexcep.h**.

Interface declarations

The IDL specification for a class of objects must contain a declaration of the **interface** these objects will support. Because, in SOM, objects are implemented using classes, the interface name is always used as a class name as well. Therefore, an interface declaration can be understood to specify a class name, and its parent (direct base) class names. This is the approach used in the following description of an interface declaration. In addition to the class name and its parents names, an interface indicates new methods (operations), and any constants, type definitions, and exception structures that the interface exports. An interface declaration has the following syntax:

interface class–name [**:** parent–class1**,** parent–class2, ...]
{
constant declarations   (optional)
type declarations   (optional)
exception declarations   (optional)
attribute declarations   (optional)
method declarations   (optional)
implementation statement   (optional)
};

```
Table 2.  STANDARD EXCEPTIONS DEFINED BY CORBA

module StExcep {

    #define ex_body { unsigned long minor; completion_status completed; }

    enum completion_status { YES, NO, MAYBE };
    enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};

    exception UNKNOWN             ex_body;   // the unknown exception
    exception BAD_PARAM           ex_body;   // an invalid parameter was passed
    exception NO_MEMORY           ex_body;   // dynamic memory allocation failure
    exception IMP_LIMIT           ex_body;   // violated implementation limit
    exception COMM_FAILURE        ex_body;   // communication failure
    exception INV_OBJREF          ex_body;   // invalid object reference
    exception NO_PERMISSION       ex_body;   // no permission for attempted op.
    exception INTERNAL            ex_body;   // ORB internal error
    exception MARSHAL             ex_body;   // error marshalling param/result
    exception INITIALIZE          ex_body;   // ORB initialization failure
    exception NO_IMPLEMENT        ex_body;   // op. implementation unavailable
    exception BAD_TYPECODE        ex_body;   // bad typecode
    exception BAD_OPERATION       ex_body;   // invalid operation
    exception NO_RESOURCES        ex_body;   // insufficient resources for request
    exception NO_RESPONSE         ex_body;   // response to req. not yet available
    exception PERSIST_STORE       ex_body;   // persistent storage failure
    exception BAD_INV_ORDER       ex_body;   // routine invocations out of order
    exception TRANSIENT           ex_body;   // transient failure – reissue request
    exception FREE_MEM            ex_body;   // cannot free memory
    exception INV_IDENT           ex_body;   // invalid identifier syntax
    exception INV_FLAG            ex_body;   // invalid flag was specified
    exception INTF_REPOS          ex_body;   // error accessing interface repository
    exception CONTEXT             ex_body;   // error processing context object
    exception OBJ_ADAPTER         ex_body;   // failure detected by object adapter
    exception DATA_CONVERSION     ex_body;   // data conversion error

};
```

Many class implementors distinguish a "class-name" by using an initial capital letter, but that is optional. The "parent-class" (or base-class) names specify the interfaces from which the interface of "class-name" instances is derived. Parent-class names are required only for the immediate parent(s). Each parent class must have its own IDL specification (which must be *#include*d in the subclass's .idl file). A parent class cannot be named more than once in the **interface** statement header.

Note: In general, an "**interface** *<className>*" header must precede any subsequent implementation that references *<className>*. For more discussion of multiple **interface** statements, refer to the later topic "Defining multiple interfaces in a .idl file."

The following topics describe the various declarations/statements that can be specified within the body of an **interface** declaration. The order in which these declarations are specified is usually optional, and declarations of different kinds can be intermixed. Although all of the declarations/statements are listed above as "optional," in some cases using one of them may mandate another. For example, if a **method** raises an **exception**, the exception structure must be defined beforehand. In general, **types, constants,** and **exceptions,** as well as **interface** declarations, must be defined before they are referenced, as in C/C++.

Constant, type, and exception declarations

The form of a **constant, type**, or **exception** declaration within the body of an **interface** declaration is the same as described previously in this chapter. **Constants** and **types** defined within an **interface** for a class are transferred by the SOM Compiler to the binding files it generates for that class, whereas **constants** and **types** defined outside of an **interface** are not.

Global types (such as, those defined outside of an interface and module) can be emitted by surrounding them with the following **#pragmas**:

```
#pragma somemittypes on
    typedef sequence <long,10> vec10;
    exception BAD_FLAG { long ErrCode; char Reason[80]; };
    typedef long long_t;
#pragma somemittypes off
```

Types, constants, and **exceptions** defined in a parent class are also accessible to the child class. References to them, however, must be unambiguous. Potential ambiguities can be resolved by prefacing a name with the name of the class that defines it, separated by the characters "::" as illustrated below:

```
MyParentClass::myType
```

The child class can redefine any of the **type, constant,** and **exception** names that have been inherited, although this is not advised. The derived class cannot, however, redefine **attributes** or **methods**. It can only replace the implementation of **methods** through overriding (as in example 3 of the Tutorial). To refer to a **constant, type,** or **exception** "name" defined by a parent class and redefined by "class-name," use the "parent-name::name" syntax as before.

Note: A name reference such as `MyParentClass::myType` required in IDL syntax is equivalent to `MyParentClass_myType` in C/C++. For a full discussion of name recognition in SOM, see "Scoping and name resolution" later in this chapter.

Attribute declarations

Declaring an **attribute** as part of an **interface** is equivalent to declaring two accessor methods: one to retrieve the value of the **attribute** (a "get" method, named "_get_<attributeName>") and one to set the value of the **attribute** (a "set" method, named "_set_<attributeName>").

Attributes are declared as follows:

[**readonly**] **attribute** *type–spec   declarators* **;**

where "type-spec" specifies any valid IDL type and "declarators" is a list of identifiers or pointer declarators, delimited by commas. (An array declarator cannot be used directly when declaring an **attribute**, but the type of an attribute can be a user-defined type that is an array.) The optional **readonly** keyword specifies that the value of the **attribute** can be accessed but not modified by client programs. (In other words, a **readonly attribute** has no "set" method.) Below are examples of **attribute** declarations, which are specified within the body of an **interface** statement for a class:

```
interface Goodbye: Hello, SOMObject
{
  void  sayBye();

  attribute short xpos;
  attribute char c1, c2;
  readonly attribute float xyz;
};
```

The preceding **attribute** declarations are equivalent to defining the following methods:

```
short _get_xpos();
void  _set_xpos(in short xpos);
char  _get_c1();
void  _set_c1(in char c1);
char  _get_c2();
void  _set_c2(in char c2);
float _get_xyz();
```

Note: Although the preceding attribute declarations are equivalent to the explicit method declarations above, these method declarations are *not* legal IDL, because the method names begin with an '_'. All IDL identifiers must begin with an alphabetic character, not including '_'.

Attributes are inherited from ancestor classes (indirect base classes). An inherited **attribute** name cannot be redefined to be a different type.

Method (operation) declarations

Method (operation) declarations define the interface of each method introduced by the class. A method declaration is similar to a C/C++ function definition:

[**oneway**] *type–spec identifier* **(** *parameter–list* **)** [*raises–expr*] [*context–expr*] **;**

where "identifier" is the name of the method and "type-spec" is any valid IDL **type** (or the keyword **void**, indicating that the method returns no value). Unlike C and C++ procedures, methods that do not return a result must specify **void** as their return type. The remaining syntax of a method declaration is elaborated in the following subtopics.

Note: Although IDL does not allow methods to receive and return values whose type is a pointer to a function, it does allow methods to receive and return <u>method names</u> (as **string** values). Thus, rather than defining methods that pass pointers to functions (and that subsequently invoke those functions), programmers should instead define methods that pass method names (and subsequently invoke those methods using one of the SOM-supplied method-dispatching or method-resolution methods or functions, such as **somDispatch**).

Oneway keyword

The optional **oneway** keyword specifies that when a client invokes the method, the invocation semantics are "best-effort", which does not guarantee delivery of the call. "Best-effort" implies that the method will be invoked at most once. A **oneway** method should not have any output parameters and should have a return type of **void**. A **oneway** method also should not include a "raises expression" (see below), although it may raise a standard exception.

If the **oneway** keyword is not specified, then the method has "at-most-once" invocation semantics if an exception is raised, and it has "exactly-once" semantics if the method succeeds. This means that a method that raises an exception has been executed zero or one times, and a method that succeeds has been executed exactly once.

Note: Currently the "oneway" keyword, although accepted, has no effect on the C/C++ bindings that are generated.

Parameter list

The "parameter-list" contains zero or more parameter declarations for the method, delimited by commas. (The target object for the method is not explicitly specified as a method parameter in IDL, nor are the **Environment** or **Context** parameters.) If there are no explicit parameters, the syntax "( )" must be used, rather than "(void)". A parameter declaration has the following syntax:

{ **in** | **out** | **inout** } *type–spec declarator*

where "type-spec" is any valid IDL type and "declarator" is an identifier, array declarator, or pointer declarator.

<u>**In, out, inout** parameters</u>: The required **in**|**out**|**inout** directional attribute indicates whether the parameter is to be passed from client to server (**in**), from server to client (**out**), or in both directions (**inout**). A method must not modify an **in** parameter. If a method raises an exception, the values of the return result and the values of the **out** and **inout** parameters (if any) are undefined. When an unbounded **string** or **sequence** is passed as an **inout** parameter, the returned value must be no longer than the input value.

The following are examples of valid method declarations in SOM IDL:

```
short meth1(in char c, out float f);
oneway void meth2(in char c);
float meth3();
```

Classes derived from **SOMObject** can declare methods that take a pointer to a block of memory containing a variable number of arguments, using a final parameter of type **va_list**. The **va_list** must use the parameter name "ap", as in the following example:

```
void MyMethod(in short numArgs, in va_list ap);
```

For **in** parameters of type **array**, C and C++ clients must pass the address of the first element of the array. For **in** parameters of type **struct, union, sequence** or **any**, C/C++ clients must pass the address of a variable of that type, rather than the variable itself.

For all IDL types except **arrays**, if a parameter of a method is **out** or **inout**, then C/C++ clients must pass the address of a variable of that type (or the value of a pointer to that variable) rather than the variable itself. (For example, to invoke method "meth1" above, a pointer to a variable of type **float** must be passed in place of parameter "f".) For **arrays**, C/C++ clients must pass the address of the first element of the **array**.

If the return type of a method is a **struct, union, sequence**, or **any** type, then for C/C++ clients, the method returns the value of the C/C++ struct representing the IDL **struct, union, sequence**, or **any**. If the return type is **string,** then the method returns a pointer to the first character of the **string**. If the return type is **array**, then the method returns a pointer to the first element of the **array**.

The pointers implicit in the parameter types and return types for IDL method declarations are made explicit in SOM's C and C++ bindings. Thus, the stub procedure that the SOM Compiler generates for method "meth1", above, has the following signature:

```
SOM_Scope short  SOMLINK meth1(char c, float *f)
```

For C and C++ clients, if a method has an **out** parameter of type **string, sequence,** or **any**, then the method must allocate the storage for the **string,** for the "_buffer" member of the struct that represents the **sequence,** or for the "_value" member of the struct that represents the **any**. It is then the responsibility of the client program to free the storage when it is no longer needed. Similarly, if the return type of a method is **string**, **sequence**, **any**, or **array**, then storage must be allocated by the method, and the client program is responsible for subsequently freeing it.

Note: The foregoing description also applies for the **_get_**<*attributeName*> method associated with an attribute of type **string**, **sequence**, **any**, or **array**. Hence, the attribute should be specified with a "noget" modifier to override automatic implementation of the attribute's "get" method. Then, needed memory can be allocated by the developer's "get" method implementation and subsequently deallocated by the caller. (The "noget" modifier is described under the topic "Modifier statements" later in this section.)

Raises expression

The optional **raises** expression ("raises-expr") in a method declaration indicates which exceptions the method may raise. (IDL exceptions are implemented by simply passing back error information after a method call, as opposed to the "catch/throw" model where an exception is implemented by a long jump or signal.) A **raises** expression is specified as follows:

raises (*identifier1, identifier2, ...* **)**

where each "identifier" is the name of a previously defined **exception**. In addition to the exceptions listed in the **raises** expression, a method may also signal any of the standard exceptions. Standard exceptions, however, should not appear in a **raises** expression. If no **raises** expression is given, then a method can raise only the standard exceptions. (See the earlier topic "Exception declarations" for information on defining exceptions and for the list of standard exceptions. See Chapter 3, the section entitled "Exceptions and error handling," for information on using exceptions.)

The optional **context expression** ("context-expr") in a method declaration indicates which elements of the client's context the method may consult. A context expression is specified as follows:

> **context (** *identifier1,* *identifier2,* ... **)**

where each "identifier" is a string literal made up of alphanumeric characters, periods, under-scores, and asterisks. (The first character must be alphabetic, and an asterisk can only appear as the last character, where it serves as a wildcard matching any characters. If convenient, identifiers may consist of period-separated valid identifier names, but that form is optional.)

The **Context** is a special object that is specified by the CORBA standard. It contains a property list — a set of property-name/string-value pairs that the client can use to store information about its environment that methods may find useful. It is used in much the same way as environment variables. It is passed as an additional (third) parameter to CORBA-compliant methods that are defined as "context-sensitive" in IDL, along with the CORBA-defined Environment structure.

The **context expression** of a method declaration in IDL specifies which property names the method uses. If these properties are present in the **Context** object supplied by the client, they will be passed to the object implementation, which can access them via the **get_values** method of the **Context** object. However, the argument that is passed to the method having a context expression is a **Context** object, *not* the names of the properties. The client program must either create a **Context** object and use the **set_values** or **set_one_value** method of the **Context** class to set the context properties, or use the **get_default_context** method. The client program then passes the **Context** object in the method invocation. Note that the CORBA standard also allows properties in addition to those in the **context** expression to be passed in the **Context** object.

In Chapter 3, "Using SOM Classes in Client Programs," the topic "Invoking Methods" describes the placement of a **context** parameter in a method call. See also chapter 6 of *The Common Object Request Broker: Architecture and Specification* for a discussion of how clients associate values with **context** identifiers. A description of the **Context** class and its methods is contained in the *SOMobjects Developer Toolkit: Programmers Reference Manual.*

Implementation statements

A SOM IDL interface statement for a class may contain an **implementation** statement, which specifies information about how the class will be implemented (version numbers for the class, overriding of inherited methods, what resolution mechanisms the bindings for a particular method will support, and so forth). If the **implementation** statement is omitted, default informa-tion is assumed.

Because the **implementation** statement is specific to SOM IDL (and is not part of the CORBA standard), the **implementation** statement should be preceded by an "#ifdef _ _SOMIDL_ _" directive and followed by an "#endif" directive. (See Example 3 in the SOM IDL Tutorial presented earlier.)

The syntax for the implementation statement is as follows:

> **#ifdef __SOMIDL__**
> **implementation**
> **{**
>   *implementation**
> **};**
> **#endif**

where each "implementation" can be a **modifier** statement, a **passthru** statement, or a declarator of an **instance variable**, terminated by a semicolon. These constructs are described below. An **interface** statement may *not* contain multiple **implementation** statements.

A **modifier** statement gives additional implementation information about IDL definitions, such as **interfaces**, **attributes**, **methods**, and **types**. Modifiers can be unqualified or qualified.

An **unqualified modifier** is <u>associated with the interface</u> it is defined in. An unqualified modifier statement has the following two syntactic forms:

modifier
modifier **=** *value*

where "modifier" is either a SOM Compiler-defined identifier or a user-defined identifier, and where "value" is an identifier, a string enclosed in double quotes (" "), or a number.

For example:

```
filestem = foo;
nodata;
persistent;
dllname = "E:/som/dlls";
```

A **qualified modifier** is <u>associated with a qualifier</u>. The qualified modifier has the syntax:

qualifier **:** *modifier*
qualifier **:** *modifier* **=** *value*
#pragma modifier *qualifier* **:** *modifier*
#pragma modifier *qualifier* **:** *modifier* **=** *value*

where "qualifier" is the identifier of an IDL definition or is user defined. If the "qualifier" is an IDL definition introduced in the current interface, module, or global scope, then the modifier is attached to that definition. Otherwise, if the qualifier is user defined, the modifier is attached to the interface it occurs in. If a user-defined modifier is defined outside of an interface body (by using **#pragma modifier**), then it is ignored.

For example, consider the following IDL file. (Notice that qualified modifiers can be defined with the "qualifier" and "modifier[=value]" in either order. Also observe that additional modifiers can be included by separating them with commas.)

```
#include <somobj.idl>
#include <somcls.idl>

typedef long newInt;
#pragma somemittypes on
#pragma modifier newInt : nonportable;
#pragma somemittypes off
module M {
    typedef long long_t;
    module  N {
        typedef short short_t;
        interface M_I : SOMClass {
            implementation {
                somInit : override;
            };
        };
        interface I : SOMObject {
            void op ();
            #pragma modifier op : persistent;
            typedef char char_t;
            implementation {
                releaseorder : op;
                metaclass = M_I;
                callstyle = oidl;
                mymod : a, b;
                mymod : c, d;
                e    : mymod;
                f    : mymod;
                op : persistent;
            };
        };
    };
};
```

From the preceding IDL file, modifiers are associated with the following definitions:

```
TypeDef ":newInt"                1  modifier: nonportable
InterfaceDef ":M:N:M_I"          1  modifier: override = somInit
InterfaceDef ":M:N:I"            9 modifiers: metaclass = M_I,
                                             releaseorder = op
                                             callstyle = oidl
                                             mymod = a,b,c,d,e,f
                                             a = mymod
                                             b = mymod
                                             c = mymod
                                             d = mymod
                                             e = mymod
                                             f = mymod
OperationDef ":M:N:I:op"         1  modifier: persistent
```

Notice how the modifiers for the user-defined qualifier "mymod":

```
mymod : a, b;
mymod : c, d;
e     : mymod;
f     : mymod;
```

map onto:

```
mymod = a,b,c,d,e,f
a       = mymod
b       = mymod
c       = mymod
d       = mymod
e       = mymod
f       = mymod
```

This enables users to look up the modifiers with "mymod", either by looking for "mymod" or by using each individual value that uses "mymod". These user-defined modifiers are available for Emitter writers (see the *Emitter Framework Guide and Reference*) and from the Interface Repository (see Chapter 7, "The Interface Repository Framework").

SOM Compiler unqualified modifiers

Unqualified modifiers (described below) include the SOM Compiler-defined identifiers: **abstract, baseproxyclass, callstyle, classinit, directinitclasses, dllname, filestem, functionprefix, majorversion, metaclass, memory_management, minorversion, somallocate,** and **somdeallocate.**

abstract	— Specifies that the class is intended for use as a parent for subclass derivations, but not for creating instances.
baseproxyclass = *class*	— Specifies the base proxy class to be used by DSOM when dynamically creating a proxy class for the current class. The base proxy class must be derived from the class **SOMDClientProxy.** The **SOMDClientProxy** class will be used if the **baseproxyclass** modifier is unspecified. (See Chapter 6, "Distributed SOM," for a discussion on customizing proxy classes.)
callstyle = oidl	— Specifies that the method stub procedures generated by SOM's C/C++ bindings will not include the CORBA-specified (*Environment *ev*) and (*context *ctx*) parameters.
classinit = *procedure*	— Specifies a user-written procedure that will be executed to initialize the class object when it is created. If the **classinit** modifier is specified in the .idl file for a class, the SOM Compiler will provide a template for the procedure in the implementation file it generates. The class implementor can then fill in the body of this procedure template.

directinitclasses = "*ancestor1, ancestor2, ...***"**

— Specifies the ancestor class(es) whose initializers (and destructors) will be directly invoked by this class's initialization (and destruction) routines. If this modifier is not explicitly specified, the default setting is the parents of the class. For further information, see "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM."

dllname = *filename*

— Specifies the name of the library file that will contain the class's implementation. If *filename* contains special characters (e.g., periods, backslashes), then *filename* should be surrounded by double quotes (""). The *filename* specified can be either a full pathname, or an unqualified (or partially qualified) filename. In the latter cases, the environment variable LIBPATH (on AIX or OS/2) or PATH (on Windows) is used to locate the file.

**filestem = ** *stem*

— Specifies how the SOM Compiler will construct file names for the binding files it generates (<stem>.h, <stem>.c, etc.). The default stem is the file stem of the .idl file for the class.

**functionprefix = ** *prefix*

— Directs the SOM Compiler to construct method-procedure names by prefixing method names with "prefix". For example, "functionprefix = xx;" within an **implementation** statement would result in a procedure name of `xxfoo` for method `foo`. The default for this attribute is the empty string. If an interface is defined in a module, then the default function prefix is the fully scoped interface name. *Tip*: Using a function prefix with the same name as the class makes it easier to remember method-procedure names when debugging.

When an .idl file defines multiple interfaces not contained within a module, use of a function prefix for each interface is essential to avoid name collisions. For example, if one interface introduces a method and another interface in the same .idl file overrides it, then the implementation file for the classes will contain two method procedures of the same name (unless function prefixes are defined for one of the classes), resulting in a name collision at compile time.

majorversion = *number*

— Specifies the major version number of the current class definition. The major version number of a class definition usually changes only when a significant enhancement or incompatible change is made to the class. The "number" must be a positive integer less than $2^{31}-1$. If a non-zero major version number is specified, SOM will verify that any code that purports to implement the class has the same major version number. The default major version number is zero.

memory_management = corba

— Specifies that all methods introduced by the class follow the CORBA specification for parameter memory management, except where a particular method has an explicit modifier indicating otherwise (either "object_owns_result" or "object_owns_parameters"). See the section in Chapter 6 entitled "Memory Management" for a discussion of the CORBA memory-management requirements.

metaclass = *class*	— Specifies the class's metaclass. The specified metaclass (or one automatically derived from it at run time) will be used to create the class object for the class. If a **metaclass** is specified, its .idl file (if separate) must be included in the **include** section of the class's .idl file. If no metaclass is specified, the metaclass will be defined automatically.
minorversion = *number*	— Specifies the minor version number of the current class definition. The minor version number of a class definition changes whenever minor enhancements or fixes are made to a class. Class implementors usually maintain backward compatibility across changes in the minor version number. The "number" must be a positive integer less than $2^{31}-1$. If a non-zero minor version number is specified, SOM will verify that any code that purports to implement the class has the same or a higher minor version number. The default minor version number is zero.
somallocate = *procedure*	— Specifies a user-written procedure that will be executed to allocate memory for class instances when the **somAllocate** class method is invoked.
somdeallocate = *procedure*	
	— Specifies a user-written procedure that will be executed to deallocate memory for class instances when the **somDeallocate** class method is invoked.

The following example illustrates the specification of unqualified interface modifiers:

```
implementation
{
  filestem = hello;
  functionprefix = hel;
  majorversion = 1;
  minorversion = 2;
  classinit = helloInit;
  metaclass = M_Hello;
};
```

SOM Compiler qualified modifiers

Qualified modifiers are categorized according to the IDL component (class, attribute, method, or type) to which each modifier applies. Listed below are the SOM Compiler-defined identifiers used as qualified modifiers, along with the IDL component to which it applies. Descriptions of all qualified modifiers are then given in alphabetical order. Recall that qualified modifiers are defined using the syntax *qualifier* : *modifier*[**=***value*].

For classes:
   releaseorder

For attributes:
   indirect, nodata, noget, noset, and **persistent**

For methods:
   caller_owns_parameters, caller_owns_result,
   const, init, method, migrate, namelookup,
   nocall, noenv, nonstatic, nooverride, noself,
   object_owns_parameters, object_owns_result,
   offset, override, procedure, reintroduce, and **select**

For variables:
   staticdata

For types:
   impctx

The following paragraphs describe each qualified modifier.

caller_owns_parameters = "*p1, p2, ..., pn*"

— Specifies the names of the method's parameters whose ownership is retained by (in the case of "in" parameters) or transferred to (for "inout" or "out" parameters) the caller. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense for parameters whose IDL type is a data item that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any), or a struct or union.

For parameters whose type is an object, ownership applies to the object *reference* rather than to the object (that is, the caller should invoke **release** on the parameter, rather than **somFree**).

caller_owns_result

— Specifies that ownership of the return result of the method is transferred to the caller, and that the caller is responsible for freeing the memory. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense when the method's return type is a data type that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any). For methods that return an object, ownership applies to the object reference rather than to the object (that is, the caller should invoke **release** on the result, rather than **somFree**).

const

— Indicates that implementations of the related method should not modify their target argument. SOM provides no way to verify or guarantee that implementations do not modify the targets of such methods, and the information provided by this modifier is not currently of importance to any of the Toolkit emitters. However, the information may prove useful in the future. For example, since modifiers are available in the Interface Repository, there may be future uses of this information by DSOM.

impctx

— Supports types that cannot be fully defined using IDL. For full information, see "Using the tk_foreign TypeCode" in Chapter 7, "The Interface Repository Framework."

indirect

— Directs the SOM Compiler to generate "get" and "set" methods for the attribute that take and return a pointer to the attribute's value, rather than the attribute value itself. For example, if an attribute x of type float is declared to be an indirect attribute, then the "_get_x" method will return a pointer to a float, and the input to the "_set_x" method must be a pointer to a float. (This modifier is provided for OIDL compatibility only.)

init

— Indicates that a method is an initializer method. For information concerning the use of this modifier, see "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM."

method or **nonstatic** or **procedure**

— Indicates the category of method implementation. Refer to the topic "The four kinds of SOM methods" in Chapter 5, "Implementing Classes in SOM," for an explanation of the meanings of these different method modifiers. If none of these modifiers is specified, the default is **method**. Methods with the **procedure** modifier cannot be invoked remotely using DSOM.

migrate = *ancestor*

— Indicates that a method originally introduced by this interface has been moved upward to a specified *<ancestor>* interface. When this is done, the method introduction must be removed from this interface (because the method is now inherited). However, the original **releaseorder** entry for the method should be retained, and **migrate** should be used to assure that clients compiled based on the original interface will not require recompilation. The ancestor interface is specified using a C–scoped interface name. For example, "Module_InterfaceName", not "Module::InterfaceName". See the later topic "Name usage in client programs" for an explanation of C-scoped names.

namelookup

— See "**offset** or **namelookup**."

nocall

— Specifies that the related method should not be invoked on an instance of this class even though it is supported by the interface.

nodata

— Directs the SOM Compiler *not* to define an instance variable corresponding to the attribute. For example, a "time" attribute would not require an instance variable to maintain its value, because the value can be obtained from the operating system. The "get" and "set" methods for "nodata" attributes must be defined by the class implementor; stub method procedures for them are automatically generated in the implementation template for the class by the SOM Compiler.

noenv

— Indicates that a direct-call procedure does not receive an environment as an argument.

noget

— Directs the SOM Compiler *not* to automatically generate a "get" method procedure for the attribute in the .ih/.xih binding file for the class. Instead, the "get" method must be implemented by the class implementor. A stub method procedure for the "get" method is automatically generated in the implementation template for the class by the SOM Compiler, to be filled in by the implementor.

nonstatic

— See "**method** or **nonstatic** or **procedure**."

nooverride

— Indicates that the method should not be overridden by subclasses. The SOM Compiler will generate an error if this method is overridden.

noself

— Indicates that a direct-call procedure does not receive a target object as an argument.

noset

— Directs the SOM Compiler *not* to automatically generate a "set" method procedure for the attribute in the .ih/.xih binding file for the class. Instead, the "set" method must be implemented by the class implementor. A stub method procedure for the "set" method is automatically generated in the implementation template for the class by the SOM Compiler.

Note: The "set" method procedure that the SOM Compiler generates by default for an attribute in the .h/.xh binding file (when the **noset** modifier is *not* used) does a shallow copy of the value that is passed to the attribute. For some attribute types, including strings and pointers, this may not be appropriate. For instance, the "set" method for an attribute of type **string** should perform a string copy, rather than a shallow copy, if the attribute's value may be needed after the client program has freed the memory occupied by the string. In such situations, the class implementor should specify the **noset** attribute modifier and implement the attribute's "set" method manually, rather than having SOM implement the "set" method automatically.

object_owns_parameters = "*p1, p2, ..., pn*"
— Specifies the names of the method's parameters whose ownership is transferred to (in the case of "in" parameters) or is retained by (for "inout" or "out" parameters) the object. For "in" parameters, the object can free the parameter at any time after receiving it. (Hence, the caller should not reuse the parameter or pass it as any other object-owned parameter in the same method call.) For "inout" and "out" parameters, the object is responsible for freeing the parameter sometime before the object is destroyed. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense for parameters whose IDL type is a data item that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any), or a struct or union.

For parameters whose type is an object, ownership applies to the object *reference* rather than to the object (that is, the object will invoke **release** on the parameter, rather than **somFree**). For "in" and "out" parameters whose IDL-to-C/C++ mapping introduces a pointer, ownership applies only to the data item itself, and not to the introduced pointer. (For example, even if an "out string" IDL parameter (which becomes a "string *" C/C++ parameter) is designated as "object-owned," the object assumes ownership of the string, but not of the pointer to the string.)

object_owns_result
— Specifies that the object retains ownership of the return result of the method, and that the caller must not free the memory. The object is responsible for freeing the memory sometime before the object is destroyed. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense when the method's return type is a data type that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any). For methods that return an object, ownership applies to the object reference rather than to the object (that is, the object will be responsible for invoking **release** on the result, rather than **somFree**).

offset or **namelookup**   — Indicates whether the SOM Compiler should generate bindings for invoking the method using offset resolution or name lookup. **Offset** resolution requires that the **class** of the method's target object be known at compile time. When different methods of the same name are defined by several classes, **namelookup** is a more appropriate technique for method resolution than is offset resolution. (See Chapter 3, the section entitled "Invoking Methods.") The default modifier is **offset.**

override   — Indicates that the method is one introduced by an ancestor class and that this class will re-implement the method. See also the related modifier, **select**.

persistent   — Indicates a persistent attribute of a persistent object. (See Chapter 8, "Persistence Framework," for a discussion of persistent objects.)

procedure   — See "**method** or **nonstatic** or **procedure**."

reintroduce   — Indicates that this interface will "hide" a method introduced by some ancestor interface, and will replace it with another implementation. Methods introduced as direct-call procedures or nonstatic methods can be reintroduced. However, static methods (the default implementation category for SOM methods) cannot be reintroduced.

releaseorder: *a, b, c, ...*   — Specifies the order in which the SOM Compiler will place the class's methods in the data structures it builds to represent the class. Maintaining a consistent release order for a class allows the implementation of a class to change without requiring client programs to be recompiled.

The release order should contain every method name introduced by the class (private and nonprivate), but should not include any inherited methods, even if they are overridden by the class. The "get" and "set" methods defined automatically for each new attribute (named "_get_<attributeName>" and "_set_<attributeName>") should also be included in the release order list. The order of the names on the list is unimportant except that once a name is on the list and the class has client programs, it should not be reordered or removed, even if the method is no longer supported by the class, or the client programs will require recompilation. New methods should be added only to the end of the list. If a method named on the list is to be moved up in the class hierarchy, its name should remain on the current list, but it should also be added to the release order list for the class that will now introduce it.

If not explicitly specified, the release order will be determined by the SOM Compiler, and a warning will be issued for each missing method. If new methods or attributes are subsequently added to the class, the default release order might change; programs using the class would then require recompilation. Thus, it is advisable to explicitly give a release order.

select = *parent*	— Used in conjunction with the **override** modifier, this indicates that an inherited static method will use the implementation inherited from the indicated *<parent>* class. The parent is specified using the C-scoped name. For example, "Module_InterfaceName", not "Module::InterfaceName". See the later topic "Name usage in client programs" for an explanation of C-scoped names.
staticdata	— Indicates that the declared variable is not stored within objects, but, instead, that the ClassData structure for the implementing class will contain a pointer to the staticdata variable. This is similar in concept to C++ static data members. The staticdata variable must also be included in the **releaseorder**. The class implementor has responsibility for loading the ClassData pointer during class initialization. This can be facilitated by writing a special class initialization function and indicating its name using the **classinit** unqualified modifier. *Note:* attributes can be declared as staticdata. This is an important implementation technique that allows classes to introduce attributes whose backing storage is not inherited by subclasses.

The following example illustrates the specification of qualified modifiers:

```
implementation
{
  releaseorder : op1, op3, op2, op5, op6, x, y;
  op1 : persistent;
  somDefaultInit : override, init;
  op2: reintroduce, procedure;
  op3: reintroduce, nonstatic;
  op4: override, select = ModuleName_parentInterfaceName;
  op5: migrate = ModuleName_ancestorInterfaceName;
  op6: procedure, noself, noenv;
  long x;
  x: staticdata;
  y: staticdata; // y is an attribute
  mymod : a, b;
};
```

Passthru statements

A **passthru** statement (used within the body of an **implementation** statement, described above) allows a class implementor to specify blocks of code (for C/C++ programmers, usually only **#include** directives) that the SOM compiler will pass into the header files it generates.

Passthru statements are included in SOM IDL primarily for backward compatibility with the SOM OIDL language, and their use by C and C++ programmers should be limited to **#include** directives. C and C++ programmers should use IDL **type** and **constant** declarations rather than **passthru** statements when possible. (Users of other languages, however, may require **passthru** statements for type and constant declarations.)

The contents of the **passthru** lines are ignored by the SOM compiler and can contain anything that needs to placed near the beginning of a header file for a class. Even comments contained in **passthru** lines are processed without modification. The syntax for specifying **passthru** lines is one of the following forms:

passthru *language_suffix*	= *literal+* ;
passthru *language_suffix*_**before**	= *literal+* ;
passthru *language_suffix*_**after**	= *literal+* ;

where "language" specifies the programming language and "suffix" indicates which header files will be affected. The SOM Compiler supports suffixes **h, ih, xh,** and **xih.** For both C and C++, "language" is specified as C.

Each "literal" is a string literal (enclosed in double quotes) to be placed verbatim into the specified header file. [Double quotes within the passthru literal should be preceded by a backslash. No other characters escaped with a backslash will be interpreted, and formatting characters (newlines, tab characters, etc.) are passed through without processing.] The last literal for a passthru statement must not end in a backslash (put a space or other character between a final backslash and the closing double quote).

When either of the first two forms is used, **passthru** lines are placed **before** the **#include** statements in the header file. When the third form is used, **passthru** lines are placed just **after** the **#include** statements in the header file.

For example, the following **passthru** statement

```
implementation
{
  passthru C_h = "#include <foo.h>";
};
```

results in the directive `#include <foo.h>` being placed at the beginning of the .h C binding file that the SOM Compiler generates.

Declaring instance variables and staticdata variables

Declarators are used within the body of an **implementation** statement (described above) to specify the *instance variables* that are introduced by a class, and the *staticdata variables* pointed to by the class's **ClassData** structure. These variables are declared using ANSI C syntax for variable declarations, restricted to valid SOM IDL types (see "Type and constant declarations," above). For example, the following **implementation** statement declares two instance variables, x and y, and a staticdata variable, z, for class "Hello," :

```
implementation
{
  short x;
  long y;
  double z;
  z: staticdata;
};
```

Instance variables are normally intended to be accessed only by the class's methods and *not* by client programs or subclasses' methods. For data to be accessed by client programs or subclass methods, *attributes* should be used instead of instance variables. (Note, however, that declaring an attribute has the effect of also declaring an instance variable of the same name, unless the "nodata" attribute modifier is specified.)

Staticdata variables, by contrast, are publicly available and are associated specifically with their introducing class. They are, however, very different in concept from class variables. *Class variables* are really instance variables introduced by a metaclass, and are therefore present in any class that is an instance of the introducing metaclass (or of any metaclass derived from this metaclass). As a result, class variables present in any given class will also be present in any class derived from this class (that is, class variables are inherited). In contrast, staticdata variables are introduced by a class (not a metaclass) and are (only) accessed from the class's **ClassData** structure — they are *not* inherited.

To declare an instance variable that is *not* a valid IDL type, a dummy typedef can be declared before the **interface** declaration and a **passthru** statement then used to pass the real typedef to the language-specific binding file(s). See also the section "Using the tk_foreign TypeCode" in Chapter 7, "The Interface Repository Framework." In the following example, the generic SOM type **somToken** is used in the IDL file for the user's type "myRealType". The **passthru**

statement then causes an appropriate #include statement to be emitted into the C/C++ binding file, so that the file defining type "MyRealType" will be included when the binding file processes:

```
typedef somToken myRealType;

interface myClass : SOMObject {
. . .
  implementation {
     myRealType myInstVar;
     passthru C_h = "#include <myTypes.h>";
  };
};
```

Comments within a SOM IDL file

SOM IDL supports both C and C++ comment styles. The characters "//" start a line comment, which finishes at the end of the current line. The characters "/*" start a block comment that finishes with the "*/". Block comments do not nest. The two comment styles can be used interchangeably.

Comments in a SOM IDL specification must be strictly associated with particular syntactic elements, so that the SOM Compiler can put them at the appropriate place in the header and implementation files it generates. Therefore, comments may appear only in these locations (in general, following the syntactic unit being commented):

- At the beginning of the IDL specification
- After a semicolon
- Before or after the opening brace of a module, interface statement, implementation statement, structure definition, or union definition
- After a comma that separates parameter declarations or enumeration members
- After the last parameter in a prototype (before the closing parenthesis)
- After the last enumeration name in an enumeration definition (before the closing brace)
- After the colon following a case label of a union definition
- After the closing brace of an interface statement

Numerous examples of the use of comments can be found in the Tutorial of Chapter 2.

Because comments appearing in a SOM IDL specification are transferred to the files that the SOM Compiler generates, and because these files are often used as input to a programming language compiler, it is best within the body of comments to avoid using characters that are not generally allowed in comments of most programming languages. For example, the C language does not allow "*/" to occur within a comment, so its use is to be avoided, even when using C++ style comments in the .idl file.

SOM IDL also supports throw-away comments. They may appear anywhere in an IDL specification, because they are ignored by the SOM Compiler and are not transferred to any file it generates. Throw-away comments start with the string "//#" and end at the end of the line. Throw-away comments can be used to "comment out" portions of an IDL specification.

To disable comment processing (that is, to prevent the SOM Compiler from transferring comments from the IDL specification to the binding files it generates), use the **–c** option of the **sc** or **somc** command when running the SOM Compiler (See Section 4.3, "The SOM Compiler"). When comment processing is disabled, comment placement is not restricted, and comments can appear anywhere in the IDL specification.

Designating 'private' methods and attributes

To designate methods or attributes within an IDL specification as "private," the declaration of the method or attribute must be surrounded with the preprocessor commands **#ifdef __PRIVATE__** (with two leading underscores and two following underscores) and **#endif**. For example, to declare a method "foo" as a private method, the following declaration would appear within the interface statement:

```
#ifdef __PRIVATE__
void foo();
#endif
```

Any number of methods and attributes can be designated as private, either within a single **#ifdef** or in separate ones. [Kernighan–Ritchie reference for the C preprocessor: pages 88-92.]

When compiling a .idl file, the SOM Compiler normally recognizes only public (nonprivate) methods and attributes, as that is generally all that is needed. To generate header files for client programs that do need to access private methods and attributes, the **–p** option should be included when running the SOM Compiler. The resulting .h or .xh header file will then include bindings for private, as well as public, methods and attributes. The **–p** option is described in the topic "Running the SOM Compiler" later in this chapter.

The SOMobjects Toolkit also provides a **pdl** (Public Definition Language) emitter that can be used with the SOM Compiler to generate a copy of a .idl file which has the portions designated as private removed. The next main section of this chapter describes how to invoke the SOM Compiler and the various emitters.

Defining multiple interfaces in a .idl file

A single .idl file can define *multiple interfaces*. This allows, for example, a class and its metaclass to be defined in the same file. When a file defines two (or more) interfaces that reference one another, forward declarations can be used to declare the name of an interface before it is defined. This is done as follows:

> **interface** *class–name* **;**

The actual definition of the **interface** for "class-name" must appear later in the same .idl file.

If multiple interfaces are defined in the same .idl file, and the classes are not a class–metaclass pair, they can be grouped into modules, by using the following syntax:

> **module** *module-name* **{** *definition*+ **};**

where each "definition" is a **type** declaration, **constant** declaration, **exception** declaration, **interface** statement, or nested **module** statement. Modules are used to scope identifiers (see below).

Alternatively, multiple interfaces can be defined in a single .idl file without using a module to group the interfaces. Whether or not a module is used for grouping multiple interfaces, the languages bindings produced from the .idl file will include support for all of the defined interfaces.

Note: When multiple interfaces are defined in a single .idl file and a **module** statement is not used for grouping these interfaces, it is necessary to use the **functionprefix** modifier to assure that different names exist for functions that provide different implementations for a method. In general, it is a good idea to always use the **functionprefix** modifier, but in this case it is essential.

Scoping and name resolution

A .idl file forms a *naming scope* (or *scope*). **Modules**, **interface** statements, **structures**, **unions**, **methods**, and **exceptions** form *nested scopes*. An identifier can only be defined once in a particular scope. Identifiers can be redefined in nested scopes.

Names can be used in an unqualified form within a scope, and the name will be resolved by successively searching the enclosing scopes. Once an unqualified name is defined in an enclosing scope, that name cannot be redefined.

Fully qualified names are of the form:

> *scoped–name**::**identifier*

For example, method name "meth" defined within interface "Test" of module "M1" would have the fully qualified name:

```
M1::Test::meth
```

A qualified name is resolved by first resolving the "scoped-name" to a particular scope, S, and then locating the definition of "identifier" within that scope. Enclosing scopes of S are not searched.

Qualified names can also take the form:

::identifier

These names are resolved by locating the definition of "identifier" within the smallest enclosing module.

Every name defined in an IDL specification is given a global name, constructed as follows:

- Before the SOM Compiler scans a .idl file, the name of the current *root* and the name of the current *scope* are empty. As each module is encountered, the string "::" and the module name are appended to the name of the current root. At the end of the module, they are removed.

- As each interface, struct, union, or exception definition is encountered, the string "::" and the associated name are appended to the name of the current scope. At the end of the definition, they are removed. While parameters of a method declaration are processed, a new un-named scope is entered so that parameter names can duplicate other identifiers.

- The global name of an IDL definition is then the concatenation of the current root, the current scope, a "::", and the local name for the definition.

The names of types, constants, and exceptions defined by the parents of a class are accessible in the child class. References to these names must be unambiguous. Ambiguities can be resolved by using a scoped name (prefacing the name with the name of the class that defines it and the characters "::", as in "parent-class::identifier"). Scope names can also be used to refer to a constant, type, or exception name defined by a parent class but redefined by the child class.

Name usage in client programs

Within a C or C++ program, the global name for a **type, constant,** or **exception** corresponding to an IDL scoped name is derived by converting the string "::" to an underscore ("_") and removing the leading underscore. Such names are referred to as *C–scoped names*. This means that types, constants, and exceptions defined within the interface statement for a class can be referenced in a C/C++ program by prepending the class name to the name of the type, constant, or exception. For example, consider the types defined in the following IDL specification:

```
typedef sequence<long,10> mySeq;
interface myClass : SOMObject
{
    enum color {red, white, blue};
    typedef string<100> longString;
    ...
}
```

These types could be accessed within a C or C++ program with the following global names:

```
mySeq,
myClass_color,
myClass_red,
myClass_white,
myClass_blue, and
myClass_longString
```

Type, constant, and exception names defined within modules similarly have the module name prepended. When using SOM's C/C++ bindings, the short form of type, constant, and exception names (such as, `color, longString`) can also be used where unambiguous, except that enumeration names must be referred to using the long form (for example: `myClass_red` and not simply `red`).

Because replacing "::" with an underscore to create global names can lead to ambiguity if an IDL identifier contains underscores, it is best to avoid the use of underscores when defining IDL identifiers.

Extensions to CORBA IDL permitted by SOM IDL

The following topics describe several SOM-unique extensions of the standard CORBA syntax that are permitted by SOM IDL for convenience. These constructs can be used in a .idl file without generating a SOM Compiler error.

If you want to verify that an IDL file contains only standard CORBA specifications, the SOM Compiler option **–mcorba** turns off each of these extensions and produces compiler errors wherever non-CORBA specifications are used. (The SOM Compiler command and options are described in the topic "Running the SOM Compiler" later in this chapter.)

Pointer '*' types

In addition to the base CORBA types, SOM IDL permits the use of pointer types ('*'). As well as increasing the range of base types available to the SOM IDL programmer, using pointer types also permits the construction of more complex data types, including self-referential and mutually recursive structures and unions.

If self-referential structures and unions are required, then, instead of using the CORBA approach for IDL sequences, such as the following:

```
struct X {
    ...
    sequence <X> self;
    ...
};
```

it is possible to use the more typical C/C++ approach. For example:

```
struct X {
    ...
    X *self;
    ...
};
```

SOM IDL does not permit an explicit '*' in sequence declarations. If a sequence is required for a pointer type, then it is necessary to typedef the pointer type before use. For example:

```
sequence <long *> long_star_seq;          // error.

typedef long * long_star;
sequence <long_star> long_star_seq;       // OK.
```

Unsigned types

SOM IDL permits the syntax "unsigned *<type>*", where *<type>* is a previously declared type mapping onto "short" or "long". (Note that CORBA permits only "unsigned short" and "unsigned long".)

Implementation section

SOM IDL permits an **implementation** section in an IDL **interface** specification to allow the addition of instance variables, method overrides, metaclass information, passthru information, and "pragma–like" information, called **modifiers**, for the emitters. See the topic "Implementation statements" earlier in this chapter.

Comment processing

The SOM IDL Compiler by default does not remove comments in the input source; instead, it attaches them to the nearest preceding IDL statement. This facility is useful, since it allows comments to be emitted in header files, C template files, documentation files, and so forth. However, if this capability is desired, this does mean that comments cannot be placed with quite as much freedom as with an ordinary IDL compiler. To turn off comment processing so that you can compile .idl files containing comments placed anywhere, you can use the compiler option **–c** or use "throw-away" comments throughout the .idl file (that is, comments preceded by //#); as a result, no comments will be included in the output files.

Generated header files

CORBA expects one header file, *<file>*.h, to be generated from *<file>*.idl. However, SOM IDL permits use of a class modifier, **filestem,** that changes this default output file name. (See "Running the SOM Compiler" later in this chapter.)

4.3 The SOM Compiler

The SOM Compiler translates the IDL definition of a SOM class into a set of "binding files" appropriate for the language that will implement the class's methods and the language(s) that will use the class. These bindings make it more convenient for programmers to implement and use SOM classes. The SOM Compiler currently produces binding files for the C and C++ languages.

Important Note: C and C++ bindings *cannot* both be generated during the same execution of the SOM compiler.

Generating binding files

The SOM Compiler operates in two phases:

- A precompile phase, in which a precompiler analyzes an OIDL or IDL class definition, and

- An emission phase, in which one or more emitter programs produce binding files.

Each binding file is generated by a separate emitter program. Setting the SMEMIT environment variable determines which emitters will be used, as described below. Note: In the discussion below, the <filestem> is determined by default from the name of the source .idl file with the ".idl"extension removed. Otherwise, a "filestem" modifier can be defined in the .idl file to specify another file name (see "Modifier statements" above).

Note: When changes to definitions in the .idl file later become necessary, the SOM Compiler should be rerun to update the current implementation template file, provided that the **c** or **xc** emitter is specified (either with the **–s** option or the SMEMIT environment variable, as described below). For more information on generating updates, see "Running incremental updates of the implementation template file" in Chapter 5, "Implementing Classes in SOM."

The emitters for the C language produce the following binding files:

<filestem>.**c**	— (produced by the **c** emitter) This is a template for a C source program that implements a class's methods. This will become the primary source file for the class. (The other binding files can be generated from the **.idl** file as needed.) This template implementation file contains "stub" procedures for each method introduced or overridden by the class. (The stub procedures are empty of code except for required initialization and debugging statements.)

After the class implementor has supplied the code for the method procedures, running the **c** emitter again will update the implementation file to reflect changes made to the class definition (in the **.idl** file). These updates include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. Existing code within method procedures is *not* disturbed, however.

The **.c** file contains an **#include** directive for the **.ih** file, described below.

The contents of the C source template is controlled by the Emitter Framework file <SOMBASE>/include/ctm.efw. This file can be customized to change the template produced. For detailed information on changing the template file see the *Emitter Framework Guide and Reference*.

<filestem>.h   — (produced by the **h** emitter)

This is the <u>header file to be included by C client programs</u> (programs that use the class). It contains the C usage bindings for the class, including macros for accessing the class's methods and a macro for creating new instances of the class. This header file includes the header files for the class's parent classes and its metaclass, as well as the header file that defines SOM's generic C bindings, **som.h.**

<filestem>.ih   — (produced by the **ih** emitter)

This is the <u>header file to be included in the implementation file</u> (the file that implements the class's methods — the .c file). It contains the implementation bindings for the class, including:

- a **struct** defining the class's instance variables,
- macros for accessing instance variables,
- macros for invoking parent methods the class overrides,
- the *<className>***GetData** macro used by the method procedures in the *<filestem>***.c** file (see "Stub procedures for methods" in Section 5.4 of Chapter 5),
- a *<className>***NewClass** procedure for constructing the class object at run time, and
- any IDL types and constants defined in the IDL interface.

<u>**The emitters for the C++ language**</u> produce the following binding files:

<filestem>.**C** (for AIX) or *<filestem>*.**cpp** (for OS/2 or Windows)

       — (produced by the **xc** emitter)

This is a <u>template for a C++ source program</u> that implements a class's methods. This will become the primary source file for the class. (The other binding files can be generated from the **.idl** file as needed.) This template implementation file contains "stub" procedures for each method introduced or overridden by the class. (The stub procedures are empty of code except for required initialization and debugging statements.)

After the class implementor has supplied the code for the method procedures, running the **xc** emitter again will update this file to reflect changes made to the class definition (in the **.idl** file). These updates include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. Existing code within method procedures is *not* disturbed, however.

The C++ implementation file contains an **#include** directive for the **.xih** file, described below.

The contents of the C++ source template is controlled by the Emitter Framework file <SOMBASE>/include/ctm.efw. This file can be customized to change the template produced. For detailed information on changing the template file see the *Emitter Framework Guide and Reference*.

<filestem>.**xh**	— (produced by the **xh** emitter) This is the <u>header file to be included by C++ client programs</u> that use the class. It contains the usage bindings for the class, including a C++ definition of the class, macros for accessing the class's methods, and the **new** operator for creating new instances of the class. This header file includes the header files for the class's parent classes and its metaclass, as well as the header file that defines SOM's generic C++ bindings, **som.xh**.
<filestem>.**xih**	— (produced by the **xih** emitter) This is the <u>header file to be included in the implementation file</u> (the file that implements the class's methods). It contains the implementation bindings for the class, including:

- a **struct** defining the class's instance variables,
- macros for accessing instance variables,
- macros for invoking parent methods the class overrides,
- the *<className>***GetData** macro (see "Stub procedures for methods" in Section 5.4 of Chapter 5),
- a *<className>***NewClass** procedure for constructing the class object at run time, and
- any IDL types and constants defined in the IDL interface.

Other files the SOM Compiler generates:

<filestem>.**hh**	— (produced by the **hh** emitter) This file is <u>a DirectToSOM C++ header file</u> that describes a SOMobjects class in a way appropriate for DTS C++. When running this emitter, you must include the **noqualifytypes** command-line modifier for the **–m** option of the SOM Compiler command **sc** or **somc**.
<filestem>.**pdl**	— (produced by the **pdl** emitter) This file is the same as the .idl file from which it is produced except that all items within the .idl file that are marked as "private" have been removed. (An item is marked as private by surrounding it with "#ifdef __PRIVATE__" and "#endif" directives.) Thus, the **pdl** (Public Definition Language) emitter can be <u>used to generate a "public" version of the .idl file</u>.
<filestem>.**def** (for OS/2 or Windows)	— (produced by the **def** emitter) This file is <u>used by the linker to package a class as a library</u>. To combine several classes into a single library, you must merge the **exports** statements from each of their **.def** files into a single **.def** file for the entire library. When packaging multiple classes in a single library, you must also write a simple C procedure named **SOMInitModule** and add it to the export list. This procedure should call the routine *<className>***NewClass** for each class packaged in the library. The **SOMInitModule** procedure is called by the SOM Class Manager when the library is dynamically loaded.

<filestem>.**exp** (for AIX) — (produced by the **exp** emitter)
This file is <u>used by the linker to package a class as a library</u>. To combine several classes into a single library, you must merge the **exports** statements from each of their **.exp** files into a single **.exp** file for the entire library. When packaging multiple classes in a single library, you must also write a simple C procedure named **SOMInitModule** and add it to the export list. This procedure should call the routine *<className>***NewClass** for each class packaged in the library. The **SOMInitModule** procedure is called by the SOM Class Manager when the library is dynamically loaded.

The **Interface Repository**   (produced by the **ir** emitter)
See Chapter 7 for a discussion on the Interface Repository.

Note: The C/C++ bindings generated by the SOM Compiler have the following limitation: If two classes named "ClassName" and "ClassNameC" are defined, the bindings for these two classes will clash. That is, if a client program uses the C/C++ bindings (includes the .h/.xh header file) for both classes, a name conflict will occur. Thus, class implementors should keep this limitation in mind when naming their classes.

SOM users can extend the SOM Compiler to generate additional files by writing their own emitters. To assist users in extending the SOM Compiler, SOM provides an Emitter Framework — a collection of classes and methods useful for writing object-oriented emitters that the SOM Compiler can invoke. For more information, see the *Emitter Framework Guide and Reference*.

Note re: porting SOM classes: The header files (binding files) that the SOM Compiler generates will only work on the platform (operating system) on which they were generated. Thus, when porting SOM classes from the platform where they were developed to another platform, the header files must be regenerated from the .idl file by the SOM Compiler on that target platform.

Environment variables affecting the SOM Compiler

To execute the SOM Compiler on one or more files that contain IDL specifications for one or more classes, use the **sc** or **somc** command, as follows:

sc [*–options*] *files*   (on AIX or OS/2)
somc [*–options*] *files*   (on Windows)

where "files" specifies one or more .idl files.

Available "–options" for the command are detailed in the next topic. The operation of the SOM Compiler (whether it produces C binding files or C++ binding files, for example) is also controlled by certain environment variables that can be set before the **sc** or **somc** command is issued. The applicable environment variables are as follows:

SMEMIT — <u>Determines which output files</u> the SOM Compiler produces. Its value consists of a list of items separated by semicolons for OS/2, or by semicolons or colons for AIX. <u>Each item designates an emitter</u> to execute. For example, the statement:

```
SET SMEMIT=c;h;ih            (for OS/2)
export SMEMIT="c;h;ih"       (for AIX)
```

directs the SOM Compiler to produce the <u>C binding files</u> "hello.c", "hello.h", and"hello.ih" from the "hello.idl" input specification. By comparison,

```
SET SMEMIT=xc;xh;xih         (for OS/2)
export SMEMIT="xc;xh;xih"    (for AIX)
```

directs the SOM Compiler to produce <u>C++ binding files</u> "hello.C" (for AIX) or "hello.cpp" (for OS/2), "hello.xh", and "hello.xih" from the "hello.idl" input specification.

By default, all output files are placed in the same directory as the input file. If the SMEMIT environment variable is not set, then a default value of "h;ih" is assumed.

Windows note: The SMEMIT environmental variable can be set by using the SET command before the **somc** command is issued. For example:

```
SET SMEMIT="c;h;ih"              (for Windows)
```

If you are running the SOM Compiler from a DOS box under Windows, make sure to define SMEMIT *before* Windows is started.

SMINCLUDE — Specifies where the SOM Compiler should look for .idl files #included by the .idl file being compiled. Its value should be one or more directory names separated by a semicolon when using OS/2 or Windows, or separated by a semicolon or colon when using AIX. Directory names can be specified with absolute or relative pathnames. For example:

```
SET SMINCLUDE=.;..\MYSCDIR;C:\TOOLKT20\C\INCLUDE;
                                    (for OS/2 or Windows)
```

```
  export SMINCLUDE=.:myscdir:/u/som/include
                                    (for AIX)
```

The default value of the SMINCLUDE environment variable is the "include" subdirectory of the directory into which SOM has been installed.

SMTMP — Specifies the directory that the SOM Compiler should use to hold intermediate output files. This directory should <u>*not* coincide</u> with the directory of the input or output files. For AIX, the default setting of SMTMP is /tmp; for OS/2 or Windows, the default setting of SMTMP is the root directory of the current drive.

OS/2 or Windows example:

```
SET SMTMP=..\MYSCDIR\GARBAGE
```

tells the SOM Compiler to place the temporary files in the GARBAGE directory.

Or, on OS/2 only:

```
SET SMTMP=%TMP%
```

tells the SOM Compiler to use the same directory for temporary files as given by the setting of the TMP environment variable (the default location for temporary system files). (On Windows, you cannot set one variable to another.)

AIX example:

```
export SMTMP=$TMP
export SMTMP=../myscdir/garbage
```

SMKNOWNEXTS — Specifies additional emitters to which the SOM Compiler should add a header. For example, if you were to write a new emitter for Pascal, called "emitpas", then by default the SOM Compiler would not add any header comments to it. However, by setting SMKNOWNEXTS=pas, as shown:

```
set SMKNOWNEXTS=pas          (for OS/2 or
                              Windows)
export SMKNOWNEXTS=pas        (for AIX)
```

the SOM Compiler will add a header to files generated with the "emitpas" emitter. The "header" added is a SOM Compiler-generated message plus any comments, such as copyright statements, that appear at the head of your .idl input file. For details on writing your own emitter, see the *Emitter Framework Guide and Reference*.

SOMIR — Specifies the name (or list of names) of the Interface Repository file. The **ir** emitter, if run, creates the Interface Repository, or checks it for consistency if it already exists. If the **–u** option is specified when invoking the SOM Compiler, the **ir** emitter also updates an existing Interface Repository.

Running the SOM Compiler

The syntax of the command for running the SOM Compiler takes the forms:

sc [–*options*] *files*   (on AIX or OS/2)
somc [–*options*] *files*   (on Windows)

The "files" specified in the **sc** or **somc** command denote one or more files containing the IDL class definitions to be compiled. If no extension is specified, .idl is assumed. By default, the <filestem> of the .idl file determines the filestem of each emitted file. Otherwise, a "filestem" modifier can be defined in the .idl file to specify another name (see "Modifier statements" discussed earlier).

Selected "–options*"* can be specified individually, as a string of option characters, or as a combination of both. Any option that takes an argument either must be specified individually or must appear as the final option in a string of option characters. Available options and their purposes are as follows:

–C *n*   Sets the maximum allowable size for a simple comment in the .idl file (default: 32767). This is only needed for very large single comments.

–D *name*[*=def*]
          Defines *name* as in a #define directive. The default *def* is 1. This option is the same as the –D option for the C compiler. Note: This option can be used to define __**PRIVATE**__ so that the SOM Compiler will also compile any methods and attributes that have been defined as private using the directive **#ifdef** __**PRIVATE**__; however, the –p option does the same thing more easily.

–E *variable=value*
          Sets an environment variable. (See the previous topic for a discussion of the available environment variables: SMEMIT, SMINCLUDE, SMTMP, and SMNOIR.)

–I *dir*   When looking for #included files, looks first in dir, then in the standard directories (same as the C compiler **–I** option).

–S *n*   Sets the total allowable amount of unique string space used in the IDL specification for names and passthru lines (default: 32767). This is only needed for very large .idl files.

–U *name*   Removes any initial definition (via a #define preprocessor directive) of symbol *name*.

–V	Displays version information about the SOM Compiler.
–c	Turns off comment processing. This allows comments to appear anywhere within an IDL specification (rather than in restricted places), and it causes comments not to be transferred to the output files that the SOM Compiler produces.
–d *directory*	Specifies a directory where all output files should be placed. If the –d option is not used, all output files are placed in the same directory as the input file.
–h or **–?**	Produces a listing of this option list. (This option is typically used in an **sc** or **somc** command that does not include a .idl file name).
–i *filename*	Specifies the name of the class definition file. Use this option to override the built-in assumption that the input file will have a .idl extension. Any *filename* supplied with the –i option is used exactly as it is specified.

–m *name*[*=value*]

Adds a global modifier. The currently supported global modifiers are as follows:

addprefixes

Adds 'functionprefixes' to the method procedure prototypes during an incremental update of the implementation template file. This option applies only when rerunning the **c** or **xc** emitter on an IDL file that previously did *not* specify a functionprefix. A class implementor who later decides to use prefixes should add a line in the 'implementation' section of the .idl file containing the specification:

functionprefix = *prefix*

(as described earlier in the topic "Modifier statements") and then rerun the c or xc emitter using the **–maddprefixes** option. The method procedure prototypes in the implementation file will then be updated so that each method name includes the assigned prefix. (This option does not support changes to existing prefix names, nor does it apply for OIDL files.)

addstar This option causes all interface references to have a '*****' added to them for the C bindings. See the earlier section entitled "Object types" for further details.

comment=*comment string*

where *comment string* can be either of the designations: "/*" or "//". This option indicates that comments marked in the designated manner in the .idl file are to be completely ignored by the SOM Compiler and will *not* be included in the output files. <u>Note</u>: Comments on lines beginning with "//#" are always ignored by the SOM Compiler.

corba This option directs the SOM Compiler to compile the input definition according to strict CORBA-defined IDL syntax. This means, for example, that comments may appear anywhere and that pointers are not allowed. When the **–mcorba** option is used, parts of a .idl file surrounded by **#ifdef __SOMIDL__** and **#endif** directives are ignored. This option can be used to determine whether all nonstandard constructs (those specific to SOM IDL) are properly protected by **#ifdef __SOMIDL__** and **#endif** directives.

–m *name*[*=value*]   (modifier options continued from the previous page)

csc This option forces the OIDL compiler to be run. This is required only if you want to compile an OIDL file that does not have an extension of .csc or .sc.

emitappend

This option causes emitted files to be appended at the end of existing files of the same name.

noheader

This option ensures that the SOM Compiler does not add a header to the beginning of an emitted file.

noint   This option directs the SOM Compiler not to warn about the portability problems of using `int`'s in the source.

nolock   This option causes the Interface Repository Emitter **emitir** (see Chapter 7, "Interface Repository Framework") to leave the IR unlocked when updates are made to it. This can improve performance on networked file systems. By not locking the IR, however, there is the risk of multiple processes attempting to write to the same IR, with unpredictable results. This option should only be used when you know that only one process is updating an IR at once.

nopp   This option directs the SOM Compiler not to run the SOM preprocessor on the .idl input file.

noqualifytypes
             This option prevents the use of C-scoped names in emitter output, and is used in conjunction with the .hh emitter.

notc   This option directs the SOM Compiler not to create TypeCode information when emitting files. This is required only when the .idl files contain some undeclared types. This option is typically used when compiling converted .csc files that have not had typing information added.

nouseshort
             This option directs the SOM Compiler not to generate short forms for type names in the .h and .xh public header files. This can be useful to save disk space.

pp=*preprocessor*
             This option directs the SOM Compiler to use the specified preprocessor as the SOM preprocessor, rather than the default "somcpp". Any standard C/C++ preprocessor can be used as a preprocessor for IDL specifications.

tcconsts This option directs the SOM Compiler to generate TypeCode constants in the .h and .xh public header files. Please refer to the Interface Repository (described in Chapter 7) for more details.

Note: All command-line **–m** modifier options can be specified in the environment by changing them to UPPERCASE and prepending "SM" to them. For example, if you want to always set the options "–mnotc" and "–maddstar", set corresponding environment variables as follows:

> On OS/2 or Windows:
> ```
> set SMNOTC=1
> set SMADDSTAR=1
> ```
>
> On AIX:
> ```
> export SMNOTC=1
> export SMADDSTAR=1
> ```

–p   Causes the "private" sections of the IDL file to be included in the compilation (that is, sections preceded by #ifdef __PRIVATE__ that contain private methods and attributes). Note: The –p option is equivalent to the earlier option –D__PRIVATE__.

–r   Checks that all names specified in the release order statement are valid method names (default: FALSE).

–s *string* Substitutes *string* in place of the contents of the **SMEMIT** environment variable for the duration of the current **sc** or **somc** command. This determines which emitters will be run and, hence, which output files will be produced. (If a list of values is given, on OS/2 *only* the list must be enclosed in double quotes.)

The –s option is a convenient way to override the SMEMIT environment variable. In OS/2 for example, the command:

```
> SC -s"h;c" EXAMPLE
```

is equivalent to the following sequence of commands:

```
> SET OLDSMEMIT=%SMEMIT%
> SET SMEMIT=H;C
> SC EXAMPLE
> SET SMEMIT=%OLDSMEMIT%
```

Similarly, in AIX the command:

```
> sc -sh";"c example
```

is equivalent to the following sequence of commands:

```
> export OLDSMEMIT=$SMEMIT
> export SMEMIT=h";"c
> sc example
> export SMEMIT=$OLDSMEMIT
```

Windows users: You should *not* use quotes in substitution strings of the **–s** option for the **somc** command:

```
> somc -sh;ih *.idl        Correct
> somc -s"h;ih" *.idl      Incorrect
```

–u   Updates the Interface Repository (default: no update). With this option, the Interface Repository will be updated even if the **ir** emitter is not explicitly requested in the SMEMIT environment variable or the **–s** option.

–v   Uses verbose mode to display informational messages (default: FALSE). This option is primarily intended for debugging purposes and for writers of emitters.

–w   Suppresses warning messages (default: FALSE).

The following sample commands illustrate various options for the **sc** command (or similarly with **somc**):

```
sc -sc hello.idl              Generates file "hello.c".
sc -hV                        Generates a help message and displays the
                                 version of the SOM Compiler currently available.
sc -vsh";"ih hello.idl        Generates "hello.h" and "hello.ih" with
                                 informational messages. Note: On Windows,
                                 omit the quotes (") when issuing this command.

sc -sxc -doutdir hello.idl
                              Generates "hello.xc" in directory "outdir".
```

4.4 The 'pdl' Facility

As discussed earlier in this chapter, the SOM Compiler provides a **pdl** (Public Definition Language) emitter. This emitter generates a file that is the same as the .idl file from which it is produced, except that it removes all items within the .idl file that are marked as "private." (An item is marked as private by surrounding it with "#ifdef _ _PRIVATE_ _" and "#endif" directives.) Thus, the **pdl** emitter can be used to generate a "public" version of a .idl file. (Generally, client programs will need only the "public" methods and attributes.)

The SOMobjects Toolkit also provides a separate program, **pdl**, which performs the same function, but can be invoked independently of the SOM Compiler. The **pdl** program is invoked as follows:

> **pdl** *files*

where "files" specifies one or more .idl files whose "PRIVATE" sections are to be removed. Filenames must be completely specified (with the .idl extension).

The **pdl** command supports the following options. (Selected *options* can be specified individually, as a string of option characters, or as a combination of both. Any option that takes an argument either must be specified individually or must appear as the final option in a string of option characters.)

–d *dir*
Specifies a directory in which the output files are to be placed. (The output files are given the same name as the input files.) If no directory is specified, the output files are named <*fileStem*>.pdl (where *fileStem* is the file stem of the input file) and are placed in the current working directory.

–f
Specifies that output files are to replace existing files with the same name, even if the existing files are read–only. By default, files are replaced only if they have write access.

–s *smemit*
Specifies that, for each specified .idl file, the **pdl** program is to invoke the SOM Compiler with *smemit* as the value of the –s option.

–c *cmd*
Specifies that, for each .idl file, the **pdl** program is to run the specified system command. This command may contain a single occurrence of the string "%s", which will be replaced with the source file name before the command is executed. For example the option –c"sc –sh %s" has the same effect as using the option –sh.

For example, to install public versions of the .idl files in the directory "pubinclude", type:

```
pdl –d pubinclude *.idl
```

Chapter 5. Implementing Classes in SOM

This chapter begins with a more in-depth discussion of SOM concepts and the SOM run-time environment than was appropriate in Chapter 2, "Tutorial for Implementing SOM Classes." Subsequent sections then provide information about completing an implementation template file, updating the template file, compiling and linking, packaging classes in libraries, and other useful topics for class implementors. During this process, you can refer to Chapter 4, "SOM IDL and the SOM Compiler," if you want to read the reference information or see the full syntax related to topics discussed in this chapter. The current chapter ends with topics describing how to customize SOMobjects execution in various ways.

5.1 The SOM Run-time Environment

As discussed in Chapter 1, the SOMobjects Developer Toolkit provides

- The **SOM Compiler**, used when creating SOM class libraries, and
- The **SOM run-time library**, for using SOM classes at execution time.

The SOM run-time library provides a set of *functions* used primarily for creating objects and invoking methods on them. The *data structures* and *objects* that are created, maintained, and used by the functions in the SOM run-time library constitute the **SOM run-time environment**.

A distinguishing characteristic of the SOM run-time environment is that <u>SOM *classes* are represented by run-time *objects*</u>; these objects are called *class objects.* By contrast, other object-oriented languages such as C++ treat classes strictly as compile-time structures that have no properties at run time. In SOM, however, each class has a corresponding run-time object. This has three advantages: First, application programs can access information about a class at run time, including its relationships with other classes, the methods it supports, the size of its instances, and so on. Second, because much of the information about a class is established at run time rather than at compile time, application programs needn't be recompiled when this information changes. Finally, because class objects can be instances of user-defined classes in SOM, users can adapt the techniques for subclassing and inheritance in order to build object-oriented solutions to problems that are otherwise not easily addressed within an OOP context.

Run-time environment initialization

When the SOM run-time environment is initialized, four primitive SOM objects are automatically created. Three of these are <u>class objects</u> (**SOMObject**, **SOMClass**, and **SOMClassMgr**), and one is an <u>instance</u> of **SOMClassMgr**, called the **SOMClassMgrObject**. Once loaded, application programs can invoke methods on these class objects to perform tasks such as creating other objects, printing the contents of an object, freeing objects, and the like. These four primitive objects are discussed below.

In addition to creating the four primitive SOM objects, initialization of the SOM run-time environment also involves <u>initializing global variables to hold data structures</u> that maintain the state of the environment. Other functions in the SOM run-time library rely on these global variables.

For application programs written in C or C++ that use the language-specific bindings provided by SOM, <u>the SOM run-time environment is automatically initialized the first time any object is created</u>. Programmers using other languages must initialize the run-time environment explicitly by calling the **somEnvironmentNew** function (provided by the SOM run-time library) before using any other SOM functions or methods.

SOMObject class object

SOMObject is the root class for all SOM *classes*. It defines the essential *behavior* common to all SOM *objects*. All user-defined SOM classes are derived, directly or indirectly, from this class. That is, every SOM class is a subclass of **SOMObject** or of some other class derived from **SOMObject**. **SOMObject** has no instance variables, thus objects that inherit from **SOMObject** incur no size increase. They do inherit a suite of methods that provide the behavior required of all SOM objects.

SOMClass class object

Because SOM classes are run-time objects, and since all run-time objects are instances of some class, it follows that a SOM class object must also be an instance of some class. The <u>class of a class is called a *metaclass*</u>. Hence, the instances of an ordinary class are individuals (nonclasses), while the instances of a metaclass are class objects.

In the same way that the class of an object defines the "instance methods" that the object can perform, the metaclass of a class defines the "class methods" that the class itself can perform. *Class methods* (sometimes called *factory methods* or *constructors*) are <u>performed by class objects</u>. Class methods perform tasks such as creating new instances of a class, maintaining a count of the number of instances of the class, and other operations of a "supervisory" nature. Also, class methods facilitate inheritance of instance methods from parent classes.

The distinction between instance methods and class methods, as well as that between objects, classes, and metaclasses, is illustrated in Figure 1. For information on the distinction between parent classes and metaclasses, see the section "Parent class vs. metaclass," below.

Class methods vs. instance methods

Metaclass "M"

Class "C"

Object "O$_1$" O$_2$ O$_3$ O$_4$ O$_5$

Class Methods defined in metaclass "M" are performed by class "C" to produce instances.

Instance Methods defined in class "C" are performed by O$_1$ ... O$_5$

("is an instance of")
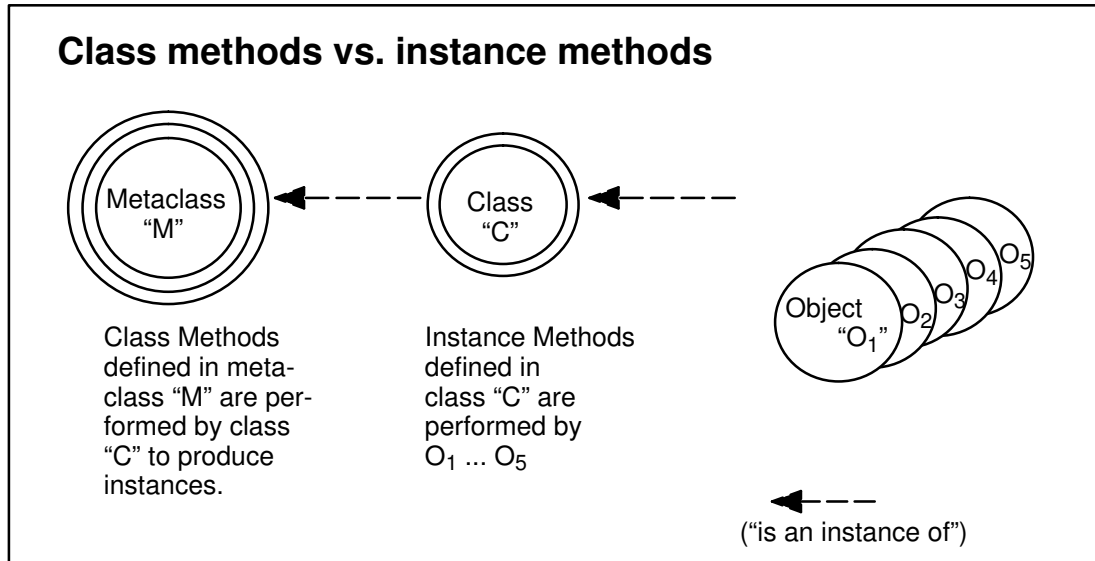
Figure 1.   Class methods vs. instance methods.

SOMClass is the root class for all SOM *metaclasses*. That is, all SOM metaclasses must be subclasses of **SOMClass** or of some metaclass derived from **SOMClass**. **SOMClass** defines the essential behavior common to all SOM *class objects*. In particular, **SOMClass** provides:

- Six class methods for creating new class instances: **somNew**, **somNewNoInit**, **somRenew**, **somRenewNoInit**, **somRenewNoZero**, and **somRenewNoInitNoZero**.

- A number of class methods that dynamically obtain or update information about a class and its methods at run time, including:
  - **somInitMIClass,** for implementing multiple inheritance from parent classes,
  - **somOverrideSMethod,** for overriding inherited methods, and
  - **somAddStaticMethod** and **somAddDynamicMethod**, for introducing new methods.

SOMClass is a subclass (or child) of **SOMObject**. Hence, SOM *class objects* can also perform the same set of basic *instance methods* common to all SOM objects. This is what allows SOM classes to be real objects in the SOM run-time environment. **SOMClass** also has the unique distinction of being its own metaclass (that is, **SOMClass** defines its own class methods).

A user-defined class can designate as its metaclass either **SOMClass** or another user-written metaclass descended from **SOMClass**. If a metaclass is not explicitly specified, SOM determines one automatically.

SOMClassMgr class object and SOMClassMgrObject

The third primitive SOM class is **SOMClassMgr**. A single instance of the **SOMClassMgr** class is created automatically during SOM initialization. This instance is referred to as the **SOMClassMgrObject**, because it is pointed to by the *global variable* SOMClassMgrObject**.** The object **SOMClassMgrObject** has the responsibility to

- Maintain a registry (a run-time directory) of all SOM classes that exist within the current process, and to

- Assist in the dynamic loading and unloading of class libraries.

For C/C++ application programs using the SOM C/C++ language bindings, the **SOMClassMgrObject** automatically loads the appropriate library file and constructs a run-time object for the class the first time an instance of a class is created. For programmers using other languages, **SOMClassMgr** provides a method, **somFindClass**, for directing the **SOMClassMgrObject** to load the library file for a class and create its class object.

Relationships among the four primitive SOM run-time objects are illustrated in Figure 2. Again, the primitive classes supplied with SOM are **SOMObject, SOMClass,** and **SOM-ClassMgr.** During SOM initialization, the latter class generates an instance called **SOM-ClassMgrObject.** The left-hand side of Figure 2 shows parent-class relationships between the built-in SOM classes, and the right-hand side shows instance/class relationships. That is, on the left **SOMObject** is the parent class of **SOMClass** and **SOMClassMgr**. On the right **SOM-Class** is the metaclass of itself, of **SOMObject,** and of **SOMClassMgr**, which are all class objects at run time. **SOMClassMgr** is the class of **SOMClassMgrObject**.
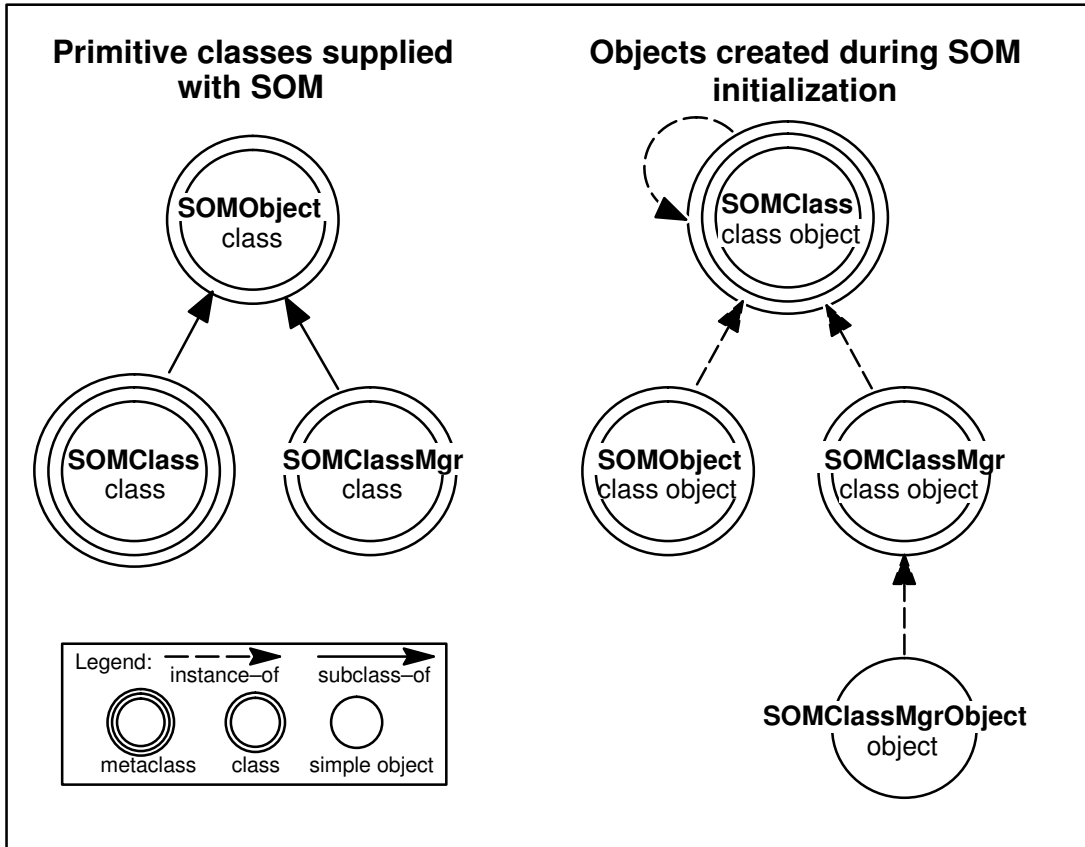

Figure 2. The SOM run-time environment provides four primitive objects, three of them class objects.

Parent class vs. metaclass

There is a distinct difference between the notions of "parent" (or base) class and "metaclass." Both notions are related to the fact that a class defines the methods and variables of its instances, which are therefore called *instance methods* and *instance variables*.

A *parent* of a given class is a class from which the given class is *derived* by *subclassing*. (Thus, the given class is called a *child* or a *subclass* of the parent.) A parent class is a class from which instance methods and instance variables are *inherited*. For example, the parent of class "Dog" might be class "Animal". Hence, the instance methods and variables introduced by "Animal" (such as methods for breathing and eating, or a variable for storing an animal's weight) would also apply to instances of "Dog", because "Dog" *inherits* these from "Animal", its parent class. As a result, any given dog instance would be able to breath and eat, and would have a weight.

A *metaclass* is a class whose instances are class objects, and whose instance methods and instance variables (as described above) are therefore the methods and variables of class objects. For this reason, a *metaclass* is said to define *class methods* — the methods that a class object performs. For example, the metaclass of "Animal" might be "AnimalMClass", which defines the methods that can be invoked on class "Animal" (such as, to create Animal instances — objects that are not classes, like an individual pig or cat or elephant or dog).

Note: It is important to distinguish the methods of a class object (that is, the methods that can be invoked on the class object, which are defined by its metaclass) from the methods that the class defines for its instances.

To summarize: the *parent* of a class provides inherited methods that the class's *instances* can perform; the *metaclass* of a class provides class methods that the *class itself* can perform. The distinctions between parent class and metaclass are summarized in Figure 3.

Characteristics of parent class vs. metaclass

Figure 3.   A class has both parent classes and a metaclass.
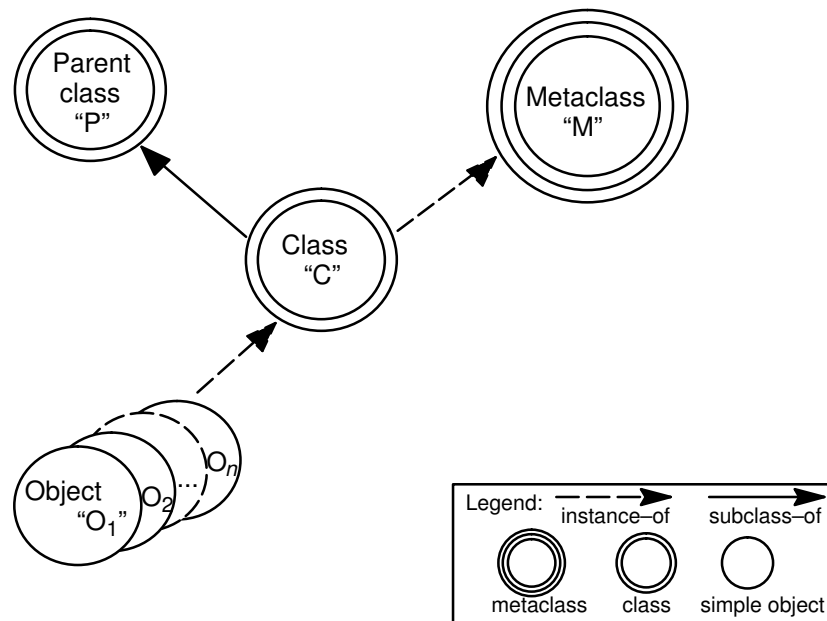
Any class "C" has both a *metaclass* and one or more *parent* class(es).

- •The *parent* class(es) of "C" provide the inherited *instance methods* that individual instances (objects "O_i") of class "C" can perform. Instance methods that an instance "O_i" performs might include (a) initializing itself, (b) performing computations using its instance variables, (c) printing its instance variables, or (d) returning its size. New instance methods are defined by "C" itself, in addition to those inherited from C's parent classes.

- • The *metaclass* "M" defines the *class methods* that class "C" can perform. For example, class methods defined by metaclass "M" include those that allow "C" to (a) inherit its parents's instance methods and instance variables, (b) tell its own name, (c) create new instances, and (d) tell how many instance methods it supports. These methods are inherited from SOMClass. Additional methods supported by "M" might allow "C" to count how many instances it creates.

- • Each class "C" has one or more parent classes and exactly one metaclass. (The single exception is SOMObject, which has no parent class.) Parent class(es) must be explicitly identified in the IDL declaration of a class. (SOMObject is given as a parent if no subsequently-derived class applies.) If a metaclass is not explicitly listed, the SOM run time will determine an applicable metaclass.

- • An instance of a metaclass is always another *class object*. For example, class "C" is an instance of metaclass "M". SOMClass is the SOM-provided metaclass from which all subsequent metaclasses are derived.

A metaclass has its own inheritance hierarchy (through its parent classes) that is independent of its instances' inheritance hierarchies. In Figure 4, a sequence of classes is defined (or derived), stemming from **SOMObject**. The child class (or subclass) at the end of this line ("C_2") inherits instance methods from all of its ancestor classes (here, **SOMObject** and "C_1"). An instance created by "C2" can perform any of these instance methods. In an analogous manner, a line of metaclasses is defined, stemming from **SOMClass.** Just as a new class is derived from an existing class (such as **SOMObject**), a new metaclass is derived from an existing metaclass (such as **SOMClass**). In this example, both **SOMObject** and class "C1" are instances of the **SOMClass** metaclass, whereas class "C2" is an instance of metaclass "M2", which inherits from **SOMClass**.

Derivation of parent classes and metaclasses
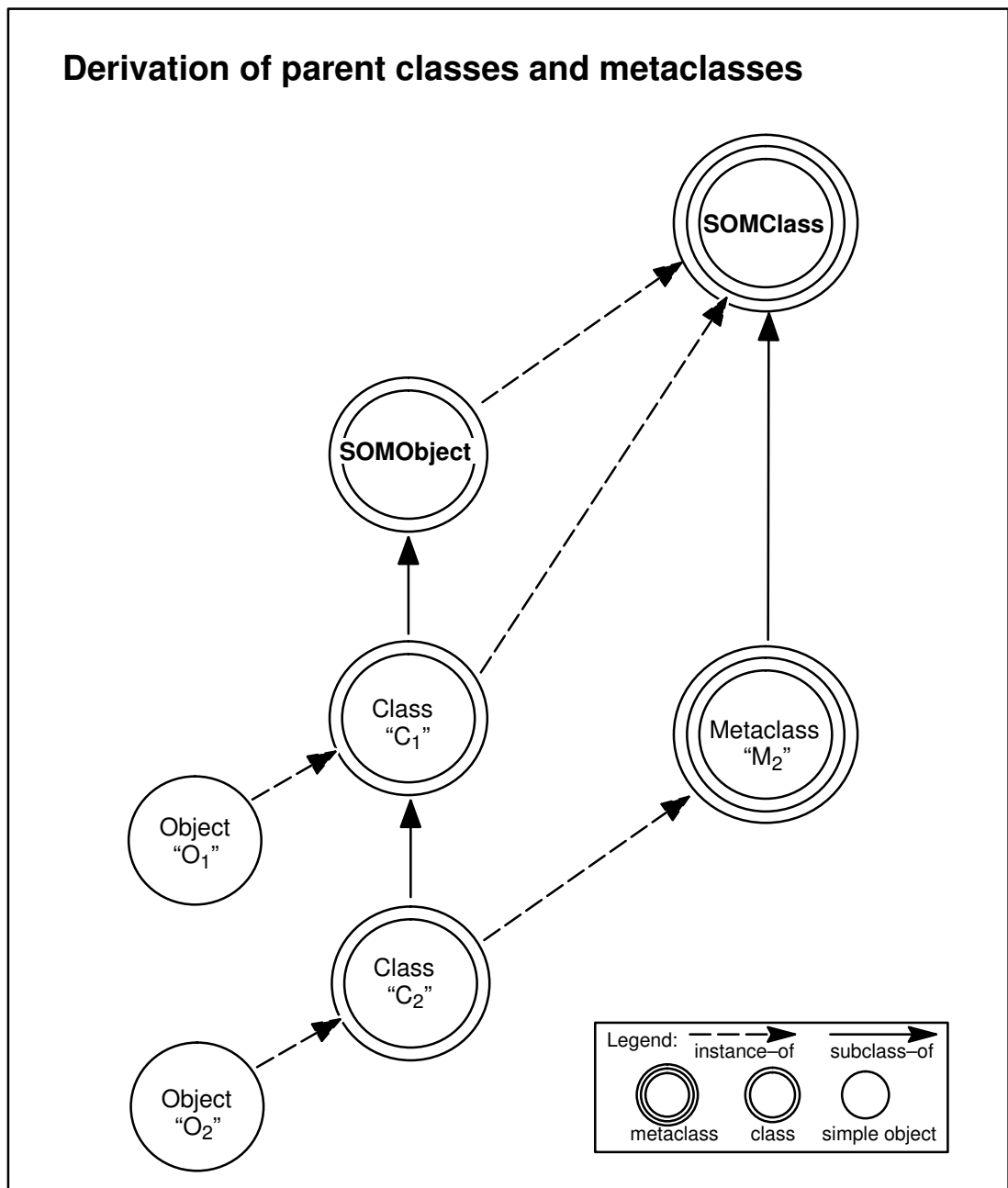
Figure 4.   Parent classes and metaclasses each have their own independent inheritance hierarchies.

SOM-derived metaclasses

As previously discussed, a class object can perform any of the class methods that its metaclass defines. New metaclasses are typically created to modify existing class methods or introduce new class method(s). Chapter 10, "Metaclass Framework," discusses metaclass programming.

Three factors are essential for effective use of metaclasses in SOM:

* First, every class in SOM is an object that is implemented by a metaclass.

* Second, programmers can define and name new metaclasses, and can use these metaclasses when defining new SOM classes.

* Finally, and most importantly, metaclasses cannot interfere with the fundamental guarantee required of every OOP system: specifically, any code that executes without method-resolution error on instances of a given class will also execute without method-resolution errors on <u>instances of any subclass</u> of this class.

Surprisingly, SOM is currently the only OOP system that can make this final guarantee while also allowing programmers to explicitly define and use named metaclasses. This is possible because SOM automatically determines an appropriate metaclass that supports this guarantee, automatically deriving new metaclasses by subclassing at run time when this is necessary.

To better understand this concept, consider the situation in Figure 5. Here, class "A" is an instance of metaclass "AMeta". Assume that "AMeta" supports a method "bar" and that "A" supports a method "foo" that uses the expression "_bar(_somGetClass(somSelf))." That is, method "foo" invokes "bar" on the class of the object on which "foo" is invoked. For example, when method "foo" is invoked on an instance of class "A" (say, object "O_1"), this in turn invokes "bar" on class "A" itself.

Example of metaclass incompatibility
(This cannot occur in SOM)

```
interface B:A {      ...
 implementation {
  metaclass = BMeta;
   ...
 };
}:
```

Legend: instance–of   subclass–of

metaclass   class   simple object
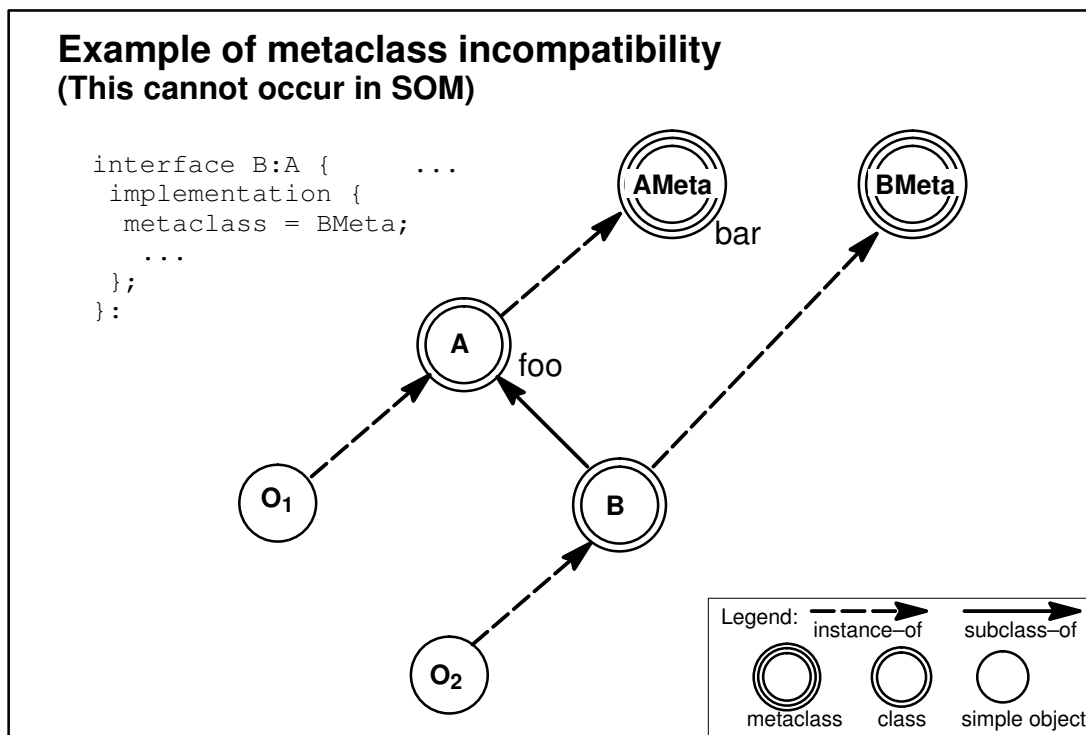
Figure 5.   Example of Metaclass Incompatibility.

Now consider what happens when "A" is subclassed by "B", a class that has the explicit metaclass "BMeta" declared in its SOM IDL source file, as shown by the code in Figure 5. If the class hierarchy were formed as in Figure 5, then an invocation of "foo" on "O_2" would fail, because metaclass "BMeta" does not support the "bar" method introduced by "AMeta".

There is only one way that "BMeta" can support this specific method — by inheriting it from "AMeta" ("BMeta" could introduce another method named "bar", but this would be a *different* method from the one introduced by "AMeta"). Therefore, in this example, because "BMeta" is not a subclass of "AMeta", "BMeta" cannot be allowed to be the metaclass of "B". That is, "BMeta" is not compatible with the requirements placed on "B" by the fundamental principle of OOP referred to above. This situation is referred to as *metaclass incompatibility*.

SOM does not allow hierarchies with metaclass incompatibilities. Instead, SOM automatically builds *derived metaclasses* when this is necessary. The resulting class hierarchy in this example is depicted in Figure 6, where SOM has automatically built the metaclass "DerivedMetaclass". This ensures that the invocation of method "foo" on instances of class "B" will not fail, and also ensures that the desired class methods provided by "BMeta" will be available on class "B".
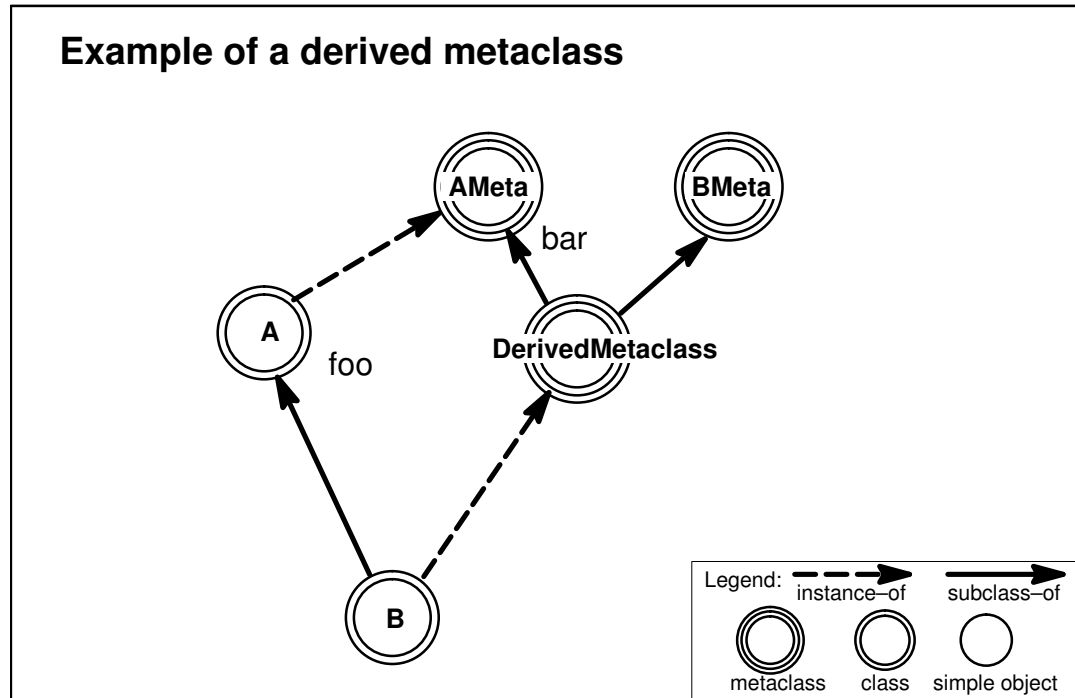

Figure 6. Example of a Derived Metaclass.

There are three important aspects of SOM's approach to derived metaclasses:

- First, the creation of SOM-derived metaclasses is integrated with programmer-specified metaclasses. If a programmer-specified metaclass already supports all the class methods and variables needed by a new class, then the programmer-specified metaclass will be used as is.

- Second, if SOM must derive a different metaclass than the one explicitly indicated by the programmer (in order to support all the necessary class methods and variables), then the SOM-derived metaclass inherits from the explicitly indicated metaclass first. As a result, the method procedures defined by the specified metaclass take precedence over other possibilities (see the following section on inheritance and the discussion of resolution of ambiguity in the case of multiple inheritance).

- Finally, the class methods defined by the derived metaclass invoke the appropriate initialization methods of its parents to ensure that the class variables of its instances are correctly initialized.

As further explanation for the automatic derivation of metaclasses, consider the following multiple-inheritance example. In Figure 7, class "C" does not have an explicit metaclass declaration in its SOM IDL, yet its parents do. As a result, class "C" requires a derived metaclass. (If you still have trouble following the reasoning behind derived metaclasses, ask yourself the following question: What class should "C" be an instance of? After a bit of reflection, you will conclude that, if SOM did not build the derived metaclass, you would have to do so yourself.)

Multiple inheritance in SOM requires derived metaclasses

```
interface C : A,B {
...
 implementation {
   ...
 };
}:
```
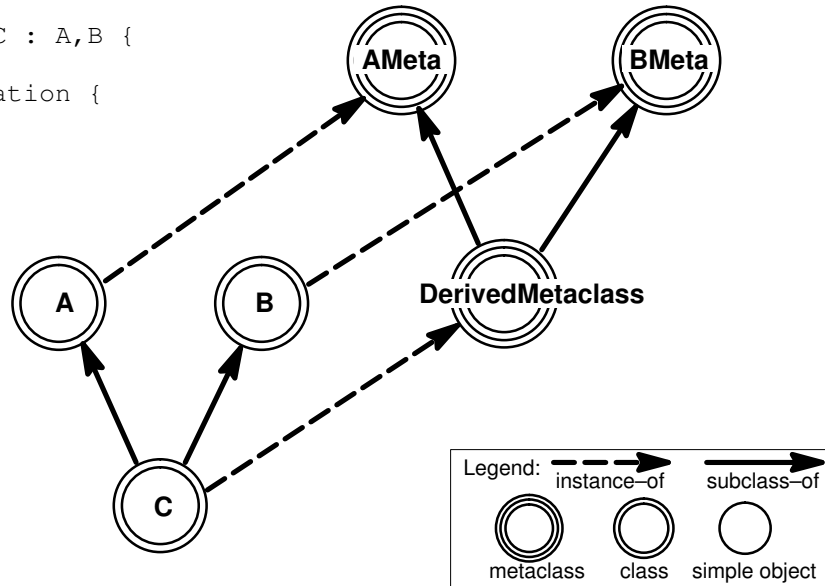
Figure 7.   Multiple Inheritance requires Derived Metaclasses.

In summary, SOM allows and encourages the definition and explicit use of named metaclasses. With named metaclasses, programmers can not only affect the behavior of class instances by choosing the parents of classes, but they can also affect the behavior of the classes themselves by choosing their metaclasses. Because the behavior of classes in SOM includes the implementation of inheritance itself, metaclasses in SOM provide an extremely flexible and powerful capability allowing classes to package solutions to problems that are otherwise very difficult to address within an OOP context.

At the same time, SOM is unique in that it relieves programmers of the responsibility for avoiding metaclass incompatibility when defining a new class. At first glance, this might seem to be merely a useful (though very important) convenience. But, in fact, it is absolutely essential, because SOM is predicated on binary compatibility with respect to changes in class implementations.

A programmer might, at one point in time, know the metaclasses of all ancestor classes of a new subclass, and, as a result, be able to explicitly derive an appropriate metaclass for the new class. Nevertheless, SOM must guarantee that this new class will still execute and perform correctly when any of its ancestor class's implementations are changed (which could even include specifying different metaclasses). Derived metaclasses allow SOM to make this guarantee. A SOM programmer need never worry about the problem of metaclass incompatibility; SOM does this for the programmer. Instead, explicit metaclasses can simply be used to "add in" whatever behavior is desired for a new class. SOM automatically handles anything else that is needed. Chapter 10 provides useful examples of such metaclasses. A SOM programmer should find numerous applications for the techniques that are illustrated there.

5.2 Inheritance

One of the defining aspects of an object model is its characterization of inheritance. This section describes SOM's model for inheritance.

A class in SOM defines an implementation for objects that support a specific interface:

- The *interface* defines the <u>methods</u> supported by objects of the class, and is specified using SOM IDL.

- The *implementation* defines what <u>instance variables</u> implement an object's state and what <u>procedures</u> implement its methods.

New classes are derived (by subclassing) from previously existing classes through inheritance, specialization, and addition. <u>Subclasses inherit interface</u> from their parent classes: any method available on instances of a class is also available on instances of any class derived from it (either directly or indirectly). <u>Subclasses also inherit implementation</u> (the procedures that implement the methods) from their parent classes *unless* the methods are *overridden* (redefined or specialized). In addition, a subclass may introduce new instance methods and instance variables that will be inherited by other classes derived from it.

SOM also supports *multiple inheritance*. That is, a class may be derived from (and may inherit interface and implementation from) multiple parent classes. Note: Multiple inheritance is available only to SOM classes whose interfaces are specified in IDL, and not to SOM classes whose interfaces are specified in SOM's earlier interface definition language, OIDL. See Appendix B for information on how to automatically convert existing OIDL files to IDL.

It is possible under multiple inheritance to encounter potential conflicts or ambiguities with respect to inheritance. All multiple inheritance models must face these issues, and resolve the ambiguities in some way. For example, when multiple inheritance is allowed, <u>it is possible that a class will inherit the same method or instance variable from different parents</u> (because each of these parents has some common ancestor that introduces the method or instance variable). In this situation, a SOM subclass inherits only one implementation of the method or instance variable. (The implementation of an instance variable within an object is just the location where it is stored. The implementation of a method is a procedure pointer, stored within a method table.) The following illustration addresses the question of which method implementation would be inherited.

Consider the situation in Figure 8. Class "W" defines a method "foo", implemented by procedure "proc1". Class "W" has two subclasses, "X" and "Y". Subclass "Y" overrides the implementation of "foo" with procedure "proc2". Subclass "X" does not override "foo". In addition, classes "X" and "Y" share a common subclass, "Z". That is, the IDL interface statement for class "Z" lists its parents as "X" and "Y" in that order.

The question is thus: which implementation of method "foo" does class "Z" inherit — procedure "proc1" defined by class "W", or procedure "proc2" defined by class "Y"? The procedure for performing inheritance that is defined by SOMClass resolves this ambiguity by using the *left path precedence* rule: when the same method is inherited from multiple ancestors, the procedure used to support the method is <u>the one used by the *leftmost ancestor* from which the method is inherited</u>. (The ordering of parent classes is determined by the order in which the class implementor lists the parents in the IDL specification for the class.)

In Figure 8, then, class "Z" inherits the implementation of method "foo" defined by class "W" (procedure "proc1"), rather than the implementation defined by class "Y" (procedure "proc2"), because "X" is the leftmost ancestor of "Z" from which the method "foo" is inherited. This rule may be interpreted as giving priority to classes whose instance interfaces are mentioned first in IDL interface definitions.

Resolution of multiple-inheritance ambiguities

Class "W"   foo ⟶ proc1

Class "X"   Class "Y"

foo ⟶ proc1   foo ⟶ proc2

Class "Z"   foo ⟶ **??**

Method resolution in class "Z":
 foo ⇒ left path precedence: proc1 (not proc2)
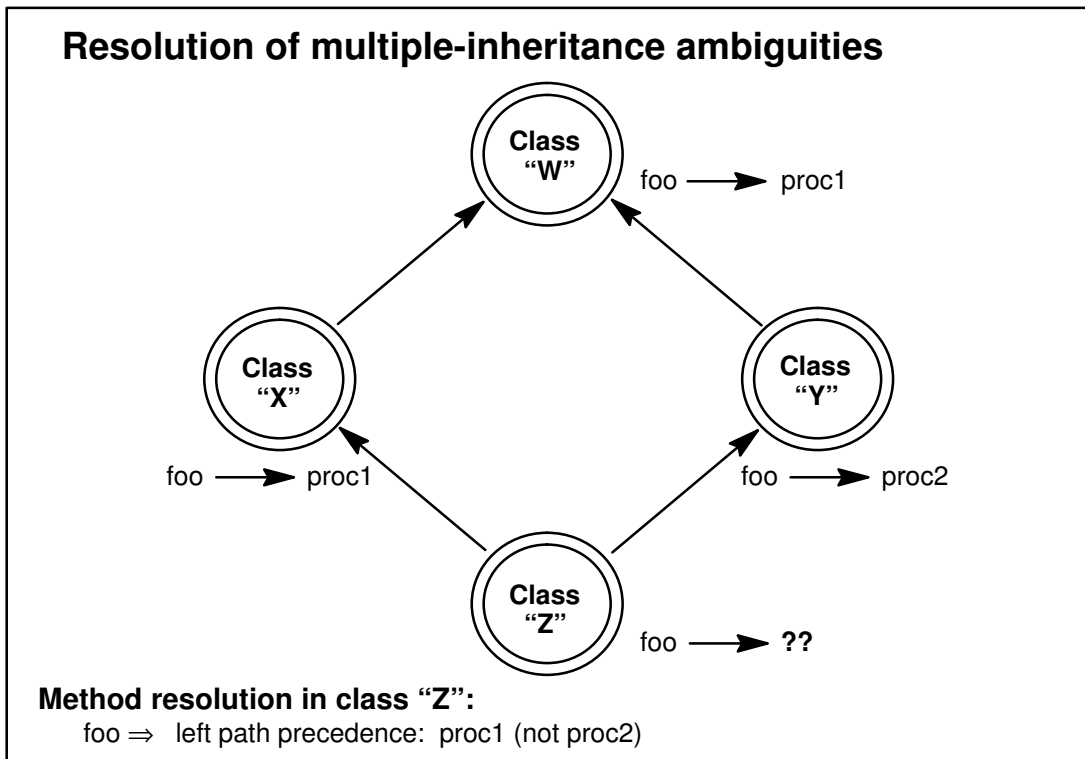
Figure 8.   SOMClass uses the *left path precedence* rule to resolve some multiple inheritance ambiguities.

If a class implementor decides that the default inherited implementation is not appropriate (for example, procedure "proc2" is desired), then SOM IDL allows the class designer to select the parent whose implementation is desired. For more information concerning this approach, see the **select** modifier, which is documented in the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."

Note: Alternatively, an explicit metaclass for class "Z" could be introduced to change the way methods are inherited. However, this would be a fairly serious step to take — it would also affect the semantics of inheritance for all of Z's descendant classes. Also, this would be done by overriding **somInitMIClass**, which is strongly discouraged until such time that SOMobjects includes the Cooperative Metaclass among its officially supported interface to the Metaclass Framework.

Resolution of multiple-inheritance ambiguities

Class "X"   Class "Y"

bar (of type *T1*)   bar (of type *T2*)

Class "Z"

? bar ?
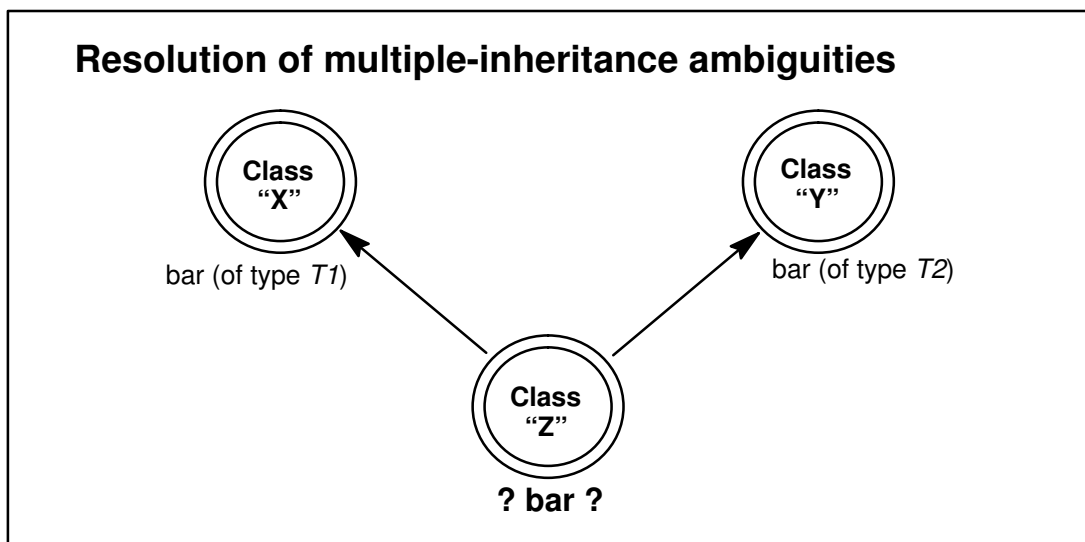
Figure 9.   Some multiple inheritance ambiguities are illegal in IDL.

Another conflict that may arise with the use of multiple inheritance is when two ancestors of a class define different methods (in general, with different signatures) with the same name. For example, consider Figure 9. Class "X" defines a method "bar" with type *T1*, and class "Y" defines a method "bar" with type *T2*. Class "Z" is derived from both "X" and "Y", and "Z" does not override method "bar".

This example illustrates a method name that is "overloaded" — that is, used to name two entirely different methods (note that overloading is completely unrelated to overriding). This is not necessarily a difficult problem to handle. Indeed, the run-time SOM API allows the construction of a class that supports the two different "bar" methods illustrated in FIgure 9. (They are implemented using two different method-table entries, each of which is associated with its introducing class.)

However, the interface to instances of such classes can*not* be defined using IDL. IDL specifically forbids the definition of interfaces in which method names are overloaded. Furthermore, within SOM itself, the use of such classes can lead to anomalous behavior unless care is taken to avoid the use of name-lookup method resolution (discussed in the following section), since, in this case, a method name alone does not identify a unique method. For this reason, (statically declared) multiple-inheritance classes in SOM are currently restricted to those whose instance interfaces can be defined using IDL. Thus, the above example cannot be constructed with the aid of the SOM Compiler.

5.3 Method Resolution

Method resolution is the step of determining which procedure to execute in response to a method invocation. For example, consider this scenario:

- Class "Dog" introduces a method "bark", and

- A subclass of "Dog", called "BigDog", overrides "bark", and

- A client program creates an instance of either "Dog" or "BigDog" (depending on some run-time criteria) and invokes method "bark" on that instance.

Method resolution is the process of determining, at run time, which method procedure to execute in response to the method invocation (either the method procedure for "bark" defined by "Dog", or the method procedure for "bark" defined by "BigDog"). This determination depends on whether the receiver of the method (the object on which it is invoked) is an instance of "Dog" or "BigDog" (or perhaps depending on some other criteria).

SOM allows class implementors and client programs considerable flexibility in deciding how SOM performs method resolution. In particular, SOM supports three mechanisms for method resolution, described in order of increased flexibility and increased computational cost: offset resolution, name-lookup resolution, and dispatch-function resolution.

Offset resolution

When using SOM's C and C++ language bindings, offset resolution is the default way of resolving methods, because it is the fastest. For those familiar with C++, this is roughly equivalent to the C++ "virtual function" concept.

Although offset resolution is the fastest technique for method resolution, it is also the most constrained. Specifically, using offset resolution requires these constraints:

- The name of the method to be invoked must be known at compile time,

- The name of the class that introduces the method must be known at compile time (although not necessarily by the programmer), and

- The method to be invoked must be part of the introducing class's static (IDL) interface definition.

To perform offset method resolution, SOM first obtains a *method token* from a global data structure associated with the class that introduced the method. This data structure is called the *ClassData structure*. It includes a method token for each method the class introduces. The method token is then used as an "index" into the receiver's *method table,* to access the appropriate method procedure. Because it is known at compile time which class introduces the method and where in that class's ClassData structure the method's token is stored, offset resolution is quite efficient. The cost of offset method resolution is currently about twice the cost of calling a C function using a pointer loaded with the function address.

An object's method table is a table of pointers to the procedures that implement the methods that the object supports. This table is constructed by the object's class and is shared among the class instances. The method table built by class (for its instances) is referred to as the class's *instance method table*. This is useful terminology, since, in SOM, a class is itself an object with a method table (created by its metaclass) used to support method calls on the class.

Usually, offset method resolution is sufficient; however, in some cases, the more flexible name-lookup resolution is required.

Name-lookup resolution

Name-lookup resolution is similar to the method resolution techniques employed by Objective-C and Smalltalk. It is currently about five times slower than offset resolution. It is more flexible, however. In particular, name-lookup resolution, unlike offset resolution, can be used when:

- The name of the method to be invoked isn't known until run time, or

- The method is added to the class interface at run time, or

- The name of the class introducing the method isn't known until run time.

For example, a client program may use two classes that define two different methods of the same name, and it might not be known until run time which of the two methods should be invoked (because, for example, it will not be known until run time which class's instance the method will be applied to).

Name-lookup resolution is performed by a class, so it requires a method call. (Offset resolution, by contrast, requires no method calls.) To perform name-lookup method resolution, the class of the intended receiver object obtains a method procedure pointer for the desired method that is appropriate for its instances. In general, this will require a name-based search through various data structures maintained by ancestor classes.

Offset and name-lookup resolution achieve the same net effect (that is, they select the same method procedure); they just achieve it differently (via different mechanisms for locating the method's method token). Offset resolution is faster, because it does not require searching for the method token, but name-lookup resolution is more flexible.

When defining (in SOM IDL) the interface to a class of objects, the class implementor can decide, for each method, whether the SOM Compiler will generate usage bindings that support name-lookup resolution for invoking the method. Regardless of whether this is done, however, application programs using the class can have SOM use either technique, on a per-method-call basis. Chapter 3, "Using SOM Classes in Client Programs," describes how client programs invoke methods."

Dispatch-function resolution

Dispatch-function resolution is the slowest, but most flexible, of the three method-resolution techniques. Dispatch functions permit method resolution to be <u>based on arbitrary rules associated with the class</u> of which the receiving object is an instance. Thus, a class implementor has complete freedom in determining how methods invoked on its instances are resolved.

With both offset and name-lookup resolution, the net effect is the same — the method procedure that is ultimately selected is the one supported by the class of which the receiver is an instance. For example, if the receiver is an instance of class "Dog", then Dog's method procedure will be selected; but if the receiver is an instance of class "BigDog", then BigDog's method procedure will be selected.

By contrast, dispatch-function resolution allows a class of instances to be defined such that the method procedure is selected using some other criteria. For example, the method procedure could be selected on the basis of the arguments to the method call, rather than on the receiver. For more information on dispatch-function resolution, see the description and examples for the **somDispatch**, and **somOverrideMtab** methods in the *SOMobjects Developer Toolkit Programmers Reference Manual*.

Customizing method resolution

Customizing method resolution requires the use of metaclasses that override SOMClass methods. This is not recommend without use of the Cooperative Metaclass that guarantees correct operation of SOMobjects in conjunction with such metaclasses. SOMobjects users who require this functionality should request access to the experimental Cooperative Metaclass used to implement the SOMobjects Metaclass Framework. Metaclasses implemented using the Cooperative Metaclass may have to be reprogrammed in the future when SOMobjects introduces an officially supported Cooperative Metaclass.

The four kinds of SOM methods

SOM supports four different kinds of methods: *static* methods, *nonstatic* methods, *dynamic* methods, and *direct-call procedures*. The following paragraphs explain these four method categories and the kinds of method resolution available for each.

Static methods

These are similar in concept to C++ virtual functions. Static methods are normally invoked using offset resolution via a method table, as described above, but all three kinds of method resolution are applicable to static methods. Each different static method available on an object is given a different slot in the object's method table. When SOMobjects Toolkit language bindings are used to implement a class, the SOM IDL **method** modifier can be specified to indicate that a given method is static; however, this modifier is rarely used since it is the default for SOM methods.

Static methods introduced by a class can be overridden (redefined) by any descendant classes of the class. When SOMobjects language bindings are used to implement a class, the SOM IDL **override** modifier is specified to indicate that a class overrides a given inherited method. When a static method is resolved using offset resolution, it is not important which interface is accessing the method — the actual class of the object determines the method procedure that is selected.

Note: All SOM IDL modifiers are described in the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."

Nonstatic methods

These methods are similar in concept to C++ nonstatic member functions (that is, C++ functions that are not virtual member functions and are not static member functions). Nonstatic methods are normally invoked using offset resolution, but all three kinds of method resolution are applicable to nonstatic methods. When the SOMobjects language bindings are used to implement a class, the SOM IDL **nonstatic** modifier is used to indicate that a given method is nonstatic.

Like static methods, nonstatic methods are given individual positions in method tables. However, nonstatic methods cannot be overridden. Instead, descendants of a class that introduces a nonstatic method can use the SOM IDL **reintroduce** modifier to "hide" the original nonstatic method with another (nonstatic or static) method of the same name and signature. When a nonstatic method is resolved, selection of the specific method procedure is determined by the interface that is used to access the method.

Dynamic methods

These methods are *not* declared when specifying an object interface using IDL. Instead, they are registered with a class object at run time using the method **somAddDynamicMethod**. Because there is no way for SOM to know about dynamic methods before run time, offset resolution is not available for dynamic methods. Only name-lookup or dispatch-function resolution can be used to invoke dynamic methods. Dynamic methods cannot be overridden.

Direct-call procedures

These are similar in concept to C++ static member functions. Direct-call procedures are not given positions in SOM method tables, but are accessed directly from a class's **ClassData** structure. Strictly speaking, none of the previous method-resolution approaches apply for invoking a direct-call procedure, although SOMobjects language bindings provide the same invocation syntax for direct-call procedures as for static or nonstatic methods. Direct-call procedures cannot be overridden, but they can be reintroduced. When SOMobjects language bindings are used to implement a class, the SOM IDL **procedure** modifier is used to indicate that a given method is a direct-call procedure. Note: Methods having the **procedure** modifier cannot be invoked remotely using DSOM.

5.4 Implementing SOM Classes

The *interface* to a class of objects contains the information that a client must know to use an object — namely, the signatures of its methods and the names of its attributes. The interface is described in a formal language independent of the programming language used to implement the object's methods. In SOM, the formal language used to define object interfaces is the Interface Definition Language (described in Chapter 4, "SOM IDL and the SOM Compiler").

The *implementation* of a class of objects (that is, the procedures that implement the methods and the instance variables that store an object's state) is written in the implementor's preferred programming language. This language can be object-oriented (for instance, C++) or procedural (for instance, C).

A completely implemented class definition, then, consists of two main files:

- An IDL specification of the interface to instances of the class — the *interface definition file* (or .idl file) and

- Method procedures written in the implementor's language of choice — the *implementation file*.

The SOM Compiler provides the link between those two files: 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. Then, the class implementor modifies this template file to fully implement the class's methods. That process is the subject of the remainder of this chapter.

The SOM Compiler can also update the implementation file to reflect changes subsequently made to a class's interface definition file (the .idl file). These *incremental updates* include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method declarations in the IDL specification. These updates to the implementation file, however, do *not* disturb existing code in the method procedures. These updates are discussed further in "Running incremental updates of the implementation template file" later in this section.

For C programmers, the SOM Compiler generates a *<filestem>*.c file. For C++ programmers, the SOM Compiler generates a *<filestem>*.C file (for AIX) or a *<filestem>*.cpp file (for OS/2 or Windows). To specify whether the SOM Compiler should generate a C or C++ implementation template, set the value of the SMEMIT environment variable, or use the **–s** option when running the SOM Compiler. (See "The SOM Compiler" in Chapter 4, "SOM IDL and the SOM Compiler.")

Note: As this chapter describes, a SOM class can be implemented by using C++ to define the instance variables introduced by the class and to define the procedures that implement the overridden and introduced methods of the class. Be aware, however, that the C++ class defined by the C++ usage bindings for a SOM class (described in Chapter 3) *cannot* be subclassed in C++ to create new C++ or SOM classes.†

The implementation template

Consider the following IDL description of the "Hello" class:

```
#include <somobj.idl>

interface Hello : SOMObject
{
    void sayHello();
    // This method outputs the string "Hello, World!".
};
```

---

† The reason why the C++ implementation of a SOM class involves the definition of C++ procedures (not C++ methods) to support SOM methods is that there is no language-neutral way to call a C++ method. Only C++ code can call C++ methods, and this calling code must be generated by the same compiler that generates the method code. In contrast, the method procedures that implement SOM methods must be callable from any language, without knowledge on the part of the object client as to which language is used to implement the resolved method procedure.

From this IDL description, the SOM Compiler generates the following C implementation template, hello.c (a C++ implementation template, hello.C or hello.cpp, is identical except that the *#include*d file is <hello.xih> rather than <hello.ih>):

```
#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");
}
```

<u>The first line</u> defines the "Hello_Class_Source" symbol, which 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 <class-Name>_Class_Source" is replaced by the directive "#define SOM_Module_<module-Name>_Source".

<u>The second line</u> (`#include <hello.ih>` for C, or `#include <hello.xih>` for C++) includes the SOM-generated implementation header file. This file defines a **struct** holding the class's instance variables, macros for accessing instance variables, macros for invoking parent methods, and so forth.

Stub procedures for methods

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.

For method "sayHello" above, the SOM Compiler generates the following prototype of the stub procedure:

```
SOM_Scope void SOMLINK sayHello(Hello somSelf, Environment *ev)
```

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 is defined by SOM; it represents the keyword needed to link to the C or C++ compiler, and 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 <u>name</u> of the procedure that implements the method. If no **functionprefix** modifier has been specified for the class, then the procedure name is the same as the method name. If a **functionprefix** modifier is in effect, then the procedure name is generated by prepending the specified prefix to the method name. For example, if the class definition contained the following statement:

```
functionprefix = xx_;
```

then the prototype of the stub procedure for method "sayHello" would be:

```
SOM_Scope void SOMLINK xx_sayHello(Hello somSelf, Environment *ev)
```

The **functionprefix** *cannot* be

```
<classname>_
```

since this is used in method invocation macros defined by the C usage bindings.

Following the procedure name is the <u>formal parameter list</u> 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 macros defined in the implementation header file rely on this convention.) The **somSelf** parameter is a <u>pointer to an object that is an instance of the class</u> being implemented (here, class "Hello") or an instance of a class derived from it.

Unless the IDL specification of the class included the **callstyle=oidl** modifier, then the formal parameter list will include one or two additional parameters before the parameters declared in the IDL specification: an **(Environment *ev)** input/output parameter, which permits the return of exception information, and, if the IDL specification of the method includes a context specification, a **(Context *ctx)** input parameter. These parameters are prescribed by the CORBA standard. For more information on using the **Environment** and **Context** parameters, see the section entitled "Exceptions and error handling" in Chapter 3, "Using SOM Classes in Client Programs," and the book *The Common Object Request Broker: Architecture and Specification*, published by Object Management Group and X/Open.

The first statement in the stub procedure for method "sayHello" is the statement:

```
/* HelloData *somThis = HelloGetData(somSelf); */
```

This statement is enclosed in comments only when the class does *not* introduce any instance variables. The purpose of this statement, for classes that do introduce instance variables, is to initialize a local variable (**somThis**) that points to a *structure* representing the instance variables introduced by the class. The **somThis** pointer is used by the macros defined in the "Hello" implementation header file to access those instance variables. (These macros are described below.) In this example, the "Hello" class introduces no instance variables, so the statement is commented out. If instance variables are later added to a class that initially had none, then the comment characters can be removed if access to the variable is required.

The "HelloData" type and the "HelloGetData" macro used to initialize the **somThis** pointer are defined in the implementation header file. Within a method procedure, class implementors can use the **somThis** pointer to access instance data, or they can use the convenience macros defined for accessing each instance variable, as described below.

To implement a method so that it can modify a local copy of an object's instance data without affecting the object's real instance data, declare a variable of type *<className>***Data** (for example, "HelloData") and assign to it the structure that **somThis** points to; then make the **somThis** pointer point to the copy. For example:

```
HelloData myCopy = *somThis;
somThis = &myCopy;
```

Next in the stub procedure for method "sayHello" is the statement:

```
HelloMethodDebug("Hello","sayHello");
```

This statement facilitates 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 (that is, if global variable **SOM_TraceLevel** is set to one in the calling program), the macro produces a message each time the method procedure is entered. (See Chapter 3, "Using SOM Classes in Client Programs," for information on debugging with SOM.)

Debugging can be permanently disabled (regardless of the **SOM_TraceLevel** setting in the calling program) by redefining the *<className>*MethodDebug macro as **SOM_NoTrace(c,m)** following the #include directive for the implementation header file. (This can yield a slight performance improvement.) For example, to permanently disable debugging for the "Hello" class, insert the following lines in the hello.c implementation file following the line "#include hello.ih" (or "#include hello.xih," for classes implemented in C++):

```
#undef HelloMethodDebug
#define HelloMethodDebug(c,m) SOM_NoTrace(c,m)
```

The way in which the stub procedure ends is determined by whether the method is a new or an overriding method.

- For non-overriding (new) methods, the stub procedure ends with a return statement (unless the return type of the method is **void**). The class implementor should customize this return statement.

- For overriding methods, the stub procedure ends by making a "parent method call" for each of the class's 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 (for example, if some other value is to be returned, or if the parent method calls should be made before the method procedure's own processing). See the next section for a discussion of parent method calls.

If a **classinit** modifier was specified to designate a user-defined procedure that will initialize the "Hello" class object, as in the statement:

```
classinit = HInit;
```

then the implementation template file would include the following stub procedure for "HInit", in addition to the stub procedures for Hello's methods:

```
void  SOMLINK HInit(SOMClass *cls)
{

}
```

This stub procedure is then filled in by the class implementor. If the class definition specifies a **functionprefix** modifier, the **classinit** procedure name is generated by prepending the specified prefix to the specified **classinit** name, as with other stub procedures.

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 SOM provides for manipulating classes and objects. Chapter 3, "Using SOM Classes in Client Programs," discusses these functions, methods, and macros.

In addition to the functions, methods, and macros SOM provides for both class clients and class implementors, SOM provides two facilities especially for class implementors. They are for (1) accessing instance variables of the object responding to the method and (2) making parent method calls, as follows.

Accessing internal instance variables

To access internal instance variables, class implementors can use either of the following forms:

_variableName

somThis–>variableName

To access internal instance variables "a", "b", and "c", for example, the class implementor could use either _a, _b, and _c, or somThis–>a, somThis–>b, and somThis–>c. These expressions can appear on either side of an assignment statement. The **somThis** pointer must be properly initialized in advance using the <className>**GetData** procedure, as shown above.

Instance variables can be accessed only within the implementation file of the class that introduces the instance variable, and 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.) For C++ programmers, the _variableName form is available only if the macro **VARIABLE_MACROS** is defined (that is, **#define VARIABLE_MACROS**) in the implementation file prior to including the .xih file for the class.

Making parent method calls

In addition to macros for accessing instance variables, the implementation header file that the SOM Compiler generates also contains definitions of macros for making "parent method calls." When a class overrides a method defined by one or more of its parent classes, often the new implementation simply needs to augment the functionality of the existing implementation(s). Rather than completely re-implementing the method, the overriding method procedure can conveniently invoke the procedure that one or more of the parent classes uses to implement that method, then perform additional computation (redefinition) as needed. The parent method call can occur anywhere within the overriding method. (See Example 3 of the SOM IDL tutorial.)

The SOM-generated implementation header file defines the following macros for making parent-method calls from within an overriding method:

*<className>*_**parent**_*<parentClassName>_<methodName>*
(for each parent class of the class overriding the method), and

*<className>*_**parents**_*<methodName>*.

For example, given class "Hello" with parents "File" and "Printer" and overriding method **somInit** (the SOM method that initializes each object), the SOM Compiler defines the following macros in the implementation header file for "Hello":

```
Hello_parent_Printer_somInit
Hello_parent_File_somInit
Hello_parents_somInit
```

Each macro takes the same number and type of arguments as *<methodName>*. The *<className>*_**parent**_*<parentClassName>_<methodName>* macro invokes the implementation of *<methodName>* inherited from *<parentClassName>*. Hence, using the macro "Hello_parent_File_somInit" invokes File's implementation of **somInit**.

The *<className>*_**parents**_*<methodName>* macro invokes the parent method for *each* parent of the child class that supports *<methodName>*. That is, "Hello_parents_somInit" would invoke File's implementation of **somInit**, followed by Printer's implementation of **somInit**. The *<className>*_**parents**_*<methodName>* macro is redefined in the binding file each time the class interface is modified, so that if a parent class is added or removed from the class definition, or if *<methodName>* is added to one of the existing parents, the macro *<className>*_**parents**_*<methodName>* will be redefined appropriately.

Converting C++ classes to SOM classes

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 by using the following shorthand syntax:

```
_methodName(arg1, arg2, ...)
```

For example, if class *X* introduces or overrides methods *m1* and *m2*, then the C++ implementation of method *m1* can invoke method *m2* on its *somSelf* argument using *_m2(arg, arg2, ...)*, rather than *somSelf–>m2(arg1, arg2, ...)*, as would otherwise be required. (The longer form is also available.) Before the shorthand form in the implementation file is used, the macro **METHOD_MACROS** must be defined (that is, use **#define METHOD_MACROS**) prior to including the .xih file for the class.

Running incremental updates of the implementation template file

Refining the .idl file for a class is typically an iterative process. For example, after running the IDL source file through the SOM Compiler and writing some code in the implementation template file, the class implementor realizes that the IDL class interface needs another method or attribute, a method needs a different parameter, or any such changes.

As mentioned earlier, the SOM Compiler (when run using the **c** or **xc** emitter) assists in this development by reprocessing the .idl file and making *incremental updates* to the current implementation file. This modify-and-update process may in fact be repeated several times before the class declaration becomes final. Importantly, these updates do not disturb existing code for the method procedures. Included in the incremental update are 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 transferred to the implementation file, reformatted appropriately.

- If the interface to a method has changed, a new method procedure prototype is placed in the implementation file. As a precaution, however, the old prototype is also preserved within comments. The body of the method procedure is left untouched (as are the method procedures for all methods).

- Similarly left intact are preprocessor directives, data declarations, constant declarations, non-method functions, and additional comments — in essence, everything else in the implementation file.

Some changes to the .idl file are *not* reflected automatically in the implementation file after an incremental update. The class implementor must manually edit the implementation file after changes such as these in the .idl file:

- Changing the name of a class or a method.

- Changing the parents of a class (see also "If you change the parents of a class..." later in this topic).

- Changing a **functionprefix** class modifier statement.

- Changing the content of a **passthru** statement directed to the implementation (.c, .C, or .cpp) file. As previously emphasized, however, **passthru** statements are primarily recommended only for placing #include statements in a binding file (.ih, .xih, .h, or .xh file) used as a header file in the implementation file or in a client program.

- If the class implementor has placed "forward declarations" of the method procedures in the implementation file, those are not updated. Updates occur only for method prototypes that are part of the method procedures themselves.

Considerations to ensure that updates work

To ensure that the SOM Compiler can properly update method procedure prototypes in the implementation file, class implementors should avoid editing changes such as the following:

- A method procedure name should *not* be enclosed in parentheses in the prototype.

- A method procedure <u>name</u> must appear in the <u>first line</u> of the prototype, excluding comments and whitespace. Thus, a newline must *not* be inserted before the procedure name.

Error messages occur while updating an existing implementation file if it contains syntax that is not ANSI C. For example, "old style" method definitions such as the example on the left generate errors:

<u>Invalid "old" syntax</u>   <u>Required ANSI C</u>

```
void foo(x)              void foo(short x) {
short x;                  ...
{                        }
 ...
}
```

Similarly, error messages occur if anything in the .idl file would produce an implementation file that is not syntactically valid for C/C++ (such as nested comments). If update errors occur, either the .idl file or the implementation file may be at fault. One way to track down the problem is to run the implementation file through the C/C++ compiler. Or, move the existing implementation file to another directory, generate a completely new one from the .idl file, and then run *it* through the C/C++ compiler. One of these steps should pinpoint the error, if the compiler is strict ANSI.

Conditional compilation (using #if and #ifdef directives) in the implementation file can be another source of errors, because the SOM Compiler does not invoke the preprocessor (it simply recognizes and ignores those directives). The programmer should be careful when using conditional compilation, to avoid a situation such as shown below; here, with apparently two open braces and only one closing brace, the **c** or **xc** emitter would report an unexpected end-of-file:

Invalid syntax   Required matching braces

```
#ifdef FOOBAR                     #ifdef FOOBAR
 {                                 {
  ...                              ...
#else                             }
 {                                #else
  ...                              {
#endif                            ...
 }                                }
                                  #endif
```

If you change the parents of a class...

Because the implementation-file emitters never change any existing code within a previously generated implementation file, changing the parents of a class requires extremely careful attention by the programmer. For example, for overridden methods, changing a class's parents may invalidate previous parent-method calls provided by the template, and require the addition of new parent-method calls. Neither of these issues is addressed by the incremental update of previously generated method-procedure templates.

The greatest danger from changing the parents of a class, however, concerns the ancestor-initializer calls provided in the stub procedures for initializer methods. (For further information on ancestor initializer calls, see "Initializing and Uninitializing Objects" later in this chapter.) Unlike parent-method calls, ancestor-initializer calls are not optional — they must be made to all classes specified in a **directinitclasses** modifier, and these calls should always include the parents of the class (the default when no **directinitclasses** modifier is given). When the parents of a class are changed, however, the ancestor-initializer calls (which must be made in a specific order) are not updated.

The easiest way to deal with this problem is to change the method name of the previously generated initializer stub procedure in the implementation template file. Then, the SOM Compiler can correctly generate a completely new initializer stub procedure (while ignoring the renamed procedure). Once this is done, your customization code from the renamed initializer procedure can be "merged" into the newly generated one, after which the renamed initializer procedure can be deleted.

Compiling and linking

After you fill in the method stub procedures, the implementation template file can be compiled and linked with a client program as shown below. In these examples, the environment variable SOMBASE represents the directory in which SOM has been installed.

For AIX: When the client program (main.c) and the implementation file (hello.c) are written in C:

```
> xlc -I. -I$SOMBASE/include -o hello main.c hello.c \
    -L$SOMBASE/lib -lsomtk
```

When the client program and the implementation file are written in C++:

```
> xlC -I. -I$SOMBASE/include -o hello main.C hello.C \
    -L$SOMBASE/lib -lsomtk
```

For OS/2: When the client program (main.c) and the implementation file (hello.c) are in C:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include -Fe hello \
    main.c hello.c somtk.lib
```

When the client program and the implementation file are written in C++:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include -Fe hello \
      main.cpp hello.cpp somtk.lib
```

For Windows: When the client program (main.c) and the implementation file (hello.c) are in C:

```
> cl -AL -Zp1 -I. -I%SOMBASE%\include -Fehello \
      main.c hello.c llibcew.lib libw.lib somtk.lib main.def
```

When the client program and the implementation file are written in C++:

```
> cl -AL -Zp1 -I. -I%SOMBASE%\include -Fehello \
      main.cpp hello.cpp llibcew.lib libw.lib somtk.lib main.def
```

Windows compiler notes: The SOM Compiler and runtime libraries under Windows only provide support for Large Memory Model systems. The "–Oi" compile flag should be used whenever a SOM Windows program contains or calls any functions that return floats or doubles. The "–F" option can be used to set the stack size if STACKSIZE is not specified in a .def file. Additionally, for this manual, compile and link commands are demonstrated using the Visual C++ compiler. Sample Makefiles shipped with SOMobjects For Windows also show how to build with other compilers.

For all users: If the class definition (in the .idl file) changes, run the SOM Compiler again. This will <u>generate new header files</u>, and it will <u>update the implementation file</u> to include any:

- New comments,

- Stub procedures for any new methods, and

- Revised method procedure prototypes for methods whose signatures have been changed in the .idl file.

After rerunning the SOM Compiler, add to the implementation file the code for any newly added method procedures, and recompile the implementation file with the client program.

5.5 Initializing and Uninitializing Objects

This section discusses the initialization and uninitialization of SOM objects. Subsequent topics introduce the methods and capabilities that the SOMobjects Developer Toolkit provides to facilitate this.

Object creation is the act that enables the execution of methods on an object. In SOM, this means storing a pointer to a method table into a word of memory. This single act converts raw memory into an (uninitialized) SOM object that starts at the location of the method table pointer.

Object initialization, on the other hand, is a separate activity from object creation in SOM. Initialization is a capability supported by certain methods available on an object. An object's class determines the implementation of the methods available on the object, and thus determines its initialization behavior.

The instance variables encapsulated by a newly created object must be brought into a consistent state before the object can be used. This is the purpose of *initialization methods*. Because, in general, every ancestor of an object's class contributes instance data to an object, it is appropriate that each of these ancestors contribute to the initialization of the object.

SOM thus recognizes *initializers* as a special kind of method. One advantage of this approach is that special metaclasses are not required for defining constructors (class methods) that take arguments. Furthermore, a class can define multiple initializer methods, thus enabling its different objects to be initialized supporting different characteristics or capabilities. This results in simpler designs and more efficient programs.

The SOMobjects Toolkit provides an overall framework that class designers can easily exploit in order to implement default or customized initialization of SOM objects. This framework is fully supported by the SOM Toolkit emitters that produce the implementation template file. The following sections describe the declaration, implementation, and use of initializer (and uninitializer) methods.

Important: All code written prior to SOMobjects Release 2.1 using documented guidelines for the earlier initialization approach based on the **somInit** method (as well as all existing class binaries) continues to be fully supported and useful.

Initializer methods

As noted above, in the SOMobjects Toolkit each ancestor of an object contributes to the initialization of that object. Initialization of an object involves a chain of ancestor-method calls that, by default, are automatically determined by the SOM Compiler emitters. The SOMobjects framework for initialization of objects is based on the following approach:

1. SOMobjects recognizes *initializers* as a special kind of method, and supports a special mechanism for ordering the execution of ancestor-initializer method procedures. The **SOMObject** class introduces an initializer method, **somDefaultInit**, that uses this execution mechanism.

2. The SOM Compiler's emitters provide special support for methods that are *declared* as initializers in the .idl file. To supplement the **somDefaultInit** method, SOM class designers can also declare additional initializers in their own classes.

Two SOM IDL modifiers are provided for declaring initializer methods and controlling their execution, **init** and **directinitclasses**:

- The **init** modifier is required in order to designate a given method is an initializer; that is, to indicate that the method both uses and supports the object-initialization protocol described here.

- The **directinitclasses** modifier can be used to control the order of execution of initializer method procedures provided by the different ancestors of the class of an object.

For full definitions of **init** and **directinitclasses**, see the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."

Every SOM class has a list that defines (in sequential order) the ancestor classes whose initializer method procedures the class should invoke. If a class's IDL does not specify an explicit **directinitclasses** modifier, the default for this list is simply the class's parents — in left-to-right order.

Using the **directinitclasses** list and the actual run-time class hierarchy above itself, each class inherits from **SOMClass** the ability to create a data structure of type **somInitCtrl**. This structure is used to control the execution of initializers. Moreover, it represents a particular visit-ordering that reaches each class in the transitive closure of **directinitclasses** exactly once. To initialize a given object, this visit-ordering occurs as follows: While recursively visiting each ancestor class whose initializer method procedure should be run, SOMobjects first runs the initializer method procedures of all of that class's **directinitclasses** if they have not already been run by another class's initializers, with ancestor classes always taken in left-to-right order.

For example, Figure 10 shows an inheritance hierarchy along with the ordering produced when an instance of the class numbered **7** is initialized, assuming that each class simply uses its parents as its **directinitclasses**. Note that the class numbered **3** is at the top of a diamond.

A default initializer ordering
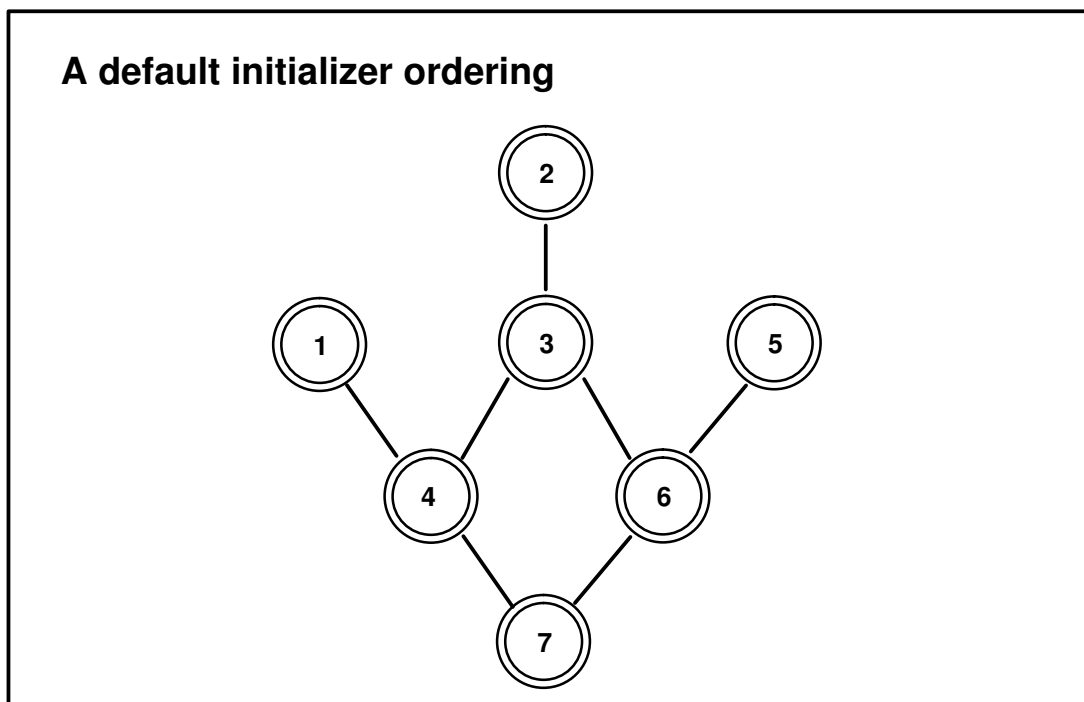
Figure 10. A default initializer ordering of a class's inheritance hierarchy.

In this example, the **somInitCtrl** data structure for class **7** is what tells node **6** in Figure 10 not to invoke node **3**'s initializer code (because it has already been executed). The code that deals with the **somInitCtrl** data structure is generated automatically within the implementation bindings for a class, and need not concern a class implementor.

As illustrated by this example, when an instance of a given class (or some descendant class) is initialized, only one of the given class's initializers will be executed, and this will happen exactly once (under control of the ordering determined by the class of the object being initialized).

The **somInitCtrl** structure solves a problem originally created by the addition of multiple inheritance to SOMobjects 2.0. With multiple inheritance, any class can appear at the top of a multiple inheritance diamond. Previously, whenever this happened, the class could easily receive multiple initialization calls. In the current version of the SOMobjects Toolkit, however, the **somInitCtrl** structure prevents this from happening.

Declaring new initializers in SOM IDL

When defining SOMobjects classes, programmers can easily declare and implement new initializers. Classes can have as many initializers as desired, and subclassers can invoke whichever of these they want. When introducing new initializers, developers must adhere to the following rules:

- All initializer methods take a **somInitCtrl** data structure as an initial **inout** parameter (its type is defined in the SOMobjects header file somapi.h), and

- All initializers return **void**.

Accordingly, the **somDefaultInit** initializer introduced by **SOMObject** takes a **somInitCtrl** structure as its (only) argument, and returns **void**. Here is the IDL syntax for this method, as declared in somobj.idl:

> **void somDefaultInit (inout somInitCtrl** *ctrl***);**

When introducing a new initializer, it is also necessary to specify the **init** modifier in the **implementation** section. The **init** modifier is what tells emitters that the new method is actually an initializer, so the method can be properly supported from the language bindings. As described below, this support includes the generation of special initializer stub procedures in the implementation template file, as well as bindings containing ancestor-initialization macros and object constructors that invoke the class implementor's new initializers.

It is a good idea to begin the names of initializer methods with the name of the class (or some other string that can be unique for the class). This is important because all initializers available on a class must be newly introduced by that class (that is, you cannot override initializers — except for **somDefaultInit**). Using a class-unique name means that subclasses will not be unnecessarily constrained in their choice of initializer names.

Here are two classes that introduce new initializers:

```
interface Example1 : SOMObject
{
    void Example1_withName (inout somInitCtrl ctrl, in string name);
    void Example1_withSize (inout somInitCtrl ctrl, in long size);
    void Example1_withNandS(inout somInitCtrl ctrl, in string name,
                                                    in long size);

        implementation {
                releaseorder: Example1_withName,
                              Example1_withSize,
                              Example1_withNandS;
                somDefaultInit: override, init;
                somDestruct: override;
                Example1_withName: init;
                Example1_withSize: init;
                Example1_withNandS: init;
        };
};


interface Example2 : Example1
{
    void Example2_withName(inout somInitCtrl ctrl, in string name);
    void Example2_withSize(inout somInitCtrl ctrl, in long size);
        implementation {
                releaseorder: Example2_withName,
                              Example2_withSize;
                somDefaultInit: override, init;
                somDestruct: override;
                Example2_withName: init;
                Example2_withSize: init;
        };
};
```

Here, interface "Example1" declares three new initializers. Notice the use of **inout somInitCtrl** as the first argument of each initializer, and also note that the **init** modifier is used in the **implementation** section. These two things are required to declare initializers. Any number of initializers can be declared by a class. "Example2" declares two initializers.

"Example1" and "Example2" both override the **somDefaultInit** initializer. This initializer method is introduced by **SOMObject** and is special for two reasons: First, **somDefaultInit** is the *only* initializer that can be overridden. And, second, SOMobjects arranges that this initializer will always be available on any class (as further explained below).

Historically in the SOMobjects Toolkit, object-initialization methods by default have invoked the **somInit** method, which class implementors could override to customize initialization as appropriate. SOMobjects continues to support this approach, so that existing code (and class binaries) will execute correctly. However, the **somDefaultInit** method is now the preferred form of initialization because it offers greatly improved efficiency.

Even if no specialized initialization is needed for a class, you should still **override** the **somDefaultInit** method in the interest of efficiency. If you do not override **somDefaultInit**, then a generic (and therefore less efficient) **somDefaultInit** method procedure will be used for your class. This generic method procedure first invokes **somDefaultInit** on the appropriate ancestor classes. Then (for consistency with earlier versions of SOMobjects), it checks to determine if the class overrides **somInit** and, if so, calls any customized **somInit** code provided by the class.

When you override **somDefaultInit**, the emitter's implementation template file will include a stub procedure similar to those used for other initializers, and you can fill it in as appropriate (or simply leave it as is). Default initialization for your class will then run much faster than with the generic method procedure. Examples of initializer stub procedures (and customizations) are given below.

In summary, the initializers available for any class of objects are **somDefaultInit** (which you should always **override**) plus any new initializers explicitly declared by the class designer. Thus, "Example1" objects may be initialized using any of four different initializers (the three that are explicitly declared, plus **somDefaultInit**). Likewise, there are three initializers for the "Example2" objects. Some examples of using initializers are provided below.

Considerations re: 'somInit' initialization from earlier SOM releases

To re-emphasize: All code written prior to SOMobjects Release 2.1 using documented guidelines for the earlier initialization approach based on the **somInit** method (as well as all existing class binaries) continues to be fully supported and useful.

Prior to SOMobjects 2.1, initialization was done with initializer methods that would simply "chain" parent-method calls upward, thereby allowing the execution of initializer method procedures contributed by all ancestors of an object's class. This chaining of initializer calls was not supported in any special way by the SOM API. Parent-method calls are simply one of the possible idioms available to users of OOP in SOM, easily available to a SOM class designer as a result of the support provided by the SOMobjects Toolkit emitters for parent-method calls.

So, SOM did not constrain initialization to be done in any particular way or require the use of any particular ordering of the method procedures of ancestor classes. But, SOM did provide an overall framework that class designers could easily utilize in order to implement default initialization of SOM objects. This framework is provided by the **somInit** object-initialization method introduced by the **SOMObject** class and supported by the SOM Toolkit emitters. The emitters create an implementation template file with stub procedures for overridden methods that automatically chain parent-method calls upward through parent classes. Many of the class methods that perform object creation called **somInit** automatically. [Note: These will now call **somDefaultInit**, which in turn calls **somInit** for legacy code, as described in the previous topic.]

Because it takes no arguments, **somInit** best served the purpose of a default initializer. SOM programmers also had the option of introducing additional "non-default" initialization methods that took arguments. In addition, by using metaclasses, they could introduce new class methods as object constructors that first create an object (generally using **somNewNoInit**) and then invoke some non-default initializer on the new object.

For a number of reasons, the **somInit** framework has been augmented by recognizing *initializers* as a special kind of method in SOMobjects. One advantage of this approach is that special metaclasses are no longer required for defining constructors that take arguments. Instead, because the **init** modifier identifies initializers, usage-binding emitters can now provide these constructors. This results in simpler designs and more efficient programs.

Although **somDefaultInit** replaces **somInit** as the no-argument initializer used for SOM objects, all previous use of **somInit** is still supported by the SOMobjects Developers Toolkit on AIX, OS/2 and 16-bit Windows. You may continue to use **somInit** on these systems if you like, although this is somewhat less efficient than using **somDefaultInit**.

However, you *cannot* use both methods. In particular, if a class overrides both **somDefaultInit** and **somInit**, its **somInit** code will never be executed. It is recommended that you always override **somDefaultInit** for object initialization. For one thing, it is likely that when SOMobjects is ported to new systems, **somInit** (and **somUninit**) may not be supported on those systems. Thus, code written using these (obsolete) methods will be less portable.

Implementing initializers

When new initializers are introduced by a class, as in the preceding examples, the implementation template file generated by the SOM Toolkit C and C++ emitters automatically contains an appropriate stub procedure for each initializer method, for the class implementor's use. The body of an initializer stub procedure consists of two main sections:

- The first section performs calls to ancestors of the class to invoke their initializers.
- The second section is used by the programmer to perform any "local" initializations appropriate to the instance data of the class being defined.

In the first section, by default, the parents of the new class are the ancestors whose initializers are called. When something else is desired, the IDL **directinitclasses** modifier can be used to explicitly designate the ancestors whose initializer methods should be invoked by a new class's initializers.

Important: Under no circumstances can the number or the ordering of ancestor initializer calls in the first section of an initializer stub procedure be changed. The control masks used by initializers are based on these orderings. (If you want to change the number or ordering of ancestor initializer calls, you must use the **directinitclasses** modifier.) The ancestor initializer calls themselves can be modified as described below.

Each call to an ancestor initializer is made using a special macro (much like a parent call) that is defined for this purpose within the implementation bindings. These macros are defined for all possible ancestor initialization calls. Initially, an initializer stub procedure invokes the default ancestor initializers provided by **somDefaultInit**. However, a class implementor can replace any of these calls with a different initializer call, as long as it calls the same ancestor (see the example in the next topic). Non-default initializer calls generally take other arguments in addition to the control argument.

In the second section of an initializer stub procedure, the programmer provides any class-specific code that may be needed for initialization. For example, the "Example2_withName" stub procedure is shown below. As with all stub procedures produced by the SOMobjects implementation-template emitters, this code requires no modification to run correctly.

```
SOM_Scope void SOMLINK Example2_withName(Example2 *somSelf,
                                         Environment *ev,
                                         somInitCtrl* ctrl,
                                         string name)
{
    Example2Data *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    Example2MethodDebug("Example2","withName");

    /*
     * first section -- calls to ancestor initializers
     */
    Example2_BeginInitializer_Example2_withName;
    Example2_Init_Example1_somDefaultInit(somSelf, ctrl);
```

```
        /*
         * second section -- local Example2 initialization code
         */
    }
```

In this example, notice that the "Example2_withName" initializer is an IDL callstyle method, so it receives an **Environment** argument. In contrast, **somDefaultInit** is introduced by the **SOMObject** class (so it has an OIDL callstyle initializer, without an environment).

Important: If a class is defined where multiple initializers have exactly the same signature, then the C++ usage bindings will not be able to differentiate among them. That is, if there are multiple initializers defined with environment and long arguments, for example, then C++ clients would *not* be able to make a call using only the class name and arguments, such as:

```
    new Example2(env, 123);
```

Rather, C++ users would be forced to first invoke the **somNewNoInit** method on the class to create an uninitialized object, and then separately invoke the desired initializer method on the object. This call would pass a zero for the control argument, in addition to passing values for the other arguments. For further discussion of client usage, see "Using initializers when creating new objects" later in this chapter.

Selecting non-default ancestor initializer calls

Often, it will be appropriate (in the first section of an initializer stub procedure) to change the invocation of an ancestor's **somDefaultInit** initializer to some other initializer available on the same class. The rule for making this change is simple: Replace **somDefaultInit** with the name of the desired ancestor initializer, and add any new arguments that are required by the replacement initializer. Important: Under no circumstances can you change anything else in the first section.

This example shows how to change an ancestor-initializer call correctly. Since there is a known "Example1_withName" initializer, the following default ancestor-initializer call (produced within the stub procedure for "Example2_withName") can be changed from

```
    Example2_Init_Example1_somDefaultInit(somSelf, ctrl);
```

to

```
    Example2_Init_Example1_Example1_withName(somSelf, ev, ctrl, name);
```

Notice that the revised ancestor-initializer call includes arguments for an **Environment** and a name, as defined by the "Example1_withname" initializer.

Using initializers when creating new objects

There are several ways that client programs can take advantage of the **somDefaultInit** object initialization. If desired, clients can use the SOM API directly (rather than taking advantage of the usage bindings). Also, the general object constructor, **somNew**, can always be invoked on a class to create and initialize objects. This call creates a new object and then invokes **somDefaultInit** on it.

To use the SOM API directly, the client code should first invoke the **somNewNoInit** method on the desired class object to create a new, uninitialized object. Then, the desired initializer is invoked on the new object, passing a null (that is, 0) control argument in addition to whatever other arguments may be required by the initializer. For example:

```
    /* first make sure the Example2 class object exists */
    Example2NewClass(Example2_MajorVersion, Example2_MinorVersion);

    /* then create a new, uninitialized Example2 object */
    myObject = _somNewNoInit(_Example2);

    /* then initialize it with the desired initializer */
    Example2_withName(myObject, env, 0, "MyName");
```

Usage bindings hide the details associated with initializer use in various ways and make calls more convenient for the client. For example, the C usage bindings for any given class already provide a convenience macro, *<className>***New**, that first assures existence of the class object, and then calls **somNew** on it to create and initialize a new object. As explained above, **somNew** will use **somDefaultInit** to initialize the new object.

Also, the C usage bindings provide object-construction macros that use **somNewNoInit** and then invoke non-default initializers. These macros are named using the form *<class-Name>***New**_*<initializerName>*. For example, the C usage bindings for "Example2" allow using the following expression to create, initialize, and return a new "Example2" object:

```
Example2New_Example2_withName(env, "AnyName");
```

In the C++ bindings, initializers are represented as overloaded C++ constructors. As a result, there is no need to specify the name of the initializer method. For example, using the C++ bindings, the following expressions could be used to create a new "Example2" object:

```
new Example2;                     // will use somDefaultInit
new Example2();                   // will use somDefaultInit
new Example2(env,"A.B.Normal");   // will use Example2_withName
new Example2(env,123);            // will use Example2_withSize
```

Observe that if multiple initializers in a class have exactly the same signatures, the C++ usage bindings would be unable to differentiate among the calls, if made using the forms illustrated above. In this case, a client could use **somNewNoInit** first, and then invoke the specific initializer, as described in the preceding paragraphs.

Uninitialization

An object should always be uninitialized before its storage is freed. This is important because it also allows releasing resources and freeing storage not contained within the body of the object. SOMobjects handles uninitialization in much the same way as for initializers: An uninitializer takes a control argument and is supported with stub procedures in the implementation template file in a manner similar to initializers.

Only a single uninitialization method is needed, so **SOMObject** introduces the method that provides this function: **somDestruct**. As with the default initializer method, a class designer who requires nothing special in the way of uninitialization need not be concerned about modifying the default **somDestruct** method procedure. However, your code will execute faster if the .idl file overrides **somDestruct** so that a non-generic stub-procedure code can be provided for the class. Note that **somDestruct** was overridden by "Example1" and "Example2" above. No specific IDL modifiers other than **override** are required for this.

Like an initializer template, the stub procedure for **somDestruct** consists of two sections: The first section is used by the programmer for performing any "local" uninitialization that may be required. The second section (which consists of a single **EndDestructor** macro invocation) invokes **somDestruct** on ancestors. The second section must *not* be modified or removed by the programmer. It must be the final statement executed in the destructor.

Using 'somDestruct'

It is rarely necessary to invoke the **somDestruct** method explicitly. This is because object uninitialization is normally done just before freeing an object's storage, and the mechanisms provided by SOMobjects for this purpose will automatically invoke **somDestruct**. For example, if an object were created using **somNew** or **somNewNoInit**, or by using a convenience macro provided by the C language bindings, then the **somFree** method can be invoked on the object to delete the object. This automatically calls **somDestruct** before freeing storage.

C++ users can simply use the **delete** operator provided by the C++ bindings. This destructor calls **somDestruct** before the C++ **delete** operator frees the object's storage.

On the other hand, if an object is initially created by allocating memory in some special way and subsequently some **somRenew** methods are used, **somFree** (or C++ **delete**) is probably not appropriate. Thus, the **somDestruct** method should be explicitly called to uninitialize the object before freeing memory.

A complete example

The following example illustrates the implementation and use of initializers and destructors from the C++ bindings. The first part shows the IDL for three classes with initializers. For variety, some of the classes use callstyle OIDL and others use callstyle IDL.

```
#include <somobj.idl>

interface A : SOMObject {
        readonly attribute long a;
        implementation {
                releaseorder: _get_a;
                functionprefix = A;
                somDefaultInit: override, init;
                somDestruct: override;
                somPrintSelf: override;
        };
};


interface B : SOMObject {
        readonly attribute long b;
        void BwithInitialValue(inout somInitCtrl ctrl,
                               in long initialValue);
        implementation {
                callstyle = OIDL;
                releaseorder: _get_b, BwithInitialValue;
                functionprefix = B;
                BwithInitialValue: init;
                somDefaultInit: override, init;
                somDestruct: override;
                somPrintSelf: override;
        };
};


interface C : A, B      {
        readonly attribute long c;
        void CwithInitialValue(inout somInitCtrl ctrl,
                               in long initialValue);
        void CwithInitialString(inout somInitCtrl ctrl,
                                in string initialString);
        implementation {
                releaseorder: _get_c, CwithInitialString,
                              CwithInitialValue;
                functionprefix = C;
                CwithInitialString: init;
                CwithInitialValue: init;
                somDefaultInit: override, init;
                somDestruct: override;
                somPrintSelf: override;
        };
};
```

Implementation code

Based on the foregoing class definitions, the next example illustrates several important aspects of initializers. The following code is a completed implementation template and an example client program for the preceding classes. Code added to the original template is given in bold.

```c
/*
 *  This file generated by the SOM Compiler and Emitter Framework.
 *  Generated using:
 *      SOM Emitter emitxtm.dll: 2.22
 */

#define SOM_Module_ctorfullexample_Source
#define VARIABLE_MACROS
#define METHOD_MACROS
#include <ctorFullExample.xih>
#include <stdio.h>

SOM_Scope void SOMLINK AsomDefaultInit(A *somSelf,
                                       somInitCtrl* ctrl)
{
    AData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    AMethodDebug("A","somDefaultInit");

    A_BeginInitializer_somDefaultInit;
    A_Init_SOMObject_somDefaultInit(somSelf, ctrl);
    /*
     * local A initialization code added by programmer
     */
    _a = 1;
}

SOM_Scope void SOMLINK AsomDestruct(A *somSelf, octet doFree,
                                    somDestructCtrl* ctrl)
{
    AData *somThis; /* set by BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    AMethodDebug("A","somDestruct");
    A_BeginDestructor;

    /*
     * local A deinitialization code added by programmer
     */
    A_EndDestructor;
}

SOM_Scope SOMObject*  SOMLINK AsomPrintSelf(A *somSelf)
{
    AData *somThis = AGetData(somSelf);
    AMethodDebug("A","somPrintSelf");

    somPrintf("{an instance of %s at location %X with (a=%d)}\n",
              _somGetClassName(),somSelf,__get_a((Environment*)0));
    return (SOMObject*)((void*)somSelf);
}
```

```
SOM_Scope void SOMLINK BBwithInitialValue(B *somSelf,
                                          somInitCtrl* ctrl,
                                          long initialValue)
{
    BData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    BMethodDebug("B","BwithInitialValue");

    B_BeginInitializer_withInitialValue;
    B_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
     * local B initialization code added by programmer
     */
    _b = initialValue;
}

SOM_Scope void SOMLINK BsomDefaultInit(B *somSelf,
                                       somInitCtrl* ctrl)
{
    BData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    BMethodDebug("B","somDefaultInit");

    B_BeginInitializer_somDefaultInit;
    B_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
     * local B initialization code added by programmer
     */
    _b = 2;
}

SOM_Scope void SOMLINK BsomDestruct(B *somSelf, octet doFree,
                                    somDestructCtrl* ctrl)
{
    BData *somThis; /* set by BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    BMethodDebug("B","somDestruct");
    B_BeginDestructor;

    /*
     * local B deinitialization code added by programmer
     */

    B_EndDestructor;
}

SOM_Scope SOMObject*  SOMLINK BsomPrintSelf(B *somSelf)
{
    BData *somThis = BGetData(somSelf);
    BMethodDebug("B","somPrintSelf");

    printf("{an instance of %s at location %X with (b=%d)}\n",
           _somGetClassName(),somSelf,__get_b());
    return (SOMObject*)((void*)somSelf);
}
```

Note: The following initializer for a C object accepts a string as an argument, converts this to an integer, and uses this for ancestor initialization of "B." This illustrates how a default ancestor initializer call is replaced with a non-default ancestor initializer call.

```
SOM_Scope void SOMLINK CCwithInitialString(C *somSelf,
                                           Environment *ev,
                                           somInitCtrl* ctrl,
                                           string initialString)
{
    CData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CMethodDebug("C","CwithInitialString");

    C_BeginInitializer_withInitialString;
    C_Init_A_somDefaultInit(somSelf, ctrl);
    C_Init_B_BwithInitialValue(somSelf, ctrl,
                            atoi(initialString)-11);

    /*
     * local C initialization code added by programmer
     */
    _c = atoi(initialString);
}

SOM_Scope void SOMLINK CsomDefaultInit(C *somSelf,
                                       somInitCtrl* ctrl)
{
    CData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CMethodDebug("C","somDefaultInit");

    C_BeginInitializer_somDefaultInit;
    C_Init_A_somDefaultInit(somSelf, ctrl);
    C_Init_B_somDefaultInit(somSelf, ctrl);

    /*
     * local C initialization code added by programmer
     */
    _c = 3;
}

SOM_Scope void SOMLINK CsomDestruct(C *somSelf, octet doFree,
                                    somDestructCtrl* ctrl)
{
    CData *somThis; /* set by BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    CMethodDebug("C","somDestruct");
    C_BeginDestructor;

    /*
     * local C deinitialization code added by programmer
     */

    C_EndDestructor;
}
```

```
SOM_Scope SOMObject*  SOMLINK CsomPrintSelf(C *somSelf)
{
    CData *somThis = CGetData(somSelf);
    CMethodDebug("C","somPrintSelf");

    printf("{an instance of %s at location %X with"
           " (a=%d, b=%d, c=%d)}\n",
        _somGetClassName(),somSelf,
        __get_a((Environment*)0),
        __get_b(),
        __get_c((Environment*)0));
    return (SOMObject*)((void*)somSelf);
}

SOM_Scope void SOMLINK CCwithInitialValue(C *somSelf,
                                          Environment *ev,
                                          somInitCtrl* ctrl,
                                          long initialValue)
{
    CData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CMethodDebug("C","CwithInitialValue");

    C_BeginInitializer_withInitialValue;
    C_Init_A_somDefaultInit(somSelf, ctrl);
    C_Init_B_BwithInitialValue(somSelf, ctrl, initialValue-11);

    /*
     * local C initialization code added by programmer
     */
    _c = initialValue;
}
```

Here is a C++ program that creates instances of "A", "B", and "C" using the initializers defined above.

```
main()
{
    SOM_TraceLevel = 1;

    A *a = new A;
    a->somPrintSelf();
    delete a;
    printf("\n");

    B *b = new B();
    b->somPrintSelf();
    delete b;
    printf("\n");

    b = new B(22);
    b->somPrintSelf();
    delete b;
    printf("\n");

    C *c = new C;
    c->somPrintSelf();
    delete c;
    printf("\n");

    c = new C((Environment*)0, 44);
    c->somPrintSelf();
    delete c;
    printf("\n");
```

```
        c = new C((Environment*)0, "66");
        c->somPrintSelf();
        delete c;
}
```

The output from the preceding program is as follows:

```
"ctorFullExample.C": 18:         In A:somDefaultInit
"ctorFullExample.C": 48:         In A:somPrintSelf
"./ctorFullExample.xih": 292:    In A:A_get_a
{an instance of A at location 20063C38 with (a=1)}
"ctorFullExample.C": 35:         In A:somDestruct


"ctorFullExample.C": 79:         In B:somDefaultInit
"ctorFullExample.C": 110:        In B:somPrintSelf
"./ctorFullExample.xih": 655:    In B:B_get_b
{an instance of B at location 20064578 with (b=2)}
"ctorFullExample.C": 97:         In B:somDestruct


"ctorFullExample.C": 62:         In B:BwithInitialValue
"ctorFullExample.C": 110:        In B:somPrintSelf
"./ctorFullExample.xih": 655:    In B:B_get_b
{an instance of B at location 20064578 with (b=22)}
"ctorFullExample.C": 97:         In B:somDestruct


"ctorFullExample.C": 150:        In C:somDefaultInit
"ctorFullExample.C": 18:         In A:somDefaultInit
"ctorFullExample.C": 79:         In B:somDefaultInit
"ctorFullExample.C": 182:        In C:somPrintSelf
"./ctorFullExample.xih": 292:    In A:A_get_a
"./ctorFullExample.xih": 655:    In B:B_get_b
"./ctorFullExample.xih": 1104:   In C:C_get_c
{an instance of C at location 20065448 with (a=1, b=2, c=3)}
"ctorFullExample.C": 169:        In C:somDestruct
"ctorFullExample.C": 35:         In A:somDestruct
"ctorFullExample.C": 97:         In B:somDestruct


"ctorFullExample.C": 196:        In C:CwithInitialValue
"ctorFullExample.C": 18:         In A:somDefaultInit
"ctorFullExample.C": 62:         In B:BwithInitialValue
"ctorFullExample.C": 182:        In C:somPrintSelf
"./ctorFullExample.xih": 292:    In A:A_get_a
"./ctorFullExample.xih": 655:    In B:B_get_b
"./ctorFullExample.xih": 1104:   In C:C_get_c
{an instance of C at location 20065448 with (a=1, b=33, c=44)}
"ctorFullExample.C": 169:        In C:somDestruct
"ctorFullExample.C": 35:         In A:somDestruct
"ctorFullExample.C": 97:         In B:somDestruct



"ctorFullExample.C": 132:        In C:CwithInitialString
"ctorFullExample.C": 18:         In A:somDefaultInit
"ctorFullExample.C": 62:         In B:BwithInitialValue
"ctorFullExample.C": 182:        In C:somPrintSelf
"./ctorFullExample.xih": 292:    In A:A_get_a
"./ctorFullExample.xih": 655:    In B:B_get_b
"./ctorFullExample.xih": 1104:   In C:C_get_c
{an instance of C at location 20065448 with (a=1, b=55, c=66)}
"ctorFullExample.C": 169:        In C:somDestruct
"ctorFullExample.C": 35:         In A:somDestruct
"ctorFullExample.C": 97:         In B:somDestruct
```

Customizing the initialization of class objects

As described previously, the **somDefaultInit** method can be overridden to customize the initialization of objects. Because classes are objects, **somDefaultInit** is also invoked on classes when they are first created (generally by invoking the **somNew** method on a meta-class). For a class object, however, **somDefaultInit** normally just sets the name of the class to "unknown," after which the **somInitMIClass** method must be used for the major portion of class initialization. Of course, metaclasses can override **somDefaultInit** to initialize introduced class variables that require no arguments for their initialization.

Note: Because **somNew** does not call **somInitMIClass**, class objects returned from invocations of **somNew** on a metaclass are not yet useful class objects.

The **somInitMIClass** method (introduced by **SOMClass**) is invoked on a new class object using arguments to indicate the class name and the parent classes from which inheritance is desired (among other arguments). This invocation is made by whatever routine is used to initialize the class. (For SOM classes using the C or C++ implementation bindings, this is handled by the **somBuildClass** procedure, which is called by the implementation bindings automatically.) The **somInitMIClass** method is often overridden by a metaclass to influence initialization of new classes in some way. Typically, the overriding procedure begins by making parent method calls, and then performs additional actions thereafter.

However, without use of the Cooperative Metaclass to guarantee correct operation of SOMobjects in general, none of the methods introduced by **SOMClass** should be overridden. As a result, customizing the initialization of class objects (other than through overriding **somDefaultInit** for initialization of class variables) is not recommended in SOMobjects 2.1. Users whose applications require this should request access to the experimental Cooperative Metaclass used to implement the SOMobjects Metaclass Framework. But, metaclasses implemented using the experimental Cooperative Metaclass may require reprogramming when SOMobjects introduces an officially supported Cooperative Metaclass.

5.6 Creating a SOM Class Library

One of the principal advantages of SOM is that it makes "black box" (or binary) reusability possible. Consequently, SOM classes are frequently packaged and distributed as class libraries. A *class library* holds the actual <u>implementation of one or more classes</u> and can be <u>dynamically loaded and unloaded as needed</u> by applications. Importantly, class libraries can also be <u>replaced independently of the applications</u> that use them and, provided that the class implementor observes simple SOM guidelines for preserving binary compatibility, can <u>evolve and expand</u> over time.

General guidelines for class library designers

One of the most important features of SOM is that it allows you to build and distribute class libraries in binary form. Because there is no "fragile base class" problem in SOM, client programs that use your libraries (by subclassing your classes or by invoking the methods in your classes) will not need to be recompiled if you later produce a subsequent version of the library, provided you adhere to some simple restrictions.

1. You should always maintain the syntax and the semantics of your existing interfaces. This means that you cannot take away any exposed capabilities, nor add or remove arguments for any of your public methods.

2. Always maintain the **releaseorder** list, so that it never changes except for additions to the end. The **releaseorder** should contain all of your public methods, the one or two methods that correspond to each public attribute, and a placeholder for each private method (or private attribute method).

3. Assign a higher **minorversion** number for each subsequent release of a class, so that client programmers can determine whether a new feature is present or not. Change the **majorversion** number only when you deliberately wish to break binary compatibility. (See the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler" for explanations of the **releaseorder, minorversion** and **majorversion** modifiers.)

4. Under Windows, you should avoid the use of methods or attributes that return structures. In the DOS/Windows environment, there is no universally agreed upon calling convention for returning structures that is observed by all popular language compilers. Instead, define attributes that return pointers to structures, or define methods that take an **out** parameter for passing a structure back to the caller.

   Note that you can always avoid this problem in classes of your own design. However, some of the attributes and methods in the frameworks that come with the SOMobjects Toolkit *do* return structures. Many of these are dictated by the OMG CORBA standard, and could not be avoided. For each of these methods two common calling conventions have been implemented:

   - Microsoft convention, where the address of the structure is returned in AX:DX, and
   - Borland convention, where the caller provides a hidden first argument to receive a copy of the returned structure.

   No action is needed on your part to use the Microsoft convention. To use the Borland convention, you should include the file BCCSTRUC.H following any other "includes" of SOM headers.

5. With each new release of your class library, you have significant degrees of freedom to change much of the implementation detail. You can add to or reorganize your instance variables, add new public or private methods, inject new base classes into your class hierarchies, change metaclasses to more derived ones, and relocate the implementation of methods upward in your class hierarchies. Provided you always retain the same capabilities and semantics that were present in your first release, none of these changes will break the client programs that use your libraries.

Types of class libraries

Since class libraries are not programs, users cannot execute them directly. To enable users to make direct use of your classes, you must also provide one or more programs that create the classes and objects that the user will need. This section describes how to package your classes in a SOM class library and what you must do to make the contents of the library accessible to other programs.

On AIX, class libraries are actually produced as AIX shared libraries, whereas on OS/2 or Windows they appear as dynamically-linked libraries (or DLLs). The term "DLL" is sometimes used to refer to either an AIX, an OS/2, or a Windows class library, and (by convention only) the file suffix ".dll" is used for SOM class libraries on all platforms.

A program can use a class library containing a given class or classes in one of two ways:

1. If the programmer employs the SOM bindings to instantiate the class and invoke its methods, the resulting client program contains static references to the class. The operating system will automatically resolve those references when the program is loaded, by also loading the appropriate class library.

2. If the programmer uses only the dynamic SOM mechanisms for finding the class and invoking its methods (for example, by invoking **somFindClass, somFindMethod, somLookupMethod, somDispatch, somResolveByName**, and so forth), the resulting client program does not contain any static references to the class library. Thus, SOM will load the class library dynamically during execution of the program. Note: For SOM to be able to load the class library, the **dllname** modifier must be set in the .idl file. (See the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler.")

Because the provider of a class library cannot predict which of these ways a class will be used, SOM class libraries must be built such that either usage is possible. The first case above requires the class library to export the entry points needed by the SOM bindings, whereas the second case requires the library to provide an initialization function to create the classes it contains. The following topics discuss each case.

Building export files

The SOM Compiler provides an "exp" emitter for AIX and a "def" emitter for OS/2 or Windows to produce the necessary exported symbols for each class. For example, to generate the necessary exports for a class "A", issue the **sc** or **somc** command with one of the following **–s** options. (For a discussion of the **sc** or **somc** command and options, see "Running the SOM Compiler" in Chapter 4, "SOM IDL and the SOM Compiler.")

For AIX, this command generates an "a.exp" file:

```
sc –sexp a.idl
```

For OS/2, this command generates an "a.def" file:

```
sc –sdef a.idl
```

For Windows, this command generates an "a.def" file:

```
somc –sdef a.idl
```

Typically, a class library contains multiple classes. To produce an appropriate export file for each class that the library will contain, you can create a new export file for the library itself by combining the exports from each "exp" or "def" file into a single file. Following are examples of a combined export "exp" file for AIX and a combined "def" file for OS/2 or Windows. Each example illustrates a class library composed of three classes, "A", "B", and "C".

AIX "exp" file:

```
#! abc.dll
ACClassData
AClassData
ANewClass
BCClassData
BClassData
BNewClass
CCClassData
CClassData
CNewClass
```

OS/2 "def" file:

```
LIBRARY abc INITINSTANCE
DESCRIPTION 'abc example class library'
PROTMODE
DATA MULTIPLE NONSHARED LOADONCALL
EXPORTS
   ACClassData
   AClassData
   ANewClass
   BCClassData
   BClassData
   BNewClass
   CCClassData
   CClassData
   CNewClass
```

Windows "def" file:

```
LIBRARY abc
DESCRIPTION 'abc example class library'
EXETYPE WINDOWS
STUB 'WINSTUB.EXE'
PROTMODE
DATA MOVEABLE SINGLE SHARED LOADONCALL
CODE MOVEABLE DISCARDABLE
HEAPSIZE 4096
STACKSIZE 4096
EXPORTS
   _ACClassData
   _AClassData
   _ANewClass
   _BCClassData
   _BClassData
   _BNewClass
   _CCClassData
   _CClassData
   _CNewClass
```

Other symbols in addition to those generated by the "def" or "exp" emitter can be included if needed, but this is not required by SOM. One feature of SOM is that a class library needs no more than three exports per class (by contrast, many OOP systems require externals for every method as well). One required export is the name of a procedure to create the class (*<class-Name>***NewClass**), and the others are two external data structures that are referenced by the SOM bindings.

Specifying the initialization function

An initialization function for the class library must be provided to support dynamic loading of the library by the SOM Class Manager. The SOM Class Manager expects that, whenever it loads a class library, the initialization function will create and register class objects for all of the classes contained in the library. These classes are then managed as a group (called an *affinity group*).

One class in the affinity group has a privileged position — namely, the class that was specifical-ly requested when the library was loaded. If that class (that is, the class that caused loading to occur) is subsequently unregistered, the SOM Class Manager will automatically unregister all of the other classes in the affinity group as well, and will unload the class library. Similarly, if the SOM Class Manager is explicitly asked to unload the class library, it will also automatically unregister and free all of the classes in the affinity group.

It is the responsibility of the class-library creator to supply the initialization function. The interface to the initialization function is given by the following C/C++ prototype:

#ifdef __IBMC__
  #pragma linkage (SOMInitModule, system)
#endif

SOMEXTERN void SOMLINK SOMInitModule (long *majorVersion***,**
                                        long *minorVersion***,**
                                        string *className***);**

The parameters provided to this function are the *className* and the major/minor version numbers of the class that was requested when the library was loaded (that is, the class that caused loading). The initialization function is free to use or to disregard this information; nevertheless, if it fails to create a class object with the required name, the SOM Class Manager considers the load to have failed. As a rule of thumb, however, if the initialization function invokes a <*className*>**NewClass** procedure for each class in the class library, this condition will always be met. Consequently, the parameters supplied to the initialization function are not needed in most cases.

Here is a typical class-library initialization function, written in C, for a library with three classes ("A", "B", and "C"):

```
#include "a.h"
#include "b.h"
#include "c.h"
#ifdef __IBMC__
   #pragma linkage (SOMInitModule, system)
#endif

SOMEXTERN void  SOMLINK SOMInitModule (long majorVersion,
                            long minorVersion, string className)
{
    SOM_IgnoreWarning (majorVersion);  /* This function makes  */
    SOM_IgnoreWarning (minorVersion);  /* no use of the passed */
    SOM_IgnoreWarning (className);     /* arguments.   */
    ANewClass (A_MajorVersion, A_MinorVersion);
    BNewClass (B_MajorVersion, B_MinorVersion);
    CNewClass (C_MajorVersion, C_MinorVersion);
}
```

The source code for the initialization function can be added to one of the implementation files for the classes in the library, or you can put it in a separate file and compile it independently.

Using Windows class libraries

Some additional considerations apply for Windows class libraries: Each class library must also supply a Windows LibMain function. The LibMain function is invoked automatically whenever a Windows DLL is loaded, and is responsible for identifying the library and its **SOMInitModule** function to the SOM Kernel. Here is an example of a typical Windows LibMain function for a SOM class library as it would appear in a C or C++ program:

```
#include <som.h>
SOMEXTERN void SOMLINK SOMInitModule (long majorVersion,
                                      long minorVersion,
                                      string className);
```

```
#include <windows.h>
int CALLBACK LibMain (HINSTANCE inst,
                      WORD ds,
                      WORD Heapsize,
                      LPSTR cmdLine)
{
    SOM_IgnoreWarning (inst);
    SOM_ignoreWarning (ds);
    SOM_IgnoreWarning (heapSize);
    SOM_IgnoreWarning (cmdLine);

    SOM_ClassLibrary ("xyz.dll");
    return 1;  /* Indicate success to loader */
}
```

The only operative statement in the LibMain function is the macro **SOM_ClassLibrary**, which identifies the actual file name of the library as it would appear in a Windows LoadLibrary call, and internally generates a reference to the library's **SOMInitModule** function. This information is passed to the SOM Kernel, which in turn registers the library and schedules the execution of the **SOMInitModule** function.

Typically, the SOM Kernel invokes the **SOMInitModule** function of each statically loaded class library during execution of the **SOM_MainProgram** macro within the using application; otherwise, **SOMInitModule** is invoked immediately upon completion of the dynamic loading of a class library by an already executing application. Regardless of the loading mechanism, the SOM Kernel guarantees that the **SOMInitModule** function executes exactly once for each class library.

Creating the import library

Finally, for each of your class libraries, you should create an import library that can be used by client programs (or by other class libraries that use your classes) to resolve the references to your classes.

Here is an example illustrating all of the steps required to create a class library ("abc.dll") that contains the three classes "A", "B", and "C".

1. Compile all of the implementation files for the classes that will be included in the library. Include the initialization function also.

   For AIX written in **C**:

   ```
   xlc -I. -I$SOMBASE/include -c a.c
   xlc -I. -I$SOMBASE/include -c b.c
   xlc -I. -I$SOMBASE/include -c c.c
   xlc -I. -I$SOMBASE/include -c initfunc.c
   ```

   For AIX written in **C++**:

   ```
   xlC -I. -I$SOMBASE/include -c a.C
   xlC -I. -I$SOMBASE/include -c b.C
   xlC -I. -I$SOMBASE/include -c c.C
   xlC -I. -I$SOMBASE/include -c initfunc.C
   ```

   For OS/2 written in **C**:

   ```
   icc -I. -I%SOMBASE%\include -Ge- -c a.c
   icc -I. -I%SOMBASE%\include -Ge- -c b.c
   icc -I. -I%SOMBASE%\include -Ge- -c c.c
   icc -I. -I%SOMBASE%\include -Ge- -c initfunc.c
   ```

   Note: The "–Ge" option is not used in all the compilers. It indicates that the object files will go into a DLL.

   For OS/2 written in **C++**:

   ```
   icc -I. -I%SOMBASE%\include -Ge- -c a.cpp
   icc -I. -I%SOMBASE%\include -Ge- -c b.cpp
   icc -I. -I%SOMBASE%\include -Ge- -c c.cpp
   icc -I. -I%SOMBASE%\include -Ge- -c initfunc.cpp
   ```

Note: The "–Ge" option is not used by all the compilers. It indicates that the object files will go into a DLL.

For Windows, written in **C**:

```
cl –ALw –G2s –Zp1 –I. –I%SOMBASE%\include –c a.c
cl –ALw –G2s –Zp1 –I. –I%SOMBASE%\include –c b.c
cl –ALw –G2s –Zp1 –I. –I%SOMBASE%\include –c c.c
cl –ALw –G2s –Zp1 –I. –I%SOMBASE%\include –c initfunc.c
```

Note: If your Windows program uses floats or doubles, also use the "–Oi" flag.

For Windows, written in **C++**:

```
cl –ALfu –G2s –Zp1 –I. –I%SOMBASE%\include –c a.cpp
cl –ALfu –G2s –Zp1 –I. –I%SOMBASE%\include –c b.cpp
cl –ALfu –G2s –Zp1 –I. –I%SOMBASE%\include –c c.cpp
cl –ALfu –G2s –Zp1 –I. –I%SOMBASE%\include –c initfunc.cpp
```

Note: If your Windows program uses floats or doubles, also use the "–Oi" flag.

2. Produce an export file for each class.

For AIX:

```
sc –sexp a.idl b.idl c.idl
```

For OS/2:

```
sc –sdef a.idl b.idl c.idl
```

For Windows:

```
somc –sdef a.idl b.idl c.idl
```

3. Manually combine the exported symbols into a single file.

For AIX, create a file "abc.exp" from "a.exp", "b.exp", and "c.exp". Do *not* include the initialization function (**SOMInitModule**) in the export list.

For OS/2, create a file "abc.def" from "a.def", "b.def", and "c.def". Include the initialization function (**SOMInitModule**) in the export list so that all classes will be initialized automatically, unless your initialization function does not need arguments and you explicitly invoke it yourself from an OS/2 DLL initialization routine.

For Windows, create a file "abc.def" from "a.def", "b.def", and "c.def". There is no need to include the **SOMInitModule** function in the export list for Windows programs.

4. Using the object files and the export file, produce a binary class library.

For AIX:

```
ld –o abc.dll –bE:abc.exp –e SOMInitModule –H512 –T512 \
    a.o b.o c.o initfunc.o –lc –L$SOMBASE/lib –lsomtk
```

The **–o** option assigns a name to the class library ("abc.dll"). The **–bE:** option designates the file with the appropriate export list. The **–e** option designates **SOMInitModule** as the initialization function. The **–H** and **–T** options must be supplied as shown; they specify the necessary alignment information for the text and data portions of your code. The **–l** options name the specific libraries needed by your classes. If your classes make use of classes in other class libraries, include a **–l** option for each of these also. The **ld** command looks for a library named "lib<*x*>.a", where <*x*> is the name provided with each **–l** option. The **–L** option specifies the directory where the "somtk" library resides.

For OS/2:

```
set LIB=%SOMBASE%\lib;%LIB%
link386 /noi /packd /packc /align:16 /exepack \
    a.obj b.obj c.obj initfunc.obj, abc.dll,,os2386 somtk, \
    abc.def
```

If your classes make use of classes in other class libraries, include the names of their import libraries immediately after "somtk" (before the next comma).

For Windows:

```
link a.obj b.obj c.obj initfunc.obj, abc.dll, nul, \
     ldllcew libw %SOMBASE%\somtk, abc.def
```

Note: /noi should not be used unless all symbols are exported as ordinals.

If your classes make use of classes in other class libraries, also include the names of their import libraries immediately after "somtk" (before the next comma).

5. Create an import library that corresponds to the class library, so that programs and other class libraries can use (import) your classes.

For AIX:

```
ar ruv libabc.a abc.exp        ◀ Note the use of the ".exp" file, not a ".o" file
```

The first filename ("libabc.a") specifies the name to give to the import library. It should be of the form "lib<x>.a", where <x> represents your class library. The second filename ("abc.exp") specifies the exported symbols to include in the import library.

Caution: Although AIX shared libraries can be placed directly into an archive file ("lib<x>.a"), this is not recommended! A SOM class library should have a corresponding import library constructed directly from the combined export file.

For OS/2:

```
implib /noi abc.lib abc.def
```

The first filename ("abc.lib") specifies the name for the import library and should always have a suffix of ".lib". The second filename ("abc.def") specifies the exported symbols to include in the import library. Note: **SOMInitModule** should be included in the <x>.dll but not in <x>.lib. If you are using an export file that contains the symbol **SOMInitModule**, delete it first; **SOMInitModule** should not appear in your import library because it needs not be exported. **SOMInitModule** should be included when creating your file <x>.dll because all classes in the <x>.dll will be initialized.

For Windows:

```
implib  abc.lib abc.def
```

The first filename ("abc.lib") specifies the name for the import library and should always have a suffix of ".lib". The second filename ("abc.def") specifies the exported symbols to include in the import library.

5.7 Customizing Memory Management

SOM is designed to be policy free and highly adaptable. Most of the SOM behavior can be customized by subclassing the built-in classes and overriding their methods, or by replacing selected functions in the SOM run-time library with application code. The next five sections contain more advanced topics describing how to customize the various aspects of SOM behavior: memory management, dynamic class loading and unloading, character output, error handling, and method resolution. Information on customizing Distributed SOM is provided in Chapter 6.

The memory management functions used by the SOM run-time environment are a subset of those supplied in the ANSI C standard library. They have the same calling interface and return the equivalent types of results as their ANSI C counterparts, but include some supplemental error checking. Errors detected in these functions result in the invocation of the error-handling function to which **SOMError** points.

The correspondence between the SOM memory-management function variables and their ANSI standard library equivalents is given in Table 1 below.

SOM Function Variable	ANSI Standard C Library Function	Return type	Argument types
SOMCalloc	**calloc( )**	**somToken**	**size_t, size_t**
SOMFree	**free( )**	**void**	**somToken**
SOMMalloc	**malloc( )**	**somToken**	**size_t**
SOMRealloc	**realloc( )**	**somToken**	**somToken, size_t**

Table 1. Memory-Management Functions

An application program can replace SOM's memory management functions with its own memory management functions to change the way SOM allocates memory (for example, to perform all memory allocations as suballocations in a shared memory heap). This replacement is possible because **SOMCalloc, SOMMalloc, SOMRealloc**, and **SOMFree** are actually *global variables* that point to SOM's default memory management functions, rather than being the names of the functions themselves. Thus, an application program can replace SOM's default memory management functions by assigning the entry-point address of the user-defined memory management function to the appropriate global variable. For example, to replace the default free procedure with the user-defined function **MyFree** (which must have the same signature as the ANSI C **free** function), an application program would require the following code:

```
#include <som.h>
/* Define a replacement routine: */
void myFree (somToken memPtr)
{
    (Customized code goes here)
}
...
SOMFree = MyFree;
```

Note: In general, all of these routines should be replaced as a group. For instance, if an application supplies a customized version of **SOMMalloc**, it should also supply corresponding **SOMCalloc**, **SOMFree**, and **SOMRealloc** functions that conform to this same style of memory management.

5.8 Customizing Class Loading and Unloading

SOM uses three routines that manage the loading and unloading of class libraries (referred to here as DLLs). These routines are called by the **SOMClassMgrObject** as it dynamically loads and registers classes. If appropriate, the rules that govern the loading and unloading of DLLs can be modified, by replacing these functions with alternative implementations.

Customizing class initialization

The **SOMClassInitFuncName** function has the following signature:

string (*SOMClassInitFuncName) ( );

This function returns the name of the function that will initialize (create class objects for) all of the classes that are packaged together in a single class library. (This function name applies to all class libraries loaded by the **SOMClassMgrObject**.) The SOM-supplied version of **SOM-ClassInitFuncName** returns the string **"SOMInitModule"**. The interface to the library initialization function is described under the topic "Creating a SOM Class Library" earlier in this chapter.

Customizing DLL loading

To dynamically load a SOM class, the **SOMClassMgrObject** calls the function pointed to by the global variable **SOMLoadModule** to load the DLL containing the class. The reason for making public the **SOMLoadModule** function (and the following **SOMDeleteModule** function) is to reveal the boundary where SOM touches the operating system. Explicit invocation of these functions is never required. However, they are provided to allow class implementors to insert their own code between the operating system and SOM, if desired. The **SOMLoadModule** function has the following signature:

long (*SOMLoadModule) (string *className***,**
                    string *fileName***,**
                    string *functionName***,**
                    long *majorVersion***,**
                    long *minorVersion***,**
                    somToken **modHandle***);**

This function is responsible for loading the DLL containing the SOM class *className* and returning either the value zero (for success) or a nonzero system-specific error code. The output argument *modHandle* is used to return a token that can subsequently be used by the DLL-unloading routine (described below) to unload the DLL. The default DLL-loading routine returns the DLL's *module handle* in this argument. The remaining arguments are used as follows:

Argument	Usage
fileName	— The file name of the DLL to be loaded, which can be either a simple name or a full path name.
functionName	— The name of the routine to be called after the DLL is successfully loaded by the **SOMClassMgrObject**. This routine is responsible for creating the class objects for the class(es) contained in the DLL. Typically, this argument has the value **"SOMInitModule"**, which is obtained from the function **SOMClassInitFuncName** described above. If no **SOMInitModule** entry exists in the DLL, the default DLL-loading routine looks in the DLL for a procedure with the name *<className>***NewClass** instead. If neither entry point can be found, the default DLL-loading routine fails.
majorVersion	— The major version number to be passed to the class initialization function in the DLL (specified by the *functionName* argument).

minorVersion	— The minor version number to be passed to the class initialization function in the DLL (specified by the *functionName* argument).

An application program can replace the default DLL-loading routine by assigning the entry point address of the new DLL-loading function (such as *MyLoadModule*) to the global variable **SOMLoadModule**, as follows:

```
#include <som.h>
/* Define a replacement routine: */
long myLoadModule (string className, string fileName,
                   string functionName, long majorVersion,
                   long minorVersion, somToken *modHandle)
{
    (Customized code goes here)
}
...
SOMLoadModule = MyLoadModule;
```

Customizing DLL unloading

To unload a SOM class, the **SOMClassMgrObject** calls the function pointed to by the global variable **SOMDeleteModule**. The **SOMDeleteModule** function has the following signature:

long (*SOMDeleteModule) (in somToken *modHandle***);**

This function is responsible for unloading the DLL designated by the *modHandle* parameter and returning either zero (for success) or a nonzero system-specific error code. The parameter *modHandle* contains the value returned by the DLL loading routine (described above) when the DLL was loaded.

An application program can replace the default DLL-unloading routine by assigning the entry point address of the new DLL-unloading function (such as, *MyDeleteModule*) to the global variable **SOMDeleteModule**, as follows:

```
#include <som.h>
/* Define a replacement routine: */
long myDeleteModule (somToken modHandle)
{
    (Customized code goes here)
}
...
SOMDeleteModule = MyDeleteModule;
```

5.9 Customizing Character Output

The SOM character-output function is invoked by all of the SOM error-handling and debugging macros whenever a character must be generated (see "Debugging" and "Exceptions and error handling" in Chapter 3, "Using SOM Classes in Client Programs"). The default character-output routine, pointed to by the global variable **SOMOutCharRoutine**, simply writes the character to "stdout," then returns 1 if successful, or 0 otherwise.

For convenience, **SOMOutCharRoutine** is supplemented by the **somSetOutChar** function. The **somSetOutChar** function enables each task to have a customized character output routine, thus it is often preferred for changing the output routine called by **somPrintf** (because **SOMOutCharRoutine** would remain in effect for subsequent tasks). On Windows, the **som-SetOutChar** function is required (rather than **SOMOutCharRoutine**); it is optional on other operating systems.

An application programmer might wish to supply a customized replacement routine to:

- Direct the output to **stderr,**

- Record the output in a log file,

- Collect characters and handle them in larger chunks,

- Send the output to a window to display it,

- Place the output in a clipboard, or

- Some combination of these.

With **SOMOutCharRoutine,** an application program would use code similar to the following to install the replacement routine:

```
#include <som.h>
#pragma linkage(myCharacterOutputRoutine, system)
/* Define a replacement routine: */
int SOMLINK myCharacterOutputRoutine (char c)
{
    (Customized code goes here)
}
...
/* After the next stmt all output */
/* will be sent to the new routine    */
SOMOutCharRoutine = myCharacterOutputRoutine;
```

With **somSetOutChar**, an application program would use code similar to the following to install the replacement routine:

```
#include <som.h>
static int irOutChar(char c);

static int irOutChar(char c)
{
    (Customized code goes here)
}

main (...)
{
    ...
    somSetOutChar((somTD_SOMOutCharRoutine *) irOutChar);
}
```

5.10 Customizing Error Handling

When an error occurs within any of the SOM-supplied methods or functions, an error-handling procedure is invoked. The default error-handling procedure supplied by SOM, pointed to by the global variable **SOMError**, has the following signature:

> **void (*SOMError) (int** *errorCode*, **string** *fileName*, **int** *lineNum***);**

The default error-handling procedure inspects the *errorCode* argument and takes appropriate action, depending on the last decimal digit of *errorCode* (see "Exceptions and error handling" in Chapter 3, "Using SOM Classes in Client Programs," for a discussion of error classifications). In the default error handler, fatal errors terminate the current process. The remaining two arguments (*fileName* and *lineNum*), which indicate the name of the file and the line number within the file where the error occurred, are used to produce an error message.

An application programmer might wish to replace the default error handler with a customized error-handling routine to:

- Record errors in a way appropriate to the particular application,

- Inform the user through the application's user interface,

- Attempt application-level recovery by restarting at a known point, or

- Shut down the application.

An application program would use code similar to the following to install the replacement routine:

```
#include <som.h>
/* Define a replacement routine: */
void myErrorHandler (int errorCode, string fileName, int lineNum)
{
    (Customized code goes here)
}
...
/* After the next stmt all errors    */
/* will be handled by the new routine */
SOMError = myErrorHandler;
```

When any error condition originates within the classes supplied with SOM, SOM is left in an internally consistent state. If appropriate, an application program can trap errors with a customized error-handling procedure and then resume with other processing. Application programmers should be aware, however, that all methods within the SOM run-time library behave *atomically.* That is, they either succeed or fail; but if they fail, partial effects are undone wherever possible. This is done so that all SOM methods remain usable and can be re-executed following an error.

5.11 Customizing Mutual Exclusion Services (Thread Safety)

The SOM kernel and the other SOMobjects frameworks (DSOM, Persistence, Replication, and so on), have been made *thread safe* with respect to multi-threaded processes. As used here, "thread safe" means that the SOMobjects run time has been implemented using *critical sections* to protect sections of the code which must only be executed by a single thread in a multi-threaded application process at one time. Critical sections in SOM are implemented using *mutual exclusion semaphores*, or "mutex semaphores," designed to block individual threads (as opposed to an entire process).

Some operating systems provide native multi-threading (for example, OS/2 and AIX 4.1). On other operating systems that do not support native multi-threading (such as, AIX 3.2), thread support may be provided as part of particular programming environments (like DCE) or libraries.

It is vital that SOM critical sections employ the mutex services that are provided by the thread package used by the application. Consequently, SOM provides a mechanism for defining and customizing mutex services.

Five mutex service functions are used to implement critical sections in SOM. These functions are called indirectly via the global pointer variables defined below. A **somToken** parameter (called "`sem`" below) is used as a generic "handle" to refer to a mutex semaphore — usually it is a pointer to a mutex semaphore variable or data structure. The actual representation of the mutex semaphore is hidden by the functions.

```
unsigned long (*SOMCreateMutexSem)(somToken *sem);
```

The referenced function creates a mutex semaphore, whose handle is returned as an output parameter in the **somToken** variable, "`sem`".

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMRequestMutexSem)(somToken sem);
```

The referenced function requests ownership of the mutex semaphore identified by the parameter, `sem`. If the semaphore is not currently owned by another thread, ownership is assigned to the calling thread. Otherwise, the calling thread is blocked until the semaphore is released by the current owner.

Important: If the same thread calls **SOMRequestMutexSem** multiple times, a reference count must be kept, so that the semaphore is released only after the same number of calls to **SOMReleaseMutexSem** . Some, but not all, thread packages provide reference counting automatically, via "counting semaphores."

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMReleaseMutexSem)(somToken sem);
```

The referenced function releases ownership of the mutex semaphore identified by the parameter, `sem`.

Important: If the same thread calls **SOMRequestMutexSem** multiple times, a reference count must be kept, so that the semaphore is released only after the same number of calls to **SOMReleaseMutexSem** . Some, but not all, thread packages provide reference counting automatically, via "counting semaphores."

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMDestroyMutexSem)(somToken sem);
```

The referenced function destroys the a mutex semaphore identified by the parameter, `sem`.

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMGetThreadId)();
```

The referenced function returns a small integer identifier for the calling thread. The ID cannot be associated with any other thread in the process. The ID is used as an index for table lookups.

If threads are not supported, the function must return 0.

The actual mutex service function prototypes and global variable declarations are found in file "somapi.h".

If the underlying operating system supports native multi-threading (for example, OS/2, AIX 4.1), SOM provides appropriate default mutex service function implementations. On those operating systems that do not support native multi-threading, the default mutex service functions have null implementations.

An application may want to use threading services different from those provided by the underlying operating system (if any); for example, DCE applications will want to use DCE threads. In that case, the default mutex service functions can be replaced by application-defined functions.

An application program would use code similar to the following to install the replacement routines:

```
#include <som.h>
/* Define a replacement routine: */
unsigned long myCreateMutexSem (somToken *sem)
{
     (Customized code goes here)
}
...
SOMCreateMutexSem = myCreateMutexSem;
```

It is important to install custom mutex service functions *before* any SOM calls are made.

Chapter 6. Distributed SOM (DSOM)

6.1 Introduction

Notice: The SOMobjects Base Toolkit provides the capability for implementing *Workstation DSOM* (distribution among processes on a single machine). Implementing an application that is distributed across a network of machines requires *Workgroup DSOM*, which is available *only* in the full-capability SOMobjects Developer Toolkit.

What is Distributed SOM?

Whereas the power of SOM technology comes from the fact that SOM insulates the client of an object from the object's *implementation*, the power of DSOM lies in the fact that DSOM insulates the client of an object from the object's *location*.

Distributed SOM (or DSOM) provides a framework that allows application programs to access objects across address spaces. That is, application programs can access objects in other processes, even on different machines. Both the location and implementation of an object are hidden from a client, and the client accesses the object (via method calls) in the same manner regardless of its location.

DSOM currently supports two types of distribution: distribution among processes on the same machine — referred to as *Workstation DSOM* — and distribution among a network of machines — referred to as *Workgroup DSOM*. DSOM runs on the AIX (Release 3.2 and above), OS/2 (Release 2.0 and above), and Windows (Release 3.1 and above) operating systems. A Workstation DSOM application can run on a machine in any environment using core capabilities of the SOMobjects system. Under the full-capability SOMobjects Developer Toolkit, Workgroup DSOM supports distribution across local area networks comprised of OS/2, AIX, and Windows systems. Future releases of DSOM may support large, enterprise-wide networks.

Support for TCP/IP and NetWare IPX/SPX is provided on AIX, OS/2, and Windows. NetBIOS support is provided for OS/2 and Windows. DSOM communications is extensible in that an application can provide its own transport (see Appendix C of the *SOMobjects Base Toolkit Users Guide*).

DSOM can be viewed in two ways:

- As an extension to the System Object Model, that allows a program to invoke methods on SOM objects in other processes, and

- As an Object Request Broker (ORB), that is, a standardized "transport" for distributed object interaction. In this respect, DSOM complies with the Common Object Request Broker Architecture (CORBA) specification, published by the Object Management Group (OMG) and x/Open ™.

This chapter describes DSOM from both perspectives.

DSOM features

Here is a quick summary of some of DSOM's more important features:

- Uses the standard SOM Compiler, Interface Repository, language bindings, and class libraries. DSOM provides a growth path for non-distributed SOM applications.

- Allows an application program to access a mix of local and remote objects. The fact that an object is remote is transparent to the program.

- Provides run-time services for creating, destroying, identifying. locating, and dispatching methods on remote objects. These services can be overridden or augmented to suit the application.

- Uses existing interprocess communication (IPC) facilities for Workstation communication, and common local area network (LAN) transport facilities for Workgroup communications. Support for TCP/IP, Netware IPX/SPX, and NetBIOS is provided. DSOM communications is extensible in that an application can provide its own transport (see Appendix C of this User's Guide).

- Provides support for writing <u>multi-threaded</u> (on OS/2 and AIX) and <u>event-driven</u> programs.
- Provides a <u>default object server program</u>, which can be easily used to create SOM objects and make those objects accessible to one or more client programs. If the default server program is used, SOM class libraries are loaded upon demand, so no server programming or compiling is necessary.
- <u>Complies with the CORBA 1.1 specification</u>, which is important for application portability.

When to use DSOM

DSOM should be used for those applications that require *sharing* of objects among multiple programs. The object actually exists in only one process (this process is known as the object's *server*); the other processes (known as *clients*) access the object via remote method invocations, made transparently by DSOM.

DSOM should also be used for applications that require objects to be *isolated* from the main program. This is usually done in cases where reliability is a concern — either to protect the object from failures in other parts of the application, or (less often), to protect the application from an object.

Note that some distributed applications may have special performance, reliability, or cooperative processing requirements, to which the SOM Replication framework is better suited. The Replication framework is oriented toward "groupware" (multi-party cooperative processing) applications, and has facilities for fault tolerance and recovery. The Replication framework is distinct from DSOM in that it maintains a complete replica of an object in each participant's address space, while DSOM establishes remote connections to shared objects. The Replication Framework is available only in the full-capability SOMobjects Developer Toolkit.

Chapter outline

Tutorial example

First, a complete example shows how an existing SOM class implementation (a "Stack") can be used, without modification, with DSOM to create a distributed "Stack" application. Using the "Stack" example as backdrop, the basic DSOM interfaces are introduced.

Programming DSOM applications

All DSOM applications involve three kinds of programming:

- *Client programming:* writing code that uses objects;
- *Server programming:* writing code that implements and manages objects; and
- *Implementing classes:* writing code that implements objects.

Three sections — "Basic Client Programming", "Basic Server Programming", and "Implementing Classes" — describe how to create DSOM applications from these three points of view. In turn, the structure and services of the relevant DSOM run-time environment are explained.

Note: The three sections are presented in the order above to aid in their explanation. However, the actual programming tasks are likely to be performed in the opposite order!

Additional examples are provided in these sections to illustrate DSOM services.

Configuring DSOM applications

The section "Configuring DSOM Applications" explains what is necessary to set up a DSOM application, once the application has been built.

Running DSOM applications

The section "Running DSOM Applications" explains what is necessary to run a DSOM application, once it has been built and configured.

DSOM and CORBA

Those readers interested in using DSOM as a CORBA-compliant ORB should read the section entitled "DSOM as a CORBA-compliant Object Request Broker." That section answers the question: How are CORBA concepts implemented in DSOM?

Advanced topics

The section on "Advanced Topics" covers the following:

- "Peer vs. client-server processes" demonstrates how peer-to-peer object interactions are supported in DSOM.

- "Dynamic Invocation Interface" details DSOM support for the CORBA dynamic invocation interface to dynamically build and invoke methods on remote objects.

- "Creating user-supplied proxy classes" describes how to override proxy generation by the DSOM run time and, instead, install a proxy object supplied by the user.

- "Customizing the default base proxy class" discusses how the **SOMDClientProxy** class can be subclassed to define a customized base class that DSOM will use during dynamic proxy-class generation.

- "Sockets class" describes how DSOM uses **Sockets** subclasses.

Error reporting and troubleshooting

The section on "Error Reporting and Troubleshooting" discusses facilities to aid in problem diagnosis.

6.2 A Simple DSOM Example

A sample "Stack" application is presented in this section as a tutorial introduction to DSOM. It demonstrates that, for simple examples like a "Stack", after very little work the class can be used to implement distributed objects that are accessed remotely. The example first presents the "Stack" application components and the steps that the implementor must perform before the application can be run, and then describes the run-time activity that results from executing the application. This run-time scenario introduces several of the key architectural components of the DSOM run-time environment.

The source code for this example is provided with the DSOM samples in the SOMobjects Developer Toolkit.

The "Stack" interface

The example starts with the assumption that the class implementor has successfully built a SOM class library DLL, called "stack.dll", in the manner described in Section 5.6, "Creating a SOM Class Library," of Chapter 5, "Implementing Classes in SOM." The DLL implements the following IDL interface.

```
#include <somobj.idl>

interface Stack: SOMObject
{
    const long stackSize = 10;
    exception STACK_OVERFLOW{};
    exception STACK_UNDERFLOW{};
    boolean full();
    boolean empty();
    long top()  raises(STACK_UNDERFLOW);
    long pop()  raises(STACK_UNDERFLOW);
    void push(in long element) raises(STACK_OVERFLOW);

    #ifdef __SOMIDL__
    implementation
    {
      releaseorder: full, empty, top, pop, push;
      somDefaultInit: override;
      long stackTop;                    // top of stack index
      long stackValues[stackSize];    // stack elements
      dllname = "stack.dll";
    };
    #endif
};
```

This DLL could have been built without the knowledge that it would ever be accessed remotely (that is, built following the procedures in Chapter 5). Note, however, that some DLLs may require changes in the way their classes pass arguments and manage memory, in order to be used by remote clients. (See the topic "Implementation Constraints" in Section 6.5, "Implementing Classes.")

The "Stack" class implementation

```
#define Stack_Class_Source
#include <stack.ih>

SOM_Scope boolean  SOMLINK full(Stack somSelf, Environment *ev)
{
    StackData *somThis = StackGetData(somSelf);
    StackMethodDebug("Stack","full");

    /* Return TRUE if stack is full. */
    return (_stackTop == stackSize);
}

SOM_Scope boolean  SOMLINK empty(Stack somSelf, Environment *ev)
{
    StackData *somThis = StackGetData(somSelf);
    StackMethodDebug("Stack","empty");

    /* Return TRUE if stack is empty.*/
    return (_stackTop == 0);
}

SOM_Scope long  SOMLINK top(Stack somSelf, Environment *ev)
{
    StackData *somThis = StackGetData(somSelf);
    StackMethodDebug("Stack","top");

    if (_stackTop > 0)
    {
       /* Return top element in stack without removing it from
        * the stack.
        */
       return (_stackValues[_stackTop-1]);
    }
    else
    {
        somSetException(ev, USER_EXCEPTION,
                        ex_STACK_UNDERFLOW, NULL);
        return (-1L);
    }
}

SOM_Scope long  SOMLINK pop(Stack somSelf, Environment *ev)
{
    StackData *somThis = StackGetData(somSelf);
    StackMethodDebug("Stack","pop");

    if (_stackTop > 0)
    {
       /* Return top element in stack and remove it from the
        * stack.
        */
       _stackTop--;
       return (_stackValues[_stackTop]);
    }
    else
    {
        somSetException(ev, USER_EXCEPTION,
                        ex_STACK_UNDERFLOW, NULL);
        return (-1L);
    }
}
```

```
SOM_Scope void  SOMLINK push(Stack somSelf,
                             Environment *ev, long el)
{
    StackData *somThis = StackGetData(somSelf);
    StackMethodDebug("Stack","push");

    if (_stackTop < stackSize)
    {
      /* Add element to top of the stack. */
      _stackValues[_stackTop] = el;
      _stackTop++;
    }
    else
    {
        somSetException(ev, USER_EXCEPTION,
                        ex_STACK_OVERFLOW, NULL);
    }
}

SOM_Scope void  SOMLINK somDefaultInit(Stack somSelf,
                                       somInitCtrl* ctrl)
{
    StackData *somThis;
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    StackMethodDebug("Stack","somDefaultInit");
    Stack_BeginInitializer_somDefaultInit;

    Stack_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /* stackTop is index into stackValues for next pushed
     * stack element.
     * stackValues[0..(stackSize-1)] holds stack elements.
     */
    _stackTop = 0;
}
```

Client program using a local stack

A simple client program *written to use a local "Stack" object* is displayed below. This C program is shown so that the differences between a local and remote client program can be highlighted.

```
#include <stack.h>

boolean OperationOK(Environment *ev);

int main(int argc, char *argv[])
{
  Environment ev;
  Stack stk;
  long num = 100;

  SOM_InitEnvironment(&ev);

  /* The StackNewClass invocation is optional and unnecessary
   * in the client program when the class object is created in
   * the SOMInitModule function that is invoked during DLL
   * initialization.
   */
  StackNewClass(Stack_MajorVersion, Stack_MinorVersion);
  stk = StackNew();
```

```
      /* Verify successful object creation */
      if ( stk != NULL )
      {
         while ( !_full(stk, &ev) )
         {
            _push(stk, &ev, num);
            somPrintf("Top: %d\n", _top(stk, &ev));
            num += 100;
         }

         /* Test stack overflow exception */
         _push(stk, &ev, num);
         OperationOK(&ev);

         while ( !_empty(stk, &ev) )
         {
            somPrintf("Pop: %d\n", _pop(stk, &ev));
         }

         /* Test stack underflow exception */
         somPrintf("Top Underflow: %d\n", _top(stk, &ev));
         OperationOK(&ev);
         somPrintf("Pop Underflow: %d\n", _pop(stk, &ev));
         OperationOK(&ev);

         _push(stk, &ev, -10000);
         somPrintf("Top: %d\n", _top(stk, &ev));
         somPrintf("Pop: %d\n", _top(stk, &ev));

         _somFree(stk);
      }

      SOM_UninitEnvironment(&ev);

      return(0);
   }

   boolean OperationOK(Environment *ev)
   {
       char *exID;

       switch (ev->_major)
       {
         case SYSTEM_EXCEPTION:
           exID = somExceptionId(ev);
           somPrintf("System exception: %s\n", exID);
           somdExceptionFree(ev);
           return (FALSE);

         case USER_EXCEPTION:
           exID = somExceptionId(ev);
           somPrintf("User exception: %s\n", exID);
           somdExceptionFree(ev);
           return (FALSE);

         case NO_EXCEPTION:
           return (TRUE);

         default:
           somPrintf("Invalid exception type in Environment.\n");
           somdExceptionFree(ev);
           return (FALSE);
      }
   }
```

Client program using a remote stack

The preceding program has been rewritten below to use DSOM to create and access a "Stack" object somewhere in the system. The exact location of the object does not matter to the application — it just wants a "Stack" object. Note that the stack operations of the two programs are identical. The main differences lie in stack creation and destruction, as highlighted below. (Also see "Memory management" later for more information on allocating and freeing memory.)

```
#include <somd.h>
#include <stack.h>

int main(int argc, char *argv[])
{
  Environment ev;
  Stack stk;
  long num = 100;

  SOM_InitEnvironment(&ev);
  SOMD_Init(&ev);

  /* The StackNewClass invocation is optional and unnecessary
   * in the client program when the class object is created in
   * the SOMInitModule function that is invoked during DLL
   * initialization.
   */
  StackNewClass(Stack_MajorVersion, Stack_MinorVersion);
  stk = _somdNewObject(SOMD_ObjectMgr, &ev, "Stack", "");

  /* Verify successful object creation */
  if ( OperationOK(&ev) )
  {
    while ( !_full(stk, &ev) )
    {
      _push(stk, &ev, num);
      somPrintf("Top: %d\n", _top(stk, &ev));
      num += 100;
    }

    /* Test stack overflow exception */
    _push(stk, &ev, num);
    OperationOK(&ev);

    while ( !_empty(stk, &ev) )
    {
      somPrintf("Pop: %d\n", _pop(stk, &ev));
    }

    /* Test stack underflow exception */
    somPrintf("Top Underflow: %d\n", _top(stk, &ev));
    OperationOK(&ev);
    somPrintf("Pop Underflow: %d\n", _pop(stk, &ev));
    OperationOK(&ev);

    _push(stk, &ev, -10000);
    somPrintf("Top: %d\n", _top(stk, &ev));
    somPrintf("Pop: %d\n", _top(stk, &ev));

    _somdDestroyObject(SOMD_ObjectMgr, &ev, stk);

    if ( OperationOK(&ev) )
    {
      somPrintf("Stack test successfully completed.\n");
    }
  }

  SOMD_Uninit(&ev);
  SOM_UninitEnvironment(&ev);

  return(0);
}
```

```
boolean OperationOK(Environment *ev)
{
   char *exID;

   switch (ev->_major)
   {
     case SYSTEM_EXCEPTION:
       exID = somExceptionId(ev);
       somPrintf("System exception: %s\n", exID);
       somdExceptionFree(ev);
       return (FALSE);

     case USER_EXCEPTION:
       exID = somExceptionId(ev);
       somPrintf("User exception: %s\n", exID);
       somdExceptionFree(ev);
       return (FALSE);

     case NO_EXCEPTION:
       return (TRUE);

     default:
       somPrintf("Invalid exception type in Environment.\n");
       somdExceptionFree(ev);
       return (FALSE);
   }
}
```

Let's step through the differences.

First, every DSOM program must include the file <somd.h> (or, when using C ++, <somd.xh>). This file defines constants, global variables, and run-time interfaces used by DSOM. Usually, this file is sufficient to establish all necessary DSOM definitions.

Next, DSOM requires its own initialization call.

```
         SOMD_Init(&ev);
```

The call to **SOMD_Init** initializes the DSOM run-time environment. **SOMD_Init** must be called before any DSOM run-time calls are made. A side-effect of calling **SOMD_Init** is that a run-time object, called the *DSOM Object Manager,* is created and a pointer to it is stored in the global variable **SOMD_ObjectMgr**, for programming convenience. The DSOM Object Manager provides basic run-time support for *clients* to find, create, destroy, and identify objects. The Object Manager is discussed in detail in the section entitled "Basic Client Programming."

Next, the local stack creation statement,

```
         stk = StackNew();
```

was replaced by

```
         stk = _somdNewObject(SOMD_ObjectMgr, &ev, "Stack", "");
```

The call to **somdNewObject** asks the DSOM Object Manager (**SOMD_ObjectMgr**) to create a "Stack" object, wherever it can find an implementation of "Stack". (There are other methods with which one can request *specific* servers.) If no object could be created, NULL is returned and an exception is raised. Otherwise, the object returned is a "Stack" proxy.

Note: On AIX, the following call may be needed before the **somdNewObject** call, if the "Stack" class implementation has been linked directly with the program executable (vs. using a dynamic link library, or DLL). This call will properly initialize the class for use by DSOM (this initialization is done in **SOMInitModule** for DLLs):

```
         StackNewClass(Stack_MajorVersion, Stack_MinorVersion);
```

A *proxy* is an object that is a local representative for a remote *target object*. A proxy inherits the target object's interface, so it responds to the same methods. Operations invoked on the proxy are not executed locally, but are forwarded to the "real" target object for execution. The client program always has a proxy for each remote target object on which it operates.

From this point on, the client program treats the "Stack" proxy exactly as it would treat a local "Stack". The "Stack" proxy takes responsibility for forwarding requests to and yielding results from the remote "Stack". For example,

```
_push(stk,&ev,num);
```

causes a message representing the method call to be sent to the server process containing the remote object. The DSOM run time in the server process decodes the message and invokes the method on the target object. The result (in this case, just an indication of completion) is then returned to the client process in a message. The DSOM run time in the client process decodes the result message and returns any result data to the caller.

At the end of the original client program, the local "Stack" was destroyed by the statement,

```
_somFree(stk);
```

whereas in the client program above, the "Stack" proxy and the remote "Stack" are destroyed by the statement,

```
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

If the client only wants to release its use of the remote object (freeing the proxy) without destroying the remote object, it can call the **somdReleaseObject** method instead of **somdDestroyObject**.

Finally, the client must shut down DSOM, so that any operating system resources acquired by DSOM for communications or process management can be returned:

```
SOMD_Uninit(&ev);
```

This call must be made at the end of every DSOM program.

Using specific servers

In DSOM, the process that manages a target object is called the object's *server.* Servers are implemented as programs that use SOM classes. Server implementations are registered with DSOM in an *Implementation Repository* — the Implementation Repository is a database queried by clients in order to find desired servers, and queried by DSOM in order to activate those servers upon demand.

The example above placed no constraints on the DSOM Object Manager as to where the remote "Stack" object should be created. The **somdNewObject** call creates a remote object of a specified class in an arbitrary server that implements that class. However, the DSOM Object Manager provides methods for finding specific servers.

For example, the client program above can be modified slightly to find a specific server named "StackServer", which has already been registered in DSOM's Implementation Repository. (Note that the programmer knew or discovered that the "StackServer" server implementation supports the "Stack" class.) The highlighted lines below show the changes that were made:

```
#include <somd.h>
#include <stack.h>

int main(int argc, char *argv[]) {
    Stack stk;
    Environment e;
    SOMDServer server;

    SOM_InitEnvironment(&ev);
    SOMD_Init(&ev);

    server =
      _somdFindServerByName(SOMD_ObjectMgr, &ev, "StackServer");
    stk = _somdCreateObj(server, &ev, "Stack", "");

    _push(stk,&ev,100);
    _push(stk,&ev,200);
```

```
            _pop(stk,&ev);
            if (!_empty(stk,&ev)) somPrintf("Top: %d\n", _top(stk,&ev));

            _somdDeleteObj(server, &ev, stk);
            _somdReleaseObject(SOMD_ObjectMgr, &ev, stk);
            _somdReleaseObject(SOMD_ObjectMgr, &ev, server);
            SOMD_Uninit(&ev);
            SOM_UninitEnvironment(&ev);

            return(0);
        }
```

This version of the program replaces the **somdNewObject** operation with calls to **somdFind-ServerByName** and **somdCreateObj**. The **somdFindServerByName** method consults the Implementation Repository to find the DSOM server implementation whose name is "Stack-Server", and creates a *server proxy,* which provides a connection to that server. Every DSOM server process has a *server object* that defines methods to assist in the creation and management of objects in that server. Server objects must be instances of **SOMDServer** or one of its subclasses. The **somdFindServerByName** returns a proxy to the **SOMDServer** object in the named server.

Once the client has the server proxy, it can create and destroy objects in that server. The **somdCreateObj** call creates an object of the class "Stack" in the server named "StackServer".

To free the remote "Stack" object, the example shows a **somdDeleteObj** request on the stack object's server. Next, **somdReleaseObject** requests are made on the DSOM Object Manager, to free the stack proxy and the server proxy in the client. (Note that these three calls are equivalent to the **somdDestroyObject** call in the previous example.)

A note on finding existing objects

The two examples above show how a remote, transient object can be created by a client, for its exclusive use. It is also likely that clients will want to find and use objects that are already in existence. In that case, the calls to **somdNewObject** or **somdCreateObj** would be replaced with other "lookup" calls on some directory object that would take an object name or identifier and return a proxy to the remote object.

Such a directory object could be implemented by the application as a persistent SOM object, using DSOM to share it among processes.

The basic mechanisms that DSOM provides for naming and locating objects will be discussed in section 6.3, "Basic Client Programming."

"Stack" server implementation

A server consists of three parts. First, a "main" program, when run, provides an address space for the objects it manages, and one or more process "threads" that can execute method calls. (Windows and AIX 3.2 currently do not have multi-thread support, while OS/2 and AIX 4.1 do.) Second, a server object, derived from the **SOMDServer** class, provides methods used to manage objects in the server process. Third, one or more class libraries provide object implementations. Usually these libraries are constructed as *dynamically linked libraries* (DLLs), so they can be loaded and linked by a server program dynamically.

In this simple example, we can use the default DSOM server program, which is already compiled and linked. The default server behaves as a simple server, in that it simply receives requests and executes them, continuously. The default server creates its server object from the class, **SOMDServer**. The default server will load any class libraries it needs upon demand.

The "Stack" class library, "stack.dll", can be used without modification in the distributed application. This is possible because the "Stack" class is "well formed" — *there are no methods that implicitly assume the client and the object are in the same address space.*

Thus, by using the default server and the existing class library, a simple "Stack" server can be provided without any additional programming!

An application may require more functionality in the server program or the server object than the default implementations provide. A discussion on how to implement server programs and server objects is found later in this chapter, in section 6.4, "Basic Server Programming".

Compiling the application

DSOM programs and class libraries are compiled and linked like any other SOM program or library. The header file "somd.h" (or for C++, "somd.xh") should be included in any source program that uses DSOM services. DSOM run-time calls can be resolved by linking with the SOMobjects Toolkit library: "libsomtk.a" on AIX and "somtk.lib" on OS/2 or Windows. (The DSOM DLL(s) — "somd.dll" for AIX or OS/2, or "somd1.dll" and "somd2.dll" for Windows — will be loaded at run time.)

For more information, see "Compiling and linking" in Chapter 3, "Using SOM Classes in Client Programs," and the same topic in Chapter 5, "Implementing Classes in SOM."

Installing the implementation

Before the application can be run, certain environment variables must be set and the stack class and server implementations must be registered in the SOM Interface Repository and DSOM Implementation Repository.

Setting environment variables

Several environment variables are used by SOM and DSOM. These variables need to be set before registering the DSOM application in the Interface and Implementation Repositories.

For this example, the following environment variables could be set as shown. A full description of the environment variables and how to set them is given in section 6.6, "Configuring DSOM."

On AIX (in the syntax of the default shell, /bin/ksh):

```
export HOSTNAME=machine3
export SOMIR=$SOMBASE/etc/som.ir:/u/myuserid/my.ir
export SOMDDIR=/u/myuserid/somddir
export LIBPATH=$LIBPATH:$SOMBASE/lib:/u/myuserid/lib
```

On OS/2:

```
set USER=pat
set HOSTNAME=machine3
set SOMDDIR=c:\somddir

rem *** The following variables are set in CONFIG.SYS by
rem *** the install program on OS/2, assuming "c:\som" is the
rem *** value of %SOMBASE% supplied by the user.
set SOMIR=c:\som\etc\som.ir;som.ir
set LIBPATH=.;c:\som\lib;<previous LIBPATH>
```

On Windows:

```
set USER=pat
set HOSTNAME=machine3
set SOMDDIR=c:\somddir

rem *** The following variables are usually set in AUTOEXEC.BAT
rem *** by the install program on Windows, assuming "c:\som"
rem *** is the value of %SOMBASE% supplied by the user.
set SOMIR=c:\som\etc\som.ir;som.ir
set PATH=.;c:\som\lib;<previous PATH>
```

USER identifies the user of a DSOM client application. DSOM sends the USER ID with every remote method call, in case the remote object wishes to perform any access-control checking. This is discussed later in the section "Basic Server Programming." (Note that USER is usually set automatically by AIX when a user logs in.)

HOSTNAME identifies the name of each machine running DSOM.

SOMIR gives a list of files that together constitute the Interface Repository. The IR is used by DSOM to guide the construction and interpretation of request messages. For DSOM, it is preferable to use *full* pathnames in the list of IR files, since the IR will be shared by several programs that may not all reside in the same directory.

SOMDDIR gives the name of a directory used to store DSOM configuration files, including the Implementation Repository.

LIBPATH (on AIX and OS/2) or PATH (on Windows) gives a list of directories where DLLs can be found.

Registering the class in the Interface Repository

Before an object can be accessed remotely by DSOM, it is necessary to register the class's interface and implementation in the Interface Repository (IR). DSOM uses the interface information when transforming local method calls on proxies into request messages transmitted to remote objects.

DSOM servers also consult the IR to find the name of the DLL for a dynamically loaded class. The DLL name for the "Stack" class must be specified using the **dllname="stack.dll"** modifier in the **implementation** statement of the "Stack" IDL. The Interface Repository is described in detail in Chapter 7, "The Interface Repository Framework."

The IDL specification of "Stack" is compiled into the Interface Repository using the following command:

```
sc -u -sir stack.idl           (on AIX or OS/2)
somc -u -sir stack.idl         (on Windows)
```

When a class has not been compiled into the Interface Repository, DSOM will generate a run-time error when an attempt is made to invoke a method from that class. The error indicates that the method's descriptor was not found in the IR.

Registering the server in the Implementation Repository

It is necessary to register a description of a server's implementation in the Implementation Repository. DSOM uses this information to assist clients in finding servers, and in activating server processes upon demand.

For this example, where the default server is used, we need only to identify the server's name, and the class that the server implements. This is accomplished using the **regimpl** utility discussed in section 6.6, "Configuring DSOM Applications". The following commands define a default server, named "StackServer", which supports the Stack class:

```
regimpl -A -i StackServer
regimpl -a -i StackServer -c Stack
```

Running the application

Starting the DSOM daemon

Before running a DSOM application, the DSOM daemon, **somdd,** must be started.

- On AIX or OS/2, the daemon can be started manually from the command line, or it could be started automatically from a start-up script run at boot time. It may be run in the background with the commands **somdd&** on AIX and **start somdd** on OS/2. (The **somdd** program requires no parameters. An optional **–q** parameter can be used to set "quiet" mode, to suppress messages.)

- On Windows, the daemon can be started with the **DSOM Daemon** icon in the SOM icon group or started in Windows from the Run option of the File menu. The **DSOM Daemon** icon will change colors to indicate that the daemon is ready, after which client and server programs can be started.

The **somdd** daemon is responsible for establishing a "binding" (that is, a connection) between a client process and a server. It will activate the desired server automatically, if necessary.

Once the DSOM daemon is running, the application may be started. This is accomplished by running the client program. If the StackServer is not running, it will be started automatically by the DSOM daemon when the client attempts to invoke a method on one of its objects.

"Stack" example run-time scenario

The following scenario steps through the actions taken by the DSOM run time in response to each line of code in the second "Stack" client program presented above. The illustration following the scenario is an illustration of the processes, and the objects within them, that participate in these actions.

- Initialize an environment for error passing:

```
SOM_InitEnvironment(&ev);
```

- Initialize DSOM:

```
SOMD_Init(&ev);
```

This causes the creation of the DSOM Object Manager (with **SOMDObjectMgr** interface). The global variable **SOMD_ObjectMgr** points to this object.

- Initialize "Stack" class object:

```
StackNewClass(Stack_MajorVersion, Stack_MinorVersion);
```

- Find the "StackServer" implementation and assign its proxy to the variable `server`:

```
server = _somdFindServerByName(SOMD_ObjectMgr, &ev, "StackServer");
```

This causes the creation of the server proxy object in the client process. Proxy objects are shown as shaded circles. Note that the "real" server object in the server process is not created at this time. In fact, the server process has not yet been started.

- Ask the server object to create a "Stack" and assign "Stack" proxy to variable `stk`:

```
stk = _somdCreateObj(server, &ev, "Stack", "");
```

This causes **somdd**, the DSOM daemon (already running), to activate the stack server process (by starting the "generic" server program). The stack server process, upon activation, creates the "real" **SOMDServer** object in the server process. The **SOMDServer** object works with the DSOM run time to create a local "Stack" object and return a "Stack" proxy to the client. (The details of this procedure are deferred until section 6.4, "Basic Server Programming".)

- Ask the "Stack" proxy to push 100 onto the remote stack:

```
_push(stk,&ev,100);
```

This causes a message representing the method call to be marshalled and sent to the server process. In the server process, DSOM demarshals the message and, with the help of the **SOMDServer**, locates the target "Stack" object upon which it invokes the method ("push"). The result (which is void in this case) is then passed back to the client process in a message.

- Invoke more "Stack" operations on the remote stack, via the proxy:

```
_push(stk,&ev,200);
_pop(stk,&ev);
if (!_empty(stk,&ev)) t = _top(stk,&ev);
```

- Explicitly destroy both the remote stack, the stack proxy, and the server proxy:

```
_somdDeleteObj(server, &ev, stk);
_somdReleaseObject(SOMD_ObjectMgr, &ev, stk);
_somdReleaseObject(SOMD_ObjectMgr, &ev, server);
```

- Free the error-passing environment:

```
SOM_UninitEnvironment(&ev);
```

This scenario has introduced the key processes in a DSOM application: client, server, and **somdd**. Also introduced are the key objects that comprise the DSOM run-time environment: the **SOMD_ObjectMgr** in the client process and the **SOMD_ServerObject** in the server process.

Summary

This example has introduced the key concepts of building, installing, and running a DSOM application. It has also introduced some (though not all) of the key components that comprise the DSOM application run-time environment, as pictured below.

The following sections, "Basic Client Programming", "Basic Server Programming" , and "Implementing Classes" provide more detail on how to use, manage, and implement remote objects, respectively.

Components of DSOM application run-time environment

CLIENT   SERVER

DSOM
Object Mgr

"real"
server

server
proxy   is a server proxy for

"real"
stack

stack
proxy   is a stack proxy for

INTERFACE
REPOSITORY

DSOM
DAEMON

(somdd)

IMPLEMENTATION
REPOSITORY
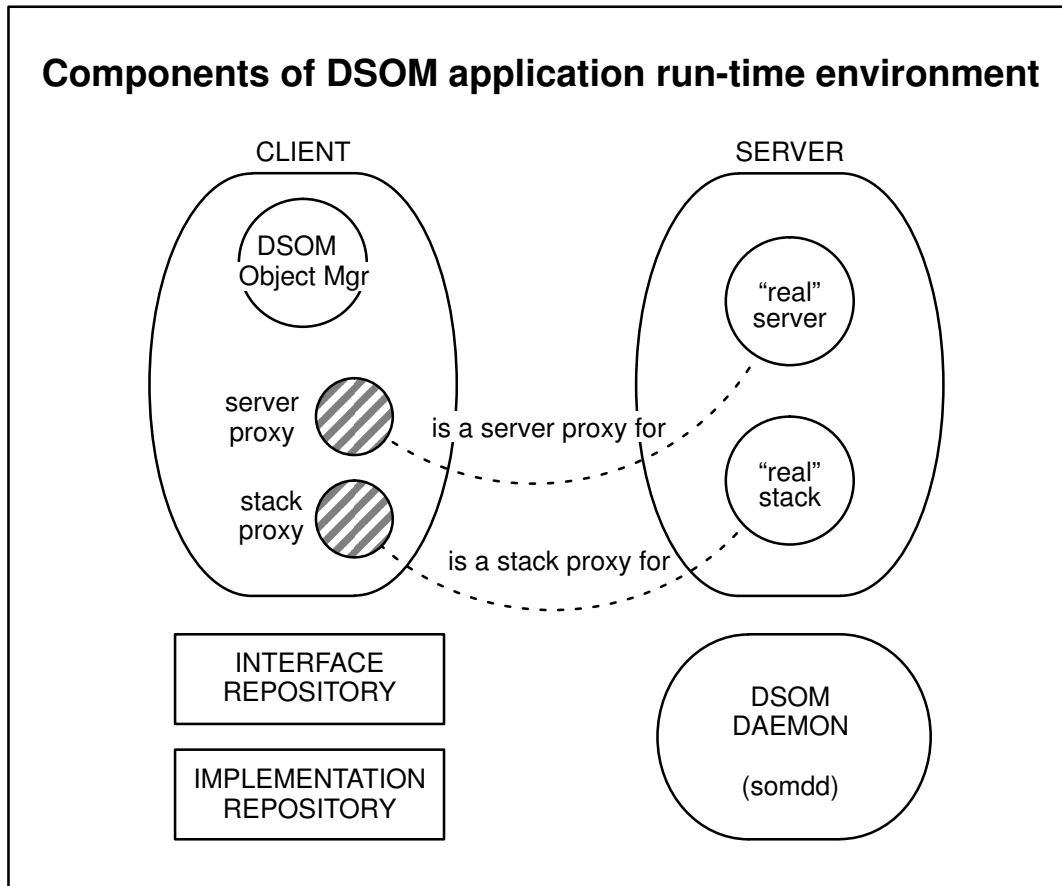
6.3 Basic Client Programming

For the most part, client programming in DSOM is exactly the same as client programming in SOM, since DSOM transparently hides the fact that an object is remote when the client accesses the object.

However, a client application writer also needs to know how to create, locate, use, save, and destroy remote objects. (This is *not* done using the usual SOM bindings.) The DSOM run-time environment provides these services to client programs primarily through the DSOM Object Manager. These run-time services will be detailed in this section. Examples of how an application developer uses these services are provided throughout the section.

DSOM Object Manager

DSOM defines a *DSOM Object Manager,* which provides services needed by clients to create, find and use objects in the DSOM run time environment.

The DSOM Object Manager is derived from an abstract, generic "object manager" class, called **ObjectMgr**. This abstract **ObjectMgr** class defines a basic set of methods that support object creation, location (with implicit activation), and destruction.

As an abstract class, **ObjectMgr** defines only an interface — there is no implementation associated with **ObjectMgr**. Consequently, an application should *not* create instances of the **ObjectMgr** class.

An abstract Object Manager class was defined under the expectation that applications will often need simultaneous access to objects implemented and controlled by a variety of object systems. Such object systems may include other ORBs (in addition to DSOM), persistent object managers, object-oriented databases, and so forth. It is likely that each object system will provide the same sort of basic services for object creation, location, and activation, but each using a different interface.

Thus, the **ObjectMgr** abstract class defines a simple and "universal" interface that can be mapped to any object system. The application would only have to understand a single, common **ObjectMgr** interface. Under this scheme, specific object managers are defined by subclassing the **ObjectMgr** class and overriding the **ObjectMgr** methods to map them into the object system-specific programming interfaces.

DSOM's Object Manager, **SOMDObjectMgr**, is defined as a specific class of **ObjectMgr**. It defines methods for:

- Finding servers that implement particular kinds of objects

- Creating objects in servers

- Obtaining object identifiers (string ids)

- Finding objects, given their identifiers

- Releasing and destroying objects

These functions will be discussed in the remainder of this section.

Note: The OMG has standardized an "object lifecycle" service, which includes support for creating and destroying distributed objects. The DSOM Object Manager may be augmented in the future with an OMG-compliant lifecycle service.

Initializing a client program

A client application must declare and initialize the DSOM run time before attempting to create or access a remote object. The **SOMD_Init** procedure initializes all of the DSOM run time, including the **SOMDObjectMgr** object. The global variable **SOMD_ObjectMgr** is initialized to point to the local DSOM Object Manager.

A client application must also initialize all application classes used by the program. For each class, the corresponding <*className*>**NewClass** call should be made.

Note: In non-distributed SOM programs, the *<className>***New** macro (and the **new** operator provided for each class by the SOM C++ bindings) implicitly calls the procedure *<class-Name>***NewClass** when creating a new object. This is not currently possible in DSOM because, when creating remote objects, DSOM uses a generic method that is not class specific.

This was shown in the "Stack" example in section 6.2. In a similar example of an application that uses "Car" and "Driver" objects, the initialization code might look like this:

```
#include <somd.h>    /* needed by all clients */
#include <Car.h>     /* needed to access remote Car */
#include <Driver.h>  /* needed to access remote Driver */

main()
{
    Environment ev; /* ev used for error passing */
    SOM_InitEnvironment(&ev);

    /* Do DSOM initialization */
    SOMD_Init(&ev);

    /* Initialize application classes */
    CarNewClass(Car_MajorVersion, Car_MinorVersion);
    DriverNewClass(Driver_MajorVersion, Driver_MinorVersion);
    ...
}
```

As shown, client programs should include the "somd.h" file (or, for C++ programs, the "somd.xh" file) in order to define the DSOM run-time interfaces.

Note also that, since **Environment**s are used for passing error results between a method and its caller, an **Environment** variable (*ev*) must be declared and initialized for this purpose.

The calls to "CarNewClass" and "DriverNewClass" are required if the client will be creating or accessing Cars and Drivers. The procedures "CarNewClass" and "DriverNewClass" create class objects for the classes "Car" and "Driver". When a DSOM Object Manager method like **somdNewObject** is invoked to create a "Car", it expects the "Car" class object to exist. If the class does not yet exist, the "ClassNotFound" exception will be returned.

Exiting a client program

At the end of a client program, the **SOMD_Uninit** procedure must be called to free DSOM run-time objects, and to release system resources such as semaphores, shared memory segments, and so on.

For example, the exit code in the client program might look like this:

```
    ...
    SOMD_Uninit(&ev);
    SOM_UninitEnvironment(&ev);
}
```

Observe also the **SOM_UninitEnvironment** call, which frees any memory associated with the specified **Environment** structure.

Note: When a Windows DSOM client receives a WM_QUIT message while processing a remote method invocation, DSOM will clean up and terminate the client without returning to the client's WinProc or WinMain.

Creating remote objects

Distributed objects can be created in several different ways in DSOM.

• The client can create an object on any server that implements that class of object.

• The client can find a specific server upon which to create an object.

- A server can create an object and register a reference to the object in some well-known directory. (An *object reference* contains information that reliably identifies a particular object.)

The first two cases are discussed immediately below. The last case is discussed near the end of this section.

Creating an object in an arbitrary server

Following is an example of how to create a new remote object in the case where the client does not care in which server the object is created. In this situation, the client defers these decisions to the DSOM Object Manager (**SOMD_ObjectMgr**) by using the **somdNewObject** method call, which has this IDL definition:

```
// (from file om.idl)

SOMObject somdNewObject(in Identifier objclass, in string hints);

// Returns a new object of the named class.  This is a "basic"
// creation method, where the decisions about where and how to
// create the object are mostly left up to the Object Manager.
// However, the Object Manager may optionally define creation
// "hints" which the client may specify in this call.
```

Here is the example of a how a remote "Car" would be created using **somdNewObject**:

```
#include <somd.h>
#include <Car.h>

main()
{
    Environment ev;
    Car car;

    SOM_InitEnvironment(&ev);
    SOMD_Init(&ev);

    /* create the class object */
    CarNewClass(Car_MajorVersion, Car_MinorVersion);

    /* create a Car object on some server, let the
       Object Manager choose which one */
    car = _somdNewObject(SOMD_ObjectMgr, &ev, "Car", "");
    ...
}
```

The main argument to the **somdNewObject** method call is a string specifying the name of the class of the desired object. The last argument is a string that may contain "hints" for the Object Manager when choosing a server. In this example, the client is providing no hints. (Currently, the DSOM Object Manager simply passes the hints to the server object in a **somdCreateObj** call.)

Proxy objects

As far as the client program is concerned, when a remote object is created, a pointer to the object is returned. However, what is actually returned is a pointer to a *proxy object*, which is a local representative for the remote *target object*.

Proxies are responsible for ensuring that operations invoked on it get forwarded to the "real" target object that it represents. The DSOM run time creates proxy objects automatically, wherever an object is returned as a result of some remote operation. The client program will always have a proxy for each remote target object on which it operates. Proxies are described further in the sections entitled "DSOM as a CORBA-compliant Object Request Broker" and "Advanced Topics".

In the example above, a pointer to a "Car" proxy is returned and put in the variable "car". Any subsequent methods invoked on "car" will be forwarded and executed on the corresponding remote "Car" object.

Proxy objects inherit behavior from the **SOMDClientProxy** class.

Servers and server objects

In DSOM, the process that manages a target object is called the object's *server.* Servers are implemented as programs that use SOM classes. The example above placed no constraints on the DSOM Object Manager as to which server should create the remote "Car" object. However, if the client desires more control over distribution of objects, the DSOM Object Manager provides methods for finding specific servers.

Server implementations are registered with DSOM in an *Implementation Repository.* Server implementations are described by a unique ID, a unique (user-friendly) name, the program name that implements the server, the classes that are implemented by the server, the machine on which the server is located, whether the server is multi-threaded, and so forth. (See section 6.6 for more information on registering server implementations.) A client can ask the DSOM Object Manager to find a particular server:

- By name,
- By ID,
- By a class it supports.

When a client asks for a "server", it is given (a proxy to) a *server object* that provides interfaces for managing the objects in the server. There is one server object per server process. All server objects are instances of the **SOMDServer** class, or its subclasses. The default method provided by **SOMDServer** for creating objects is:

```
// (from file somdserv.idl)

SOMObject somdCreateObj(in Identifier objclass, in string hints);

// Creates an object of the specified class.  This method
// may optionally define creation "hints" which the client
// may specify in this call.  (Hints are ignored by default.)
```

Section 6.4 explains how to create application-specific server objects, derived from **SOMD-Server,** which override **SOMDServer** methods and introduce their own methods for object management.

Creating an object in a specific server

The following example demonstrates how a client application creates a new object *in a remote server chosen by the client.* The DSOM Object Manager method **somdFindServerByName** is used to find and create a proxy to the server object for the server implementation named "myCarServer". The method **somdCreateObj** is then invoked on the server object to create the remote "Car". A proxy to the remote "Car" is returned. (The "Stack" client presented in the previous section used the same methods to create a remote "Stack".)

```
/* find a specific Car server */
    server =
      _somdFindServerByName(SOMD_ObjectMgr, &ev, "myCarServer");

/* create a remote Car object on that server */
    car = _somdCreateObj(server, &ev, "Car", "");
    ...
}
```

Note: If the specified server does *not* provide any implementation of the desired class, a NULL pointer will be returned and a "ClassNotFound" exception will be raised.

Three other methods can be invoked on the DSOM Object Manager to find server implementations: **somdFindServer**, **somdFindServersByClass**, and **somdFindAnyServerByClass**. The IDL declarations of these methods follow:

```
SOMDServer somdFindServer(in ImplId serverid);

sequence<SOMDServer> somdFindServersByClass(in Identifier objclass);

SOMDServer somdFindAnyServerByClass(in Identifier objclass);
```

The **somdFindServer** method is similar to the **somdFindServerByName** method, except that the server's *implementation ID* (of type **ImplId**) is used to identify the server instead of the server's user-friendly name (or "alias"). The implementation ID is a unique string generated by the Implementation Repository during server registration. (See section 6.6 for more details.)

The **somdFindServersByClass** method, given a class name, returns a sequence of *all* servers that support the given class. The client program may then choose which server to use, based on the server's name, program, or other implementation attributes (for example, the server is multi-threaded). (See the topic below, "Inquiring about a remote object's implementation.")

Finally, the **somdFindAnyServerByClass** method simply selects any one of the server implementations registered in the Implementation Repository that supports the given class, and returns a server proxy for that server.

Once the server proxy is obtained, methods like **somdCreateObj**, shown in the example above, can be invoked upon it to create new objects.

Inquiring about a remote object's implementation

A client may wish to inquire about the (server) implementation of a remote object. All objects in a server, including the "server object", share the same implementation definition. This is common when using the **somdFindServersByClass** call, where a sequence of server proxies is returned, and some choice must be made about which to use.

When a proxy is obtained by a client, the client can inquire about the underlying server implementation by obtaining its corresponding **ImplementationDef**. An **ImplementationDef** object contains a set of attributes that describe a server implementation. To get the **ImplementationDef** associated with a remote object, the **get_implementation** method (implemented on **SOMDObject** and inherited by **SOMDClientProxy**) can be called.

For example, if a program has a proxy for a remote server object, it can get the ImplementationDef for the server with method calls similar to the following:

```
ImplementationDef implDef;
SOMDServer server;

...
implDef = _get_implementation(server, &ev);
```

Once the **ImplementationDef** has been obtained, the application can access its attributes using the **_get_impl_xxx** methods.

The **ImplementationDef** class is discussed further in section 6.6, "Configuring DSOM."

Destroying remote objects

There are several ways of destroying objects or their proxies in DSOM, just as there are several ways to create objects. Remote objects can be asked to destroy themselves, or, the **SOMDObjectMgr** and the **SOMDServer** can participate in the deletion.

Destroying objects via a proxy

DSOM provides means for deleting remote objects via their proxies. For example, if **somFree** is invoked on a proxy, the **somFree** call gets forwarded directly to the target object, just like any other target method call. For example,

```
_somFree(car);
```

frees the remote car. Note that, by default, invoking **somFree** on the proxy does not free the proxy, only the remote object. However, the following call can be issued as part of a client-program initialization, so that invoking **somFree** on a proxy frees both the remote object and the proxy:

```
__set_somd21somFree(SOMD_ObjectMgr, ev, TRUE);
```

All subsequent invocations of **somFree** on a proxy object will result in both the remote object and the proxy being freed.

To be explicit about whether the proxy or the remote object is being deleted, the methods **somdTargetFree** and **somdProxyFree**, defined on proxies, can be used:

```
_somdTargetFree(car, &ev);
```

frees the remote "Car" (but not the proxy) and

```
_somdProxyFree(car, &ev);
```

frees the proxy (but not the remote "Car").

Note: CORBA specifies a third method for deleting object references. (Proxies are a specialized type of object reference.) The method

```
_release(car, &ev);
```

deletes the proxy (but not the target object).

Destroying objects via the DSOM Object Manager

Having created a remote object with **somdNewObject** or **somdCreateObj**, the remote object and its local proxy may be destroyed by invoking the method **somdDestroyObject** on the DSOM Object Manager using the proxy as an argument. For example,

```
/* create the car */
car = _somdNewObject(SOMD_ObjectMgr, &ev, "Car", "");
...
/* destroy the car (and its proxy) */
_somdDestroyObject(SOMD_ObjectMgr, &ev, car);
```

If the client does not want to destroy the remote object, but is finished working with it, the **somdReleaseObject** method should be used instead. For example,

```
_somdReleaseObject(SOMD_ObjectMgr, &ev, car);
```

This deletes the local proxy, but not the remote object.

Both **somdDestroyObject** and **somdReleaseObject** are defined on the **ObjectMgr**, so that the Object Manager is aware of the client's actions, in case it wants to do any bookkeeping.

The object passed to either the **somdDestroyObject** method or the **somdReleaseObject** method can be either a local SOM object or a DSOM proxy object. When a local SOM object is passed, **somdDestroyObject** has the same behavior as **somFree**. If a local SOM object is passed to **somdReleaseObject**, however, this has no effect.

Destroying objects via a server object

The **somdDestroyObject** method described above sends a request to delete a remote object to the object's server. It does so to ensure that the server has an opportunity to participate in, if not perform, the deletion. The method defined on the **SOMDServer** class for destroying objects is **somdDeleteObj**. If the client has a proxy for the server object, it can also invoke **somdDeleteObj** directly, instead of calling **somdDestroyObject.**

Destroying objects via the server object, rather than asking the object itself (as in **somFree** or **somdTargetFree**), allows the server object do any clean-up that is needed. For simple applications, this may not be necessary, but for applications that provide their own application-tailored server objects, it may be critical. See, for example, the persistent server example in the Section 6.4, entitled "Basic Server Programming."

Creating remote objects using user-defined metaclasses

An application may wish to define its own constructor methods for a particular class, via a user-supplied metaclass. In this case, the **somdNewObject** method should not be used, since it simply calls the default constructor method, **somNew**, defined by **SOMClass**.

Instead, the application can obtain a proxy to the actual class object in the server process. It can do so via the **somdGetClassObj** method, invoked on the **SOMDServer** proxy returned by one of the **somdFindServerXxx** methods. The application-defined constructor method can then be invoked on the proxy for the remote class object.

Note: The same issues apply to destructor methods. If the application defines its own destructor methods, they can be called via the class object returned by **somdGetClassObj**, as opposed to calling **somdDestroyObject**.

The following example creates a new object in a remote server using an application-defined constructor method, "makeCar", which is assumed to have been defined in the metaclass of "Car", named "MetaCar".

```
#include <somd.h>
#include <Car.h>
main( )
{
    Environment ev;
    SOMDServer server;
    Car car;
    MetaCar carClass;

    SOM_InitEnvironment(&ev);
    SOMD_Init(&ev);

    /* find a Car server */
    server = _somdFindAnyServerByClass(SOMD_ObjectMgr, &ev, "Car");

    /* get the class object for Car */
    carClass = (MetaCar) _somdGetClassObj(server, &ev, "Car");

    /* create the car object */
    car = _makeCar(carClass, &ev, "Red", "Toyota", "2-door");

    ...
}
```

Saving and restoring references to objects

A proxy is a kind of "object reference". An *object reference* contains information that is used to identify a target object.

To enable clients to save references to remote objects (in a file system, for example) or exchange references to remote objects (with other application processes), DSOM must be able to externalize proxies. To "externalize a proxy" means to create a string ID for a proxy that can be used by any process to identify the remote target object. DSOM must also support the translation of string IDs back into proxies.

The DSOM Object Manager defines two methods for converting between proxies and their string ids: **somdGetIdFromObject** and **somdGetObjectFromId**.

Here is an example client program that creates a remote "Car" object. It generates a string ID corresponding to the proxy, and saves the string ID to a file for later use.

```
#include <stdio.h>
#include <somd.h>
#include <Car.h>
main( )
{
    Environment ev;
    Car car;
    string somdObjectId;
    FILE* file;

    SOM_InitEnvironment(&ev);
    SOMD_Init(&ev);

    /* create a remote Car object */
    car = _somdNewObject(SOMD_ObjectMgr, &ev, "Car", "");

    /* save the reference to the object */
    somdObjectId = _somdGetIdFromObject(SOMD_ObjectMgr, &ev, car);
    file = fopen("/u/joe/mycar", "w");
    fprintf(file, "%s", somdObjectId);
...
```

Next is an example client program that retrieves the string ID and regenerates a valid proxy for the original remote "Car" object (assuming the remote "Car" object can still be found in the server).

```
...
    Environment ev;
    Car car;
    char buffer[256];
    string somdObjectId;
    FILE* file;

...
    /* restore proxy from its string form */
    file = fopen("/u/joe/mycar", "r");
    somdObjectId = (string) buffer;
    fscanf(file, "%s", somdObjectId);
    car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
```

Once the proxy has been regenerated, methods can be invoked on the proxy and they will be forwarded to the remote target object, as always.

Note: The **somdGetIdFromObject** and **somdGetObjectFromId** methods directly correspond to the CORBA methods **ORB_object_to_string** and **ORB_string_to_object**, defined on the **ORB** class.

Finding existing objects

The **SOMDObjectMgr** and **SOMDServer** classes support the methods described above, which allow clients to create objects in servers. However, it is also likely that clients will want to find and use objects that have already been created, usually by the servers that implement them. For example, a print service will create printer objects, and must then export them to clients. In that case, the calls to **somdNewObject** or **somdCreateObj** would be replaced with other "lookup" calls on some directory (server) object which would take an object name or identifier and return a proxy to a corresponding remote object. Likewise, the server that owns the object would register the exported object in the directory.

It is important to understand that DSOM does not provide a directory service such as the one described. But such a directory object could be implemented by the application, where a table or collection object maps object names to proxies. The string ids for the proxies in the directory object could be saved using a file (as above) or a persistent object (via the Persistence Framework of the SOMobjects Developer Toolkit). A directory server implemented using DSOM could be used to share the directory among processes.

Upon a lookup call, the directory server could find the corresponding proxy (or its string ID) in the directory, and return it to the caller.

Finding server objects

The DSOM Object Manager can be used to find server object proxies using the **somdFindServer*Xxx*** methods. However, it is important to point out that an application can also augment those services, by managing server proxies itself. Server proxies can be maintained in an application-specific directory, stored in a file, or passed from process to process, just as any other proxies.

Invoking methods on remote objects

As described earlier, DSOM proxies are local representatives of remote objects, and as such, they can be treated like the target objects themselves. Method calls are invoked in exactly the same manner as if the object is local. This is true both for method calls using the *static bindings* (as most of our examples have shown), as well as for *dynamic dispatching* calls, where SOM facilities (such as the **somDispatch** method) are used to construct method calls at run time.

CORBA 1.1 also defines a dynamic invocation interface that is implemented by DSOM. It is described later in section 6.9, "Advanced Topics".

The DSOM run time is responsible for transporting any input method argument values supplied by the caller (defined by legal IDL types) to the target object in a remote call. Likewise, the DSOM run time transports the return value and any output argument values back to the caller following the method call.

Note: DSOM uses the Interface Repository (IR) to discover the "signature" of a method (that is, the method's prototype). It is important that the contents of the IR match the method bindings used by the application program (that is, the same IDL file is used to update the IR and to generate bindings).

DSOM can make remote invocations only of methods whose parameter types are among the following IDL types: basic types (**short, long, unsigned short, unsigned long, float, double, char, boolean, octet**), **enum, struct, union, sequence, string, array, any,** and **object**. The members of a **struct, union, sequence,** or **array,** and the value of an **any,** must also be from the above list of supported DSOM types.

In addition to the preceding types, DSOM also supports method parameters of type **pointer** to one of the above types (for example, **long***). Pointers to **pointers** are not supported, however, and **pointers** embedded within one of the above types (for example, a **pointer** within a **struct**) are not supported. The "**void ***" type is also not supported. Currently, DSOM has the limitation that NULL **pointer** values cannot be returned as **inout** or **out** method arguments, although it is expected that this limitation will be addressed in a future release.

Types declared as SOMFOREIGN types are not currently supported by DSOM. Because the SOM **somId** type is declared as a SOMFOREIGN type, this implies that any method having a parameter of type **somId** cannot be invoked remotely using DSOM. This restriction includes the SOM methods: **somRespondsTo**, **somSupportsMethod**, **somGetMethodDescriptor**, **somGetMethodIndex**, and **somGetNthMethodInfo**.

When a method parameter is an **object** type (that is, an instance of **SOMObject** or some class derived from **SOMObject**), a client program making a remote invocation of that method must pass an object reference for that parameter, rather than passing a local **SOMObject**, unless the client program is also a DSOM server program, in which case DSOM will automatically convert the local object into an object reference.

Methods having the **procedure** SOM IDL modifier cannot be invoked remotely using DSOM. This is because these "methods" are called directly, rather than via the normal method resolution mechanisms on which DSOM relies.

Determining memory allocation and ownership

When a method is invoked that returns a result of type **string**, **sequence**, or **array**, DSOM will allocate memory in the client's address space for the result. Ownership of this memory becomes the responsibility of the client program. When the client program has finished using it, the client should free the memory using the **ORBfree** function, rather than using **free** or **SOMFree**. (This is because the memory has been allocated by DSOM using special memory management techniques; therefore, the client should ask DSOM to also free the memory.)

When invoking a method using DSOM, the client program is responsible for providing storage for all **in** arguments and for all **inout/out** arguments, with the following exceptions: DSOM will allocate storage for a **string** or for the **_buffer** field of a **sequence** when used as an **out** argument, and will allocate storage for the **_value** field of an **any** when used as an **inout** or **out** argument. This storage becomes the responsibility of the client program and should later be freed using **ORBfree**. For a **string** or **sequence** used as an **inout** argument, the **out** result is constrained to be no larger than the size of the **in** argument allocated by the client.

Passing object references in method calls

When pointers to objects are returned as method output values (as in the previous examples), DSOM automatically converts the object pointers (in the server) to object proxies in the client.

Likewise, when a client passes object (proxy) pointers as input arguments to a method, DSOM automatically converts the proxy argument in the client to an appropriate object reference in the server.

Note: If the proxy is for an object that is in the same server as the target object, DSOM gives the object reference to the server object for resolution to a SOM object pointer. Otherwise, DSOM leaves the proxy alone, since the proxy must refer to an object in some process other than the target's server.

Memory management

DSOM programs must manage four different kinds of memory resources: objects, object references, Environment structures, and blocks of memory. There are different techniques for allocating and releasing each kind of resource.

Objects and object references

Creating and destroying remote objects was discussed previously in this section (see "Creating remote objects" and "Destroying remote objects"). Creating and destroying local objects is described in section 3.2, "Using SOM Classes — the Basics," in Chapter 3, "Using SOM Classes in Client Programs." Object references are typically created automatically by DSOM as needed by the client program. They are also released in the client program by using either the **release** method or the **somdProxyFree** method. (The two methods are equivalent.)

Environment structures

When a client invokes a method and the method returns an exception in the Environment structure, it is the client's responsibility to free the exception. This is done by calling either **exception_free** or **somdExceptionFree** on the Environment structure in which the exception was returned. (The two functions are equivalent.) A similar function, **somExceptionFree**, is available for SOM programmers; however DSOM programmers can use **somdExceptionFree** to free all exceptions (regardless of whether they were returned from a local or remote method call).

Blocks of memory

For allocating and releasing blocks of memory within a client program, SOM provides the **SOMMalloc** and **SOMFree** functions (analogous to the C "malloc" and "free" functions). The "Memory Management" section of Chapter 3 describes these functions. To release memory allocated by DSOM in response to a remote method call, however, DSOM client programs should use the **ORBfree** function.

For example, when a method is invoked that returns a result of type string, sequence, or array, DSOM will allocate memory for the result in the client's address space. Ownership of this memory becomes the responsibility of the client program. When finished using this memory, the client program should free it using the **ORBfree** function, rather than **free** or **SOMFree**. This is because the memory has been allocated by DSOM using special memory-management techniques; therefore, the client should ask DSOM to also free the memory. If the storage is freed using **SOMFree** rather than **ORBfree**, then memory leaks will result.

The differences between the **SOMFree** and **ORBfree** functions are twofold:

- First, **SOMFree** should only be used to free memory *not* allocated by DSOM (for example, memory the client program allocated itself using **SOMMalloc**), while **ORBfree** should be used to free memory allocated by DSOM in response to a remote method call.

- Second, **SOMFree** only frees a single block of memory (in the same way that the C "free" function does), while **ORBfree** will free an entire data structure, including any allocated blocks of memory within in. For example, if a remote method call returns a sequence of structs, and each struct contains a string, **ORBfree** will free, with a single call, not only the sequence's "_buffer" member, but also each struct and all the strings within the structs. Freeing a similar data structure using **SOMFree** would require multiple calls (one for each call to **SOMMalloc** used to build the data structure).

Some programmers may wish to use a single function to free blocks of memory, regardless of whether they were allocated locally or by DSOM in response to a remote method call. For these programmers, DSOM provides a function, **SOMD_NoORBfree**, which can be called just after calling **SOMD_Init** in the client program. (It requires no arguments and returns no value.) This function specifies that the client program will free all memory blocks using **SOMFree**, rather than **ORBfree**. In response to this call, DSOM will *not* keep track of the memory it allocates for the client. Instead, it assumes that the client program will be responsible for walking all data structures returned from remote method calls, while calling **SOMFree** for each block of memory within.

Memory management for method parameters

For each method, five SOM IDL modifiers are available to specify the method's memory-management policy (that is, whether the caller or the object owns the parameters' memory after the method is invoked). These modifiers are **memory_management**, **caller_owns_result**, **caller_owns_parameters**, **object_owns_result**, and **object_owns_parameters**. For a complete description of these modifiers and their meanings, see the section entitled "Implementation Statements" in Chapter 4, "SOM IDL and the SOM Compiler."

Note that the memory-management policy for a particular parameter applies to the parameter and all the memory embedded within it (for example, if a struct is owned by the caller, then so are all the struct's members). Also note that the "object-owned" memory-management policy, specified by the **object_owns_result** and **object_owns_parameters** modifiers, is *not* supported by DSOM for methods invoked using the Dynamic Invocation Interface (DII). (This is because the "object-owned" policy is not CORBA-compliant, and because it precludes reusing **Request** objects to invoke a method multiple times.)

The CORBA policy for parameter memory management

When a class contains the SOM IDL modifier **memory_management = corba**, this signifies that all methods introduced by the class follow the CORBA specification for parameter memory management, except where a particular method has an explicit modifier (**object_owns_result** or **object_owns_parameters**) that indicates otherwise. The remainder of this section describes the CORBA specification for parameter memory management.

Caller frees parameters and return results

The CORBA memory-management policy specifies that the caller of a method is responsible for freeing all parameters and the return result after the method call is complete. This applies regardless of whether the parameter was allocated by the caller or the object (or, in the case of a remote method call, by DSOM). In other words, the CORBA policy asserts that parameters are uniformly "caller-owned".

Allocation responsibilities

Whether the parameter or return result should be allocated by the caller or by the object depends on the *type* of the parameter and its *mode* ("in", "inout", "out", or "return"). In general, the caller is responsible for allocating storage for most parameters and return results. More specifically, CORBA requires that storage for all "in" arguments, for all "inout" or "out" arguments, and for all "return" results must be provided by the client program, with certain exceptions as itemized below.

The object is responsible for allocating storage as follows:

- for **strings** when used as "out" arguments or as "return" results,
- for the "_buffer" field of **sequences** when used as "out" arguments or as "return" results,
- for the "_value" field of **anys** when used as "inout" or "out" arguments or as "return" results,
- for **pointer** types when used as "inout" or "out" arguments or as "return" results,
- for **arrays** when used as "return" results, and
- for objects when used as "inout" or "out" arguments or as "return" results.
    Note: For "inout" **strings** and **sequences**, the "out" result is constrained to be no larger than the size of the "in" argument allocated by the client.

Ownership of memory allocated in the above cases becomes the responsibility of the client program. For remote method calls, when a remote object allocates memory for a parameter or "return" value, DSOM subsequently allocates memory in the client's address space for the parameter or result. For a parameter/result that is an object (rather than a block of memory) DSOM automatically creates an object reference (a proxy object) in the client's address space. In each case, the memory or the proxy object becomes the responsibility of the client program and should later be freed by the client, using **ORBfree** for blocks of memory or **release** for proxy objects.

The 'somdReleaseResources' method and object-owned parameters

As stated earlier, the CORBA policy asserts that method parameters and return results are uniformly caller-owned. This means the method *caller* has the responsibility for freeing memory after invoking a method, regardless of whether the memory was allocated by the caller or the object.

A class implementor can designate certain method parameters and results as object-owned, however, by using the **object_owns_result** and **object_owns_parameters** SOM IDL modifiers. These modifiers signify that the *object*, rather than the caller, is responsible for freeing the memory associated with the parameter/result. For "in" parameters, the object can free the memory any time after receiving it; for "inout" and "out" parameters, and for return results, the object will free the memory sometime before the object is destroyed. (See the section entitled "Implementation statements" in Chapter 4, "SOM IDL and the SOM Compiler," for more information on these modifiers.)

When a DSOM client program makes a remote method invocation, via a proxy, and the method being invoked has an object-owned parameter or return result, then the client-side memory associated with the parameter/result will be owned by the caller's proxy, and the server-side memory will be owned by the remote object. The memory owned by the caller's proxy will be freed when the proxy is released by the client program. (The time at which the server-side memory will be freed depends on the implementation of the remote object.)

A DSOM client can also instruct a proxy object to free all memory that it owns on behalf of the client without releasing the proxy (assuming that the client program is finished using the object-owned memory), by invoking the **somdReleaseResources** method on the proxy object. Calling **somdReleaseResources** can prevent unused memory from accumulating in a proxy.

For example, consider a client program repeatedly invoking a remote method "get_string", which returns a string that is designated (in SOM IDL) as "object-owned". The proxy on which the method is invoked will store the memory associated with all the returned strings, even if the strings are not unique, until the proxy is released. If the client program only uses the last result returned from "get_string", then unused memory accumulates in the proxy. The client program can prevent this by invoking **somdReleaseResources** on the proxy object periodically (for example, each time it finishes using the result of the last "get_string" call).

Writing clients that are also servers

In many applications, processes may need to play both client and server roles. That is, objects in the process may make requests of remote objects on other servers, but may also implement and export objects, requiring that it be able to respond to incoming requests. Details of how to write programs in this peer-to-peer style are explained in section 6.9, "Advanced Topics".

Compiling and linking clients

All client programs must include the header file "somd.h" (or for C++, "somd.xh") in addition to any *"<className>*.h" (or *"<className>*.xh") header files they require from application classes. All DSOM client programs must link to the SOMobjects Toolkit library: "libsomtk.a" on AIX and "somtk.lib" on OS/2 or Windows. For more information, see the topic "Compiling and linking" in Chapter 3, "Using SOM Classes in Client Programs."

6.4 Basic Server Programming

Server programs execute and manage object implementations. That is, they are responsible for:

- Notifying the DSOM daemon that they are ready to begin processing requests,
- Accepting client requests,
- Loading class library DLLs when required,
- Creating/locating/destroying local objects,
- Demarshalling client requests into method invocations on their local objects,
- Marshalling method invocation results into responses to clients, and
- Sending responses back to clients.

As mentioned previously, DSOM provides a simple, "generic" server program that performs all of these tasks. All the server programmer needs to provide are the application class library(ies) DLL that the implementor wants to distribute. Optionally, the programmer can also supply an application-specific server class, derived from **SOMDServer**. (The **SOMDServer** class can be used by default.) The server program does the rest automatically.

The "generic" server program is called **somdsvr** and can be found in `/usr/lpp/som/bin/somdsvr` on AIX and in `%SOMBASE%\bin\somdsvr.exe` on OS/2 or Windows.

Some applications may require additional flexibility or functionality than what is provided by the generic server program. In that case, application-specific server programs can be developed. This section discusses the steps involved in writing such a server program.

To create a server program, a server writer needs to know what services the DSOM run-time environment will provide and how to use those services to perform the duties (listed above) of a server. The DSOM run-time environment provides several key objects that can be used to perform server tasks. These objects and the services they provide will be discussed in this section. Examples showing how to use the run-time objects to write a server are also shown.

Server run-time objects

There are three DSOM run-time objects that are important in a server:

- The server's *implementation definition* (**ImplementationDef**),
- The *SOM Object Adapter* (**SOMOA**), and
- The application-specific *server object* (an instance of either **SOMDServer** or a class derived from **SOMDServer**).

Server implementation definition

A *server's implementation definition* must be registered in the *Implementation Repository* before a server can be used. When a client attempts to invoke a method on a remote object, DSOM consults the Implementation Repository to find the location of the target object's server.

An implementation definition is represented by an object of class **ImplementationDef**, whose attributes describe a server's ID, user-assigned alias, host name, program pathname, the class of its server object, whether or not it is multi-threaded, and so forth. Implementation IDs uniquely identify servers within the Implementation Repository, and are used as keys into the Implementation Repository when retrieving the **ImplementationDef** for a particular server.

It is possible to change the implementation characteristics of a server, even to the point of using a completely different server program on another machine (with Workgroup DSOM). Thus, the implementation ID identifies a *logical server*, and the **ImplementationDef** describes the current implementation of that logical server.

See the topic "Registering Servers and Classes" in section 6.6 for details on server registration. Two registration methods are described: "manual" (via the **regimpl**, the **wregimpl**, or the **pregimpl** utility), and "programmatic," via **ImplRepository** methods.

When a server is initialized, it must retrieve a copy of its **ImplementationDef**, and keep it in a global variable (**SOMD_ImplDefObject**). <u>This variable is used by the DSOM run time</u>. (Client-only programs may leave the **SOMD_ImplDefObject** variable set to NULL.)

SOM Object Adapter (SOMOA)

The *SOM Object Adapter* (**SOMOA**) is the main interface between the server application and the DSOM run time. The **SOMOA** is responsible for most of the server duties listed at the beginning of this section. In particular, the **SOMOA** object handles all communications and interpretation of inbound requests and outbound results. When clients send requests to a server, the requests are received and processed by the **SOMOA**.

The **SOMOA** works together with the server object to create and resolve DSOM references to local objects, and dispatch methods on objects.

There is one **SOMOA** object per server process. (The **SOMOA** class is implemented as a *single instance* class.)

Server object

Each server process contains a single *server object*, which has the following responsibilities for managing objects in the server:

*   Provides an interface to *client applications* for basic object creation and destruction services, as well as any other application-specific object-management services that may be required by clients. For example, a print server may have a method that returns a list of all printers managed by that server. Clients may call this method to find out what printers are available.

*   Provides an interface to the *SOM Object Adapter* for support in the creation and management of DSOM object references (which are used to identify an object in the server), and for dispatching requests.

The server class, **SOMDServer**, defines the base interface that *must* be supported by any server object. In addition, **SOMDServer** provides a default implementation that is suited to managing transient SOM objects in a server. This section will show how an application might override the basic **SOMDServer** methods and introduce new methods in order to tailor the server object functionality to a particular application.

Server activation

Server programs may be activated either

*   *Automatically* by the DSOM daemon, **somdd**, or

*   *Manually* via command line invocation, or under application control.

When a server is activated automatically by **somdd**, it will be passed a single argument (in argv[1]) that is the *implementation ID* assigned to the server implementation when it was registered into the Implementation Repository (discussed above and in section 6.6, "Configuring DSOM Applications"). This is useful when the server program cannot know until activation which "logical" server it is implementing. (This is true for the generic server provided with DSOM.) The implementation ID is used by the server to retrieve its **ImplementationDef** from the Implementation Repository.

A server that was not activated by **somdd** may obtain its **ImplementationDef** from the Implementation Repository in any manner that is convenient: by ID, by alias, and so forth. Moreover, a server may choose to "register itself" dynamically, as part of its initialization. To do so, the server would use the programmatic interface to the Implementation Repository.

For example, suppose that the server program "myserver" was designed so that it could be activated either automatically or manually. This requires that it be written to expect the implementation ID as its first argument, and to use that argument to retrieve its **ImplementationDef** from the Implementation Repository. If an application defines a server in the Implementation Repository whose implementation ID is `2bcdc4f2-0f62f780-7f-00-10005aa8afdc,` then "myserver" could be run as that server by invoking the following command:

```
myserver 2bcdc4f2-0f62f780-7f-00-10005aa8afdc
```

AIX users should be aware that, unless the SetUserID mode bit is set on the file containing the server program, the UID for the server process will be inherited from the **somdd** process. To set the SetUserID mode bit from the AIX command line, type one of the following commands:

chmod 4000 <filename>   – or –
chmod u+s <filename>

where "<filename>" denotes the name of the file containing the server program. For additional details, see the "chmod" command in InfoExplorer or consult the man pages.

Initializing a server program

Initializing the DSOM run-time environment

The first thing the server program should do is to initialize the DSOM run time by calling the **SOMD_Init** function. This causes the various DSOM run-time objects to be created and initialized, including the Implementation Repository (accessible via the global variable **SOMD_ImplRepObject**), which is used in the next initialization step.

Initializing the server's ImplementationDef

Next, the server program is responsible for initializing its **ImplementationDef**, referred to by the global variable **SOMD_ImplDefObject**. It is initialized to NULL by **SOMD_Init**. (For client programs it should be left as NULL.) If the server implementation was registered with the Implementation Repository before the server program was activated (as will be the case for all servers that are activated automatically by **somdd**), then the **ImplementationDef** can be retrieved from the Implementation Repository. Otherwise, the server program can register its implementation with the Implementation Repository dynamically (as shown in section 6.6, "Configuring DSOM applications").

The server can retrieve its **ImplementationDef** from the Implementation Repository by invoking the **find_impldef** method on **SOMD_ImplRepObject**. It supplies, as a key, the implementation ID of the desired **ImplementationDef**.

The following code shows how a server program might initialize the DSOM run-time environment and retrieve its **ImplementationDef** from the Implementation Repository.

```
#include <somd.h> /* needed by all servers */
main(int argc, char **argv)
{
    Environment ev;
    SOM_InitEnvironment(&ev);

/* Initialize the DSOM run-time environment */
    SOMD_Init(&ev);

/* Retrieve its ImplementationDef from the Implementation
    Repository by passing its implementation ID as a key */
    SOMD_ImplDefObject =
        _find_impldef(SOMD_ImplRepObject, &ev, argv[1]);
...
}
```

Initializing the SOM Object Adapter

The next step the server must take before it is ready to accept and process requests from clients is to create a **SOMOA** object and initialize the global variable **SOMD_SOMOAObject** to point to it. This is accomplished by the assignment:

```
SOMD_SOMOAObject = SOMOANew();
```

Note: The **SOMOA** object is not created automatically by **SOMD_Init** because it is only required by server processes.

After the global variables have been initialized, the server can do any application-specific initialization required before processing requests from clients. Finally, when the server is ready to process requests, it must call the **impl_is_ready** method on the **SOMOA**:

```
_impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
```

The **SOMOA** will then set up a communications port for incoming messages, which it registers with the DSOM daemon. Once the DSOM daemon has been notified of the server's port, it assists client applications in "binding" (that is, establishing a connection) to that server.

The **impl_is_ready** method also causes the server object, whose class is defined in the server's **ImplementationDef**, to be created. The server object can be referenced through the global variable, **SOMD_ServerObject**.

When initialization fails

It is possible that a server will encounter some error when initializing itself. Servers must attempt to notify DSOM that their activation failed, using the **activate_impl_failed** method. This method is called as follows:

```
/* tell the daemon (via SOMOA) that activation failed */
_activate_impl_failed(SOMD_SOMOAObject,&ev, SOMD_ImplDefObject, rc);
```

Server writers should be aware, however, that until the server's **SOMD_ImplDefObject** has been initialized, it is not possible to call the **_activate_impl_failed** method on the DSOM daemon.

Note: A server program should *not* call **activate_impl_failed** once it has called **impl_is_ready**.

Processing requests

The **SOMOA** is the object in the DSOM run-time environment that receives client requests and transforms them into method calls on local server objects. In order for **SOMOA** to listen for a request, the server program must invoke one of two methods on **SOMD_SOMOAObject.** If the server program wishes to turn control over to **SOMD_SOMOAObject** completely (that is, effectively have **SOMD_SOMOAObject** go into an infinite request-processing loop), then it invokes the **execute_request_loop** method on **SOMD_SOMOAObject** as follows:

```
rc = _execute_request_loop(SOMD_SOMOAObject, &ev, SOMD_WAIT);
```

Note: This is the way the DSOM-provided "generic" server program interacts with **SOMD_SOMOAObject**.

The **execute_request_loop** method takes an input parameter of type **Flags**. The value of this parameter should be either SOMD_WAIT or SOMD_NO_WAIT. If SOMD_WAIT is passed as argument, **execute_request_loop** will return only when an error occurs. If SOMD_NO_WAIT is passed, it will return when there are no more outstanding messages to be processed. SOMD_NO_WAIT is usually used when the server is being used with the event manager. See the topic "Peer vs. client-server processes" in section 6.9, "Advanced Topics," for more details.

If the server wishes to incorporate additional processing between request executions, it can invoke the **execute_next_request** method to receive and execute requests one at a time:

```
for(;;) {
    rc = _execute_next_request(SOMD_SOMOAObject, &ev, SOMD_NO_WAIT);
    /* perform app-specific code between messages here, e.g., */
        if (!rc) numMessagesProcessed++;
}
```

Just like **execute_request_loop**, **execute_next_request** has a **Flags** argument that can take one of two values: SOMD_WAIT or SOMD_NO_WAIT. If **execute_next_request** is invoked with the SOMD_NO_WAIT flag and no message is available, the method returns immediately with a return code of SOMDERROR_NoMessages. If a request is present, it will execute it. Thus, it is possible to "poll" for incoming requests using the SOMD_NO_WAIT flag.

Exiting a server program

When a server program exits, it should notify the DSOM run time that it is no longer accepting requests. This should be done whether the program exits normally, or as the result of an error. If this is not done, **somdd** will continue to think that the server program is active, allowing clients to attempt to connect to it, as well as preventing a new copy of that server from being activated.

To notify DSOM when the server program is exiting, the **deactivate_impl** method defined on **SOMOA** should be called. For example,

```
/* tell DSOM (via SOMOA) that server is now terminating */
   _deactivate_impl(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
```

Note: For robustness, it would be worthwhile to add appropriate "exit handlers" or "signal handlers" to your application servers that call the **deactivate_impl** method upon abnormal program termination. This ensures that the DSOM daemon is made aware of the server's termination, so that client connections are no longer allowed.

Finally, at the end of a server program, the **SOMD_Uninit** procedure must be called to free DSOM run-time objects, and to release semaphores, shared memory segments, and any other system resources.

For example, the exit code in the server program might look like this:

```
    ...
    SOMD_Uninit(&ev);
    SOM_UninitEnvironment(&ev);
}
```

Observe also the **SOM_UninitEnvironment** call, which frees any memory associated with the specified **Environment** structure.

When a Windows DSOM server application receives a WM_QUIT message while processing incoming requests, the **execute_request_loop** method will return SOMDERROR_WMQUIT. When the server receives SOMDERROR_WMQUIT, it should perform its usual clean up and termination procedures.

Managing objects in the server

Object references, ReferenceData, and the ReferenceData table

One of **SOMOA**'s responsibilities is to support the creation of object references (**SOMDObject**s). Recall from the "Stack" example discussion (in Section 6.2) that an *object reference* is an exportable "handle" to an object and that proxies are examples of object references. The **SOMOA** interface supports three operations for creating object references: **create**, **create_constant**, and **create_SOM_ref**.

The **create** and **create_constant** methods allow a server to associate application-specific data about an object with an object reference for that object. This data, called *reference data,* is represented in a sequence of up to 1024 bytes of information about the object. This sequence, defined by the type **ReferenceData**, may contain the object's location, state, or any other characteristics meaningful to the application. Usually, **ReferenceData** is used by a server process to locate or activate an object in the server. **ReferenceData**, and hence the methods **create** and **create_constant**, are usually only used in connection with persistent objects (objects whose lifetimes exceed that of the process that created them).

The **create** method differs from the **create_constant** method in the following way: **ReferenceData** associated with an object reference constructed by **create_constant** is immutable whereas the the **ReferenceData** associated with an object reference created by **create** can be changed (via the **change_id** method). References created with **create_constant** return true when the method **is_constant** is invoked on them.

The **create** method stores the **ReferenceData** in a *ReferenceData table* associated with the server, while **create_constant** maintains the **ReferenceData** as a constant part of the object reference. The **ReferenceData** associated with an object reference (whether it was constructed using **create** or **create_constant**) can be retrieved via the **SOMOA** method **get_id**.

The IDL **SOMOA** interface declarations of **create**, **create_constant, get_id**, and **change_id**, and the **SOMDObject** interface declaration of **is_constant** are presented below.

```
/* From the SOMOA interface */

    sequence <octet,1024> Referencedata;
    SOMDObject create(in ReferenceData id, in InterfaceDef intf,
                      in ImplementationDef impl);

    SOMDObject create_constant(in ReferenceData id,
                               in InterfaceDef intf,
                               in ImplementationDef impl);

    ReferenceData get_id(in SOMDObject objref);

    void change_id(in SOMDObject objref, in ReferenceData id);

/* From the SOMDObject interface */

    boolean is_constant();
```

An example of how **ReferenceData** can be used by an application follows the description of **SOMDServer** objects in the next section.

Simple SOM object references

In order to efficiently support the generation and interpretation of references to SOM objects, the **SOMOA** defines another method called **create_SOM_ref**.

The **create_SOM_ref** method creates a simple DSOM reference (**SOMDObject**) for a local SOM object. The reference is "special" in that, unlike **create** and **create_constant**, there is no user-supplied **ReferenceData** associated with the object and because the reference is only valid while the SOM object exists in memory. The **SOMObject** to which it refers can be retrieved via the **get_SOM_object** method. The **is_SOM_ref** method can be used to tell if the reference was created using **create_SOM_ref** or not. The IDL declarations for **create_SOM_ref**, **get_SOM_object**, and **is_SOM_ref** are displayed below:

```
/* from SOMOA's interface */

    SOMDObject create_SOM_ref(in SOMObject somobj,
                              in ImplementationDef impl);

    SOMObject get_SOM_object(in SOMDObject somref);

/* from SOMDObject's interface */

    boolean is_SOM_ref();
```

SOMDServer (default server-object class)

Every server has a server object that implements three kinds of activities:

- Creation and destruction of SOM objects
- Mapping between **SOMObject**s and **SOMDObject**s, and
- Dispatching methods on SOM objects

Additional, application-specific server methods (for initialization, server control, etc.) can be defined in a subclass of the **SOMDServer** class. The class of the server object to be used with a server is contained in the server's **ImplementationDef**.

Following are the IDL declarations of the **SOMDServer** operations:

```
// methods called by a client

    SOMObject somdCreateObj(in Identifier objclass, in string hints);

    void somdDeleteObj(in SOMObject somobj);

    SOMClass somdGetClassObj(in Identifier objclass);

// methods called by SOMOA

    SOMDObject somdRefFromSOMObj(in SOMObject somobj);

    SOMObject somdSOMObjFromRef(in SOMDObject objref);

    void somdDispatchMethod(in SOMObject somobj,
                            out somToken retValue,
                            in somId methodId,
                            in va_list ap);
```

Creation and destruction of SOM objects

The **SOMDServer** class defines methods for the basic creation of SOM objects in the server process (**somdCreateObj**), and for finding the SOM class object for a specified class (**somdGetClassObj**). With **somdGetClassObj**, a client can get a proxy to a class object on the server, so that methods introduced in the class's metaclass (for example, class-specific constructors, etc.) may be invoked directly on the class object. Examples of client use of these two methods were presented earlier in Sections 6.2 and 6.3.

With **somdDeleteObj**, the client can involve the server object in object destruction. (The methods **somdTargetFree** and **somFree** are defined on the objects themselves and do not involve the server object.) Involving the server object in object creation and destruction can be important for applications that need more control over how objects are created and destroyed, or if the application needs to keep track of an object's creation and destruction.

Mapping objects to object references

SOMDServer also defines methods that implement mappings between **SOMObject**s and **SOMDObject**s (object references) and a method for dispatching method calls on SOM objects. These methods are used by the SOM Object Adapter (**SOMOA**) when converting remote requests into method calls and results into responses.

Recall from the topic "Proxy objects" in Section 6.3, "Basic Client Programming", that servers return *proxies* to remote objects as method results, not the remote objects themselves. Recall also that class libraries need not be designed to be distributed (that is, the code that implements the classes need not be aware of the existence of proxy objects at all). Thus, it is up to the DSOM run-time environment to ensure that proxies, rather than remote objects, are returned to clients. The **SOMD_SOMOAObject** and **SOMD_ServerObject** work together to perform this service. Whenever a result from a remote method call includes a **SOMObject**, the **SOMD_SOMOAObject** invokes the **somdRefFromSOMObj** method on **SOMD_ServerObject**, asking it to create a **SOMDObject** from the **SOMObject**.

The default implementation (or **SOMDServer**'s implementation) for **somdRefFromSOMObj** uses the **create_SOM_ref** method to return a "simple" reference for the **SOMObject**. Application-specific server objects (instances of a subclass of **SOMDServer**) may elect to use **create** or **create_constant** to construct the object reference if the application requires **ReferenceData** to be stored.

Hints on the use of create vs. create_constant

Enough context now exists so that the following question may be answered: "If object references constructed with **create** support changeable **ReferenceData,** but object references constructed with **create_constant** do not, why would I ever want to use **create_constant**?"

Invocations of **create** add entries to a table called the *ReferenceData Table*. The ReferenceData Table is persistent; that is, **ReferenceData** saved in it persists between server activations.

Two calls to **create** with the same arguments do not return the same **SOMDObject** (per CORBA 1.1 specifications). That is, if **create** is called twice with the same arguments, two entries in the ReferenceData Table will be created. If a server using **create** wishes to avoid cluttering up the ReferenceData Table with multiple references to the same object, it must maintain a table of its own to keep track of the references it has created to avoid calling **create** twice with the same arguments.

The **create_constant** method stores the **ReferenceData** as part of the **SOMDObject**'s state; that is, it does not add entries to the ReferenceData Table. The **create_constant** method, then, might be used by a server that does not want to maintain a table of references nor pay the penalty of cluttering up the ReferenceData Table with multiple entries.

Mapping object references to objects

The **somdSOMObjFromRef** method maps **SOMDObject**s to **SOMObject**s. This method is invoked by **SOMOA** on the server object**,** for each object reference found as a parameter in a request.

The default implementation for **somdSOMObjFromRef** returns the address of the **SOMObject** for which the specified object reference was created (using the **somdRefFromSOMObj** method). If the object reference was not created by the same server process, then an exception (BadObjref) is raised. The default implementation does not, however, verify that the original object (for which the object reference was created) still exists. If the original object has been deleted (for example, by another client program), then the address returned will not represent a valid object, and any methods invoked on that object pointer will result in server failure. Note: The default implementation of **somdSOMObjFromRef** does not check that the original object address is still valid because the check is very expensive and seriously degrades server performance.

To have a server verify that all results from **somdSOMObjFromRef** represent valid objects, server programmers can subclass **SOMDServer** and override the **somdSOMObjFromRef** method to perform a validity check on the result (using the **somIsObj** function). For example, a subclass "MySOMDServer" of **SOMDServer** could implement the **somdSOMObjFromRef** method as follows:

```
SOM_Scope SOMObject SOMLINK somdSOMObjFromRef(MySOMDServer somSelf,
                                              Environment * ev,
                                              SOMDObject objref)
{
    SOMObject obj;
    StExcep_INV_OBJREF *ex;


    /* MySOMDServerData *somThis = MySOMDServerGetData(somSelf); */
    MySOMDServerMethodDebug("MySOMDServer", "somdSOMObjFromRef");


    obj = MySOMDServer_parent_SOMDServer_somdSOMObjFromRef(somSelf,
                                              ev, objref);
    if (somIsObj(obj))
        return (obj);
    else {
        ex = (StExcep_INV_OBJREF *)
                SOMMalloc(sizeof(StExcep_INV_OBJREF));
        ex->minor = SOMDERROR_BadObjref;
        ex->completed = NO;
        somSetException(ev, USER_EXCEPTION,
                        ex_StExcep_INV_OBJREF, ex);
        return (NULL);
    }
}
```

Dispatching a method

After **SOMOA** (with the help of the local server object) has resolved all the **SOMDObject**s present in a request, it is ready to invoke the specified method on the target. Rather than invoking **somDispatch** *directly* on the target, it calls the **somdDispatchMethod** method on the server object. The parameters to **somdDispatchMethod** are the same as the parameters for **SOMObject::somDispatch** (see the *SOMobjects Developer Toolkit Programmers Reference Manua*l for a complete description).

The default implementation for **somdDispatchMethod** in **SOMDServer** simply invokes **SOMObject::somDispatch** on the specified target object with the supplied arguments. The reason for this indirection through the server object is to give the server object a chance to intercept method calls coming into the server process, if desired.

Example: Writing a persistent object server

This section shows an example of how to provide a server class implementation for persistent SOM objects. (The Persistence Framework of the full-capability SOMobjects Developer Toolkit can be used to write a persistent object server; an example of that type is given in the *SOMobjects Developer Toolkit Users Guide.*) All of the persistent object management is contained in the server class; this class can be used with the DSOM "generic" server program, **somdsvr**.

The following example describes a user-supplied server class "MyPServer" that is derived from **SOMDServer.** The "MyPServer" class introduces five new methods:

isPObj
assignRefDataToPObj
deletePObj
getRefDataFromPObj and
activatePObjFromRefData

and overrides four **SOMDServer** methods:

somdCreateObj
somdDeleteObj
somdRefFromSOMObj and
somdSOMObjFromRef.

The example shows how a server class might use and manage **ReferenceData** in object references to find and activate persistent objects.

The IDL specification for "MyPServer" follows:

```
interface MyPServer : SOMDServer {
    boolean isPObj (in SOMObject obj);
    void assignRefDataToPObj(in SOMObject pobj);
    void deletePObj(in SOMObject pobj);
    ReferenceData getRefDataFromPObj(in SOMObject pobj);
    SOMObject activatePObjFromRefData(in ReferenceData rd);
  #ifdef __SOMIDL__
    implementation {
        somdCreateObj     : override;
        somdDeleteObj     : override;
        somdRefFromSOMObj : override;
        somdSOMObjFromRef : override:
    };
  #endif
};
```

The "isPObj" method returns TRUE if the object passed to it is a persistent object. It is implemented as follows:

```
SOM_Scope boolean SOMLINK
isPObj(MyPServer somSelf, Environment *ev, SOMObject obj) {
    return(obj && _somIsA(obj, MyPersistentObjectNewClass(0, 0));
}
```

The following two procedures override **SOMDServer**'s implementations of **somdCreateObj** and **somdDeleteObj**.

```
SOM_Scope SOMObject SOMLINK
              somdCreateObj(MyPServer somSelf, Environment *ev,
              Identifier objclass, string hints)
{
   /* create the object as usual */
   SOMObject obj =
      parent_somdCreateObj(somSelf, ev, objclass, hints);
   /* if obj is persistent, assign Ref Data to it */
   if (_isPObj(somSelf, ev, obj))) {
      _assignRefDataToPObj(somSelf, ev, obj)
   }
   return(obj);
}
```

The implementation of **somdCreateObj** first creates the object as usual by employing the implementation of **SOMDServer** (MyPServer's parent). If the newly created object is persistent, the job of "assignRefDataToPObj" is to associate with the object a piece of data that (1) identifies the object, (2) is retrievable from the object, and (3) can be coerced into **Reference-Data** so that it can be used to create a **SOMDObject** (an object reference).

```
SOM_Scope void SOMLINK
    somdDeleteObj(MyPServer somSelf, Environment *ev, SOMObject obj)
{
   /* is obj persistent, have the persistence framework delete it */
   if (_isPObj(somSelf, ev, obj)) {
      _deletePObj(somSelf, ev, obj);
   } else /* obj is not persistent, so delete as usual */
       parent_somdDeleteObj(somSelf, ev, obj);
}
```

The **somdDeleteObj** implementation, when the object to be deleted is persistent, invokes "deletePObj" to delete the object. When the object is not persistent, the **SOMDServer** implementation of **somdDeleteObj** deletes the object.

The following two procedures override **SOMDServer**'s implementations of the methods **somdRefFromSOMObj** and **somdSOMObjFromRef**:

```
SOM_Scope SOMDObject SOMLINK
              somdRefFromSOMObj(MyPServer somSelf, Environment *ev,
              SOMObject obj)
{
  SOMDObject objref;

  /* is obj persistent */
  if (_isPObj(somSelf, ev, obj {
    /* Create an object reference based on identifying data. */
    ReferenceData rd = _getRefDataFromPObj(somSelf, ev, obj);
    InterfaceDef intf =
        _lookup_id(SOM_InterfaceRepository,ev,somGetClassName(obj));
    objref = _create_constant(SOMD_SOMOAObject, ev, &rd, intf,
                                SOMD_ImplDefObject);
    _somFree(intf);
    SOMFree(rd._buffer);
  } else /* obj is not persistent, so get Ref in usual way */
     objref = parent_somdRefFromSOMObj(somSelf, ev, obj);
  return(objref);
}
```

Method **somdRefFromSOMObj** is responsible for producing a **SOMDObject** (the "Ref" in **somdRefFromSOMObj**) from a **SOMObject**. As mentioned earlier, **SOMOA** exports two methods for creating **SOMDObject**s: **create** and **create_constant**. This implementation uses **create_constant** because it does not want to store the **ReferenceData** in the ReferenceData Table. If it did use **create** and store the **ReferenceData** in the persistent table, the server object would either (1) have to keep a persistent table that maps **SOMObject**s to **SOMDObject**s so that it didn't call **create** twice with the same arguments (recall that **create** always returns a new **SOMDObject** even when called twice with the same arguments), or (2) fill up the ReferenceData table with **SOMDObject**s that contain the same **ReferenceData**.

The prerequisites for asking **SOMOA** to create a **SOMDObject** are (1) some **ReferenceData** to be associated with the **SOMDObject**, (2) an **InterfaceDef** that describes the interface of the object, and (3) an **ImplementationDef** that describes the object's implementation. The **InterfaceDef** is retrieved from the SOM Interface Repository using the object's class name as key. The **ImplementationDef** is held in the variable **SOMD_ImplDefObject** that is set when the server process is initialized. The "MyPServer" method "getRefDataFromPObj" is used to retrieve the identifying data from the object and coerce it into **ReferenceData**. With these three arguments, **SOMOA**'s **create_constant** is called to create the **SOMDObject**.

```
SOM_Scope SOMObject SOMLINK
            somdSOMObjFromRef(MyPServer somSelf, Environment *ev,
            SOMDObject objref)
{   SOMObject obj;

    /* test if objref is mine */
    if (_is_constant(objref, ev)) {
      /* objref was mine, activate persistent object myself */
      ReferenceData rd = _get_id(SOMD_SOMOAObject, ev, objref);
      obj = _activatePObjFromRefData(somSelf, ev, &rd);
      SOMFree(rd._buffer);
    } else
      /* it's not one of mine, let parent activate object */
      obj = parent_somdSOMObjFromRef(somSelf, ev, objref);
    return obj;
}
```

This implementation of **somdSOMObjFromRef** is a little different from the others in that the server object must determine whether the **SOMDObject** is one that it created (that is, one that represents a persistent object), or is just a **SOMDObject** that was created by the **SOMDServer** code (its parent). This is done by asking the **SOMDObject** if it is a "constant" object reference (that is, one created by **create_constant**). If the **SOMDObject** says that it is a "constant", then the "MyPServer" may safely assume that the **SOMDObject** represents a persistent object that it created. If the **SOMDObject** is determined to represent a persistent object, then its **ReferenceData** is used to locate/activate the object it represents (via the method "activatePObjFromRefData").

Identifying the source of a request

CORBA 1.1 specifies that a Basic Object Adapter should provide a facility for identifying the *principal* (or user) on whose behalf a request is being performed. The **get_principal** method, defined by **BOA** and implemented by **SOMOA**, returns a **Principal** object, which identifies the caller of a particular method. From this information, an application can perform access control checking.

In CORBA 1.1, the interface to **Principal** is not defined, and is left up to the ORB implementation. In the current release of DSOM, a **Principal** object is defined to have two attributes:

userName (string)

— Identifies the name of the user who invoked a request.

hostName (string)

— Identifies the name of the host from which the request originated.

Currently, the value of the **userName** attribute is obtained from the USER environment variable in the calling process. Likewise, the **hostName** attribute is obtained from the HOSTNAME environment variable. This facility is intended to provide basic information about the source of a request, and currently, is *not* based on any specific authentication (or "login") scheme. More rigorous authentication and security mechanisms will be considered for future DSOM implementations.

The IDL prototype for the **get_principal** method, defined on **BOA** (**SOMOA**) is as follows:

```
Principal  get_principal (in SOMDObject obj,
                          in Environment *req_ev);
```

This call will typically be made either by the target object or by the server object, when a method call is received. The **get_principal** method uses the Environment structure associated with the request, and an object reference for the target object, to produce a **Principal** object that defines the request initiator.

Note: CORBA 1.1 defines a "tk_Principal" **TypeCode** which is used to identify the type of **Principal** object arguments in requests, in case special handling is needed when building the request. Currently, DSOM does not provide any special handling of objects of type "tk_Principal"; they are treated like any other object.

Compiling and linking servers

The server program must include the "somd.h" header file. Server programs must link to the SOMobjects Toolkit library: "libsomtk.a" on AIX, and "somtk.lib" on OS/2 or Windows.

For more information, see the topic "Compiling and linking" in Chapter 5, "Implementing Classes in SOM."

6.5 Implementing Classes

DSOM has been designed to work with a wide range of object implementations, including SOM class libraries as well as non-SOM object implementations. This section describes the necessary steps in using SOM classes or non-SOM object implementations with DSOM.

Using SOM class libraries

It is quite easy to use SOM classes in multi-process DSOM-based applications as exemplified by the sample DSOM application presented in section 6.2, "A Simple DSOM Example". In fact, in many cases, existing SOM class libraries may be used in DSOM applications *without* requiring any special coding or recoding for distribution. This is possible through the use of DSOM's *generic server program*, which uses SOM and the *SOM Object Adapter* (**SOMOA**) to load SOM class libraries on demand, whenever an object of a particular class is created or activated.

The topic "Registering servers and classes" in section 6.6 "Configuring DSOM Applications" discusses how to register a server implementation consisting of a DSOM generic server process and one or more SOM class libraries.

Role of DSOM generic server program

The generic server program provides basic server functionality: it continuously receives and executes requests (via an invocation of the **SOMOA's execute_request_loop** method), until the server is stopped. Some requests result in the creation of SOM objects; the generic server program will find and load the DLL for the object's class automatically, if it has not already been loaded.

When generic server program functionality is not sufficient for the particular application, application-specific server programs can be developed. For example, some applications may want to interact with a user or I/O device between requests. The previous section, entitled "Basic Server Programming," discussed the steps involved in writing a server program.

Role of SOM Object Adapter

The SOM Object Adapter is DSOM's standard object adapter. It provides basic support for receiving and dispatching requests on objects. As an added feature, the **SOMOA** and the server process's server object collaborate to automate the task of converting SOM object pointers into DSOM object references, and vice versa. That is, whenever an object pointer is passed as an argument to a method, the **SOMOA** and the server object convert the pointer to a DSOM object reference (since a pointer to an object is meaningless outside the object's address space).

Role of SOMDServer

The server process's server object (whose default class is **SOMDServer**) is responsible for creating/destroying objects on the server via **somdCreateObj**, **somdGetClassObj**, and **somdDeleteObj**, for mapping between object references (**SOMDObject**s) and **SOMObject**s via **somdRefFromSOMObj** and **somdSOMObjFromRef**, and for dispatching remote requests to server process objects via **somdDispatchMethod**. These last three methods are invoked on the server object by the **SOMOA** when objects are to be returned to clients, when incoming requests contain object references, and when the method is ready to be dispatched, respectively. By partitioning out these mapping and dispatching functions into the server object, the application can more easily customize them, without having to build object adapter subclasses.

SOMDServer can be subclassed by applications that want to manage object location, object activation, and method dispatching. An example of such an application (which provides a server class implementation for persistent SOM objects) is shown in section 6.4, "Basic Server Programming."

These features of **SOMOA** and **SOMDServer** make it possible to take existing SOM classes, which have been written for a single-address space environment, and use them unchanged in a DSOM application. More information on the **SOMOA** and server objects can be found in the "Basic Server Programming" section.

Implementation constraints

The generic server program (**somdsvr**), the **SOMOA**, and the **SOMDServer** make it easy to use SOM classes with DSOM. However, if there are any parts of the class implementation that were written expecting a single-process environment, the class may have to be modified to behave properly in a client-server environment. Some common implementation practices to *avoid* are listed below:

- **Printing to standard output.** Any text printed by a method will appear at the server, as opposed to the client. In fact, the server may not be attached to a text display device or window, so the text may be lost completely. It is preferred that any textual output generated by a method be returned as an output string.

  Note: Passing textual output between the client program and the called method via an "inout string" parameter is *strongly* discouraged. As discussed in the CORBA 1.1 specification (page 94), the size of the output string is constrained by the size of the input string. If there was no input string value, the size of the output string would be constrained to 0 bytes. Instead, it is preferred that textual data be returned either as an output string (DSOM provides the storage), or by passing a character array buffer (client provides the storage).

- **Creating and deleting objects.** Methods that create or delete objects may have to be modified if the created objects are intended to be remote. The calls to create local objects are different than the calls to create remote objects.

- **Using pointers to client-allocated memory in instance variables.** Consider the following example: A class has a method that accepts a pointer to a data value created by the client (for example, a string or a struct), and simply stores the pointer in an instance variable or attribute. However, in DSOM, the called method is passed a pointer to a *copy* of the value (in the request message body), but the copy is freed at the end of the request. If the data value is meant to persist between requests, the object is responsible for making its own copy of it. (The implementation of the "_set_printerName" method in the topic "Wrapping a printer API" later in this section is an example of a method performing such a copy.)

- **Using "procedure" methods.** Methods having the **procedure** SOM IDL modifier cannot be invoked remotely using DSOM. This is because these "methods" are called directly, rather than via the normal method resolution mechanisms on which DSOM relies.

In addition to those coding practices which simply do not "port" to a distributed environment, there are a few other restrictions that are imposed by DSOM's (current) implementation.

- **Using parameter types not supported by DSOM.** DSOM can make remote invocations only of methods whose parameter types are among the following IDL types: basic types (**short, long, unsigned short, unsigned long, float, double, char, boolean, octet**), **enum, struct, union, sequence, string, array, any,** and **object** (an interface name, designating a pointer to an object that supports that interface). The members of a **struct, union, sequence,** or **array**, and the value of an **any**, must also be from the above list of supported DSOM types.

  In addition to the above types, DSOM also supports method parameters of type **pointer** to one of the above types (for example, **long***). Pointers to pointers are not supported, however, and pointers embedded within one of the above types (for example, a pointer within a struct) are not supported. The "**void ***" type is also not supported. Currently, DSOM has the limitation that NULL pointer values cannot be returned as **inout** or **out** method arguments, although it is expected that this limitation will be addressed in a future release.

  Types declared as SOMFOREIGN types are not currently supported by DSOM.

- **Packing of structures used as method arguments.** If a compiler option is used to pack or optimize storage of **structs** (including reordering of struct members) or **unions**, it is important to indicate the exact alignment of the structures using *alignment modifiers* expressed in the implementation section of the IDL file. This information must then be updated in the Interface Repository. See the topic "Providing 'alignment' information" in Chapter 7, "The Interface Repository Framework."

Some applications may need to associate specific identification information with an object, to support application-specific object location or activation. In that case, an application server should create object references explicitly, using the **create** or **create_constant** method in **SOMOA**. A logical place to put these calls is in a subclass of **SOMDServer,** as it is the server object that is responsible for producing/activating objects from object references.

Using other object implementations

As an Object Request Broker, DSOM must support a wide range of object implementations, including non-SOM implementations. For example, in a print spooler application, the implementation of a print queue object may be provided by the operating system, where the methods on the print queue are executable programs or system commands. As another example, consider an application that uses a large, existing class library that is not implemented using SOM. Finally, consider a class library where persistence is implemented by something other than the Persistence Framework.

In each of these examples, the application must *participate* in object identification, activation, initialization, and request dispatching. Each server supplies a server object (derived from **SOMDServer**) that works in conjunction with the **SOMOA** for this purpose.

Wrapping a printer API

Presented below is a simple example showing how an existing API could be "wrapped" as SOM objects. The API is admittedly trivial, but it is hoped that readers understand this simple example well enough to create more sophisticated applications of their own.

The "API" wrapped in this example is comprised of two OS/2 system calls. The first one asks for a file to be printed on a specific printer:

```
print /D:<printerName> <filename>
```

The second one asks for the file currently being printed on device *<printerName>* to be cancelled.

```
print /D:<printerName> /C
```

Two IDL interfaces are declared in the module "PrinterModule": "Printer" and "PrinterServer". The "Printer" interface wraps the two system calls. The "PrinterServer" interface describes a subclass of **SOMDServer**. "PrinterModule::PrinterServer" will be the class of the server object in the print-server application.

```
#include <somdserv.idl>

module PrinterModule {
  interface Printer : SOMObject {
    attribute string printerName;
    void print(in string fname);
    void cancel();
    #ifdef __SOMIDL__
    implementation {
      printerName: noset;
      };
    #endif
  };

  interface PrinterServer : SOMDServer {
    #ifdef __SOMIDL__
    implementation {
      somdCreateObj: override;
      somdRefFromSOMObj: override;
      somdSOMObjFromRef: override;
      };
    #endif
  };

};
```

Note that the "Printer" interface defines one attribute, "printerName", that will be used to identify the printer. It will be set when a "Printer" is created. Printer's two operations, "print" and "cancel", correspond to the two system commands the interface is encapsulating. The "PrinterServer" interface does not introduce any new attributes or operations. It does specify that three of **SOMDServer**'s methods will have their implementations overridden. The three method procedures below show how the "Printer" interface is implemented for the "_set_printerName", "print", and "cancel" methods.

```
SOM_Scope void  SOMLINK PrinterModule_Printer_set_printerName(
 PrinterModule_Printer somSelf, Environment *ev, string printerName)
{
    PrinterModule_PrinterData *somThis =
      PrinterModule_PrinterGetData(somSelf);

    if (_printerName) SOMFree(_printerName);
    _printerName = (string)SOMMalloc(strlen(printerName) + 1);
    strcpy(_printerName, printerName);
}


SOM_Scope void  SOMLINK PrinterModule_Printerprint(
 PrinterModule_Printer somSelf, Environment *ev, string fname)
{
    long rc;
    PrinterModule_PrinterData *somThis =
      PrinterModule_PrinterGetData(somSelf);
    string printCommand = (string)
      SOMMalloc(strlen(_printerName) + strlen(fname) + 10 + 1);

    sprintf(printCommand,"print /D:%s %s",_printerName,fname);
    rc = system(printCommand);
    if (rc) raiseException(ev,rc);
}


SOM_Scope void  SOMLINK PrinterModule_Printercancel(
 PrinterModule_Printer somSelf, Environment *ev)
{
    long rc;
    PrinterModule_PrinterData *somThis =
      PrinterModule_PrinterGetData(somSelf);
    string printCommand =
      (string) SOMMalloc(strlen(_printerName) + 12 + 1);

    sprintf(printCommand,"print /D:%s /C",_printerName);
    rc = system(printCommand);
    if (rc) raiseExeception(ev,rc);
}
```

Note: The implementation of the "raiseException" procedure shown in the example above must be provided by the application. However, it is not shown in this example.

The three method procedures that implement the "PrinterServer" interface's three overridden methods of **SOMDServer** are very similar to the method procedures of the "MyPServer" server-object class presented in the previous section (6.4), and therefore have not been shown.

Parameter memory management

There are five SOM IDL modifiers available for specifying the memory-management policy for the parameters of a method (regardless of whether the caller or the object owns the parameters' memory after the method is invoked). These modifiers are:
      memory_management, **caller_owns_result**, **caller_owns_parameters**,
      object_owns_result, and **object_owns_parameters**.

See the section entitled "Implementation Statements" in Chapter 4, "SOM IDL and the SOM Compiler," for a complete description of these modifiers and their meanings. Note that the memory-management policy for a particular parameter applies to the parameter and all the memory embedded within it (for example, if a struct is owned by the caller, then so are all the struct's members).

When a class contains the **memory_management = corba** SOM IDL modifier, this signifies that all methods introduced by the class follow the CORBA specification for parameter memory management, except where a particular method has an explicit modifier (**object_owns_result** or **object_owns_parameters**) that indicates otherwise. For a description of the CORBA specification, see the earlier subtopic entitled "The CORBA policy for parameter memory management" (under the topic "Memory Management" in Section 6.3 of this chapter).

Building and registering class libraries

The generic server uses SOM's run-time facilities to load class libraries dynamically. Thus, *dynamically linked libraries* (DLLs) should be created for the classes, just as they would be for non-distributed SOM-based applications. For more information, see the topic "Creating a SOM class library" in Chapter 5, "Implementing Classes in SOM."

During the development of the DLL, it is important to remember the following steps:

- Export a routine called **SOMInitModule** in the DLL, which will be called by SOM to initialize the class objects implemented in that library. **SOMInitModule** should contain a *<className>***NewClass** call for each class in the DLL.

- For each class in the DLL, specify the DLL name in the class's IDL file. The DLL name is specified using the **dllname=***<name>* modifier in the *implementation statement* of the interface definition. If not specified, the DLL filename is assumed to be the same as the class name.

- For each class in the DLL, compile the IDL description of the class into the Interface Repository. This is accomplished by invoking the following command syntax:

```
sc –sir –u stack.idl          (On AIX or OS/2)
somc –sir –u stack.idl        (on Windows)
```

    Note: If the classes are not compiled into the Interface Repository, DSOM will generate a run-time error (30056: SOMDERROR_BadDescriptor) when an attempt is made to lookup the signature of a method in the class (for example, on a method call).

- Put the DLL in one of the directories listed in LIBPATH for AIX or OS/2, or listed in PATH for Windows. (This is necessary for OS/2, AIX, and Windows.)

6.6 Configuring DSOM Applications

Preparing the environment

Some environment variables must be defined before running DSOM. Unless noted, these environment variables are required in all operating-system environments.

HOSTNAME=<name> — Each machine that is running DSOM must have its HOST-NAME variable set.

USER=<name> — USER specifies the name of the DSOM user running a client program.

SOMIR=<file(s)> — SOMIR specifies a list of files (separated by a colon on AIX or by a semicolon on OS/2 and Windows) which together make up the Interface Repository. See Chapter 7, "The Interface Repository Framework," for more information on how to set this variable.

Note: For DSOM, it is preferable to use *full* pathnames in the list of IR files, since the IR will be shared by several programs that may not all be started in the same directory.

SOMSOCKETS=<name> — SOMSOCKETS specifies the name of the SOM **Sockets** subclass that implements the sockets services.

Note: For Workstation DSOM, this variable is effectively ignored. (However, it <u>may</u> be used by the Event Management Framework.)

SOMDDIR=<directory> — SOMDDIR specifies the directory where various DSOM files should be located, including the Implementation Repository files. See the later section in this chapter entitled "Registering servers and classes" for more information.

Note: If this value is not set, DSOM will attempt to use a default directory: $SOMBASE/etc/dsom on AIX, and %SOM-BASE%\ETC\DSOM on OS/2 and Windows.

SOMDPORT=<integer> — In DSOM, servers, clients and DSOM daemons communicate with each other using a "sockets" abstraction. In particular, DSOM clients establish connections to DSOM servers by communicating with the DSOM daemon, **somdd**, running on each server machine. The daemon is designed to listen for client requests on a well-known port.

Normally, **somdd** will look in the /etc/services (for AIX) or %ETC%\SERVICES (for OS/2 and Windows) file for its well-known port number. However, if the user has set the SOMD-PORT environment variable, the value of SOMDPORT will be used and the "services" file will not be consulted. The user should pick a 16-bit integer that is not likely to be in use by another application (check the "services" file for ports reserved for use on your machine). Typically, values below 1024 are reserved and should *not* be used.

Note: If there is no "services" file and the SOMDPORT environment variable is not set, DSOM will use a default port number (currently 9393).

SOMDTIMEOUT=<integer> — SOMDTIMEOUT specifies how long a receiver should wait for a message, or how long a sender should wait for an acknowledgement. The value should be expressed in seconds. The default value is 600 seconds (10 minutes).

SOMDDEBUG=<integer> — SOMDDEBUG may optionally be set to enable DSOM run-time error messages. If set to 0, error reporting is disabled. If set to 1, error reporting is enabled. Error reports may be directed to the file named by SOMDMESSAGELOG, if set.

SOMDMESSAGELOG=<file> — SOMDMESSAGELOG may optionally be set to the name of a file where DSOM run-time error messages are recorded. If not set, error messages will be reported on the standard output device.

SOMDNUMTHREADS=<integer>

— SOMDNUMTHREADS may optionally be set to the maximum number of request threads created per server. If SOMDNUMTHREADS is not set, then a separate thread will be created for each request (when using **somdsvr** on OS/2 as a multi-threaded server).

Registering class interfaces

DSOM relies heavily on the Interface Repository for information on method *signatures* (that is, a description of the method's parameters and return value). It is important to compile the IDL for all application classes into the IR before running the application.

For each class in the DLL, compile the IDL description of the class into the Interface Repository. This is accomplished by invoking the following command syntax:

```
sc –sir –u stack.idl          (On AIX or OS/2)
somc –sir –u stack.idl        (On Windows)
```

If the default SOM IR (supplied with the SOMobjects Toolkit and Runtimes) is not used by the application, the user's IR must include the interface definitions for:

- the appropriate **Sockets** class (if the SOMSOCKETS environment variable is set),

- the server class (derived from **SOMDServer**), and

- the definitions of the standard DSOM exceptions (found in file "stexcep.idl") that may be returned by a method call.

Registering servers and classes

Implementation definitions

The Implementation Repository holds **ImplementationDef** objects. The **ImplementationDef** class defines attributes necessary for the **SOMOA** to find and activate the implementation of an object. Details of the **ImplementationDef** object are not currently defined in the CORBA 1.1 specification; the attributes that have been defined are required by DSOM.

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

impl_id (string)

— Contains the DSOM-generated identifier for a server implementation.

impl_alias (string)

— Contains the "alias" (user-friendly name) for a server implementation.

impl_program (string)

— Contains the name of the program or command file which will be executed when a process for this server is started automatically by **somdd**. If the full pathname is not specified, the directories specified in the PATH environment variable will be searched for the named program or command file.

— Optionally, the server program can be run under control of a "shell" or debugger, by specifying the shell or debugger name first, followed by the name of the server program. (A space separates the two program names.) For example,

```
dbx myprogram
```

will start the program "myprogram" under control of "dbx".

— Servers that are started automatically by **somdd** will always be passed their **impl_id** as the first parameter, in order to retrieve their **ImplementationDef** (if desired).

impl_flags (Flags)

— Contains a bit-vector of flags used to identify server options (for example, the IMPLDEF_MULTI_THREAD flag indicates multi-threading).

impl_server_class (string)

— Contains the name of the **SOMDServer** class or subclass created by the server process.

impl_refdata_file (string)

— Contains the full pathname of the file used to store **ReferenceData** for the server.

impl_refdata_bkup (string)

— Contains the full pathname of the backup mirror file used to store **ReferenceData** for the server. This file can be used to restore a copy of the primary file in case it becomes corrupted. (It would be a good idea to keep the primary and backup files in different disk volumes.)

impl_hostname (string)

— Contains the hostname of the machine where the server is located.

The 'regimpl,' 'pregimpl,' and 'wregimpl' registration utilities

Before an implementation (a server program and class libraries) can be used by client applications, it must be registered with DSOM by running the implementation registration utility, **regimpl** (on AIX), **pregimpl** (on OS/2) or **wregimpl** (on Windows). The **regimpl** utility can also be executed from the DOS command line; this facility is available primarily for use in batch files. During execution of **regimpl**, **pregimpl**, or **wregimpl**, DSOM updates its database to include the new server implementation and the associated classes. This enables DSOM to find and, if necessary, to activate the server so that clients can invoke methods on it.

Typically, DSOM users employ the generic SOM-object server program, described below. A discussion on how to write a specific (non-generic) server program is found in the earlier section, "Basic Server Programming."

Registration steps using 'regimpl'

Registering a server implementation and its classes requires the steps described in the following paragraphs.

First, make sure the SOMDDIR environment variable is defined to the name of the Implementation Repository directory, as discussed in the section "Preparing the Environment."

Then, to run the **regimpl** utility, at the system prompt enter:

```
> regimpl
```

This brings up the DSOM Implementation Registration Utility menu, shown below. To begin registering the new implementation, select "1.Add" from the IMPLEMENTATION OPERATIONS section — that is, at the "Enter operation:" prompt, enter "1" (as shown in bold):

```
DSOM IMPLEMENTATION REGISTRATION UTILITY
(C) Copyright IBM Corp. 1992,1993.  All rights reserved.

Implementation data being loaded from: /u/xyz/dsomRepos/

[ IMPLEMENTATION OPERATIONS ]
 1.Add  2.Delete  3.Change
 4.Show one  5.Show all  6.List aliases
[ CLASS OPERATIONS ]
 7.Add  8.Delete  9.Delete from all  10.List classes
[ SAVE & EXIT OPERATIONS ]
 11.Save data  12.Exit
Enter operation: 1
```

The **regimpl** utility then issues several prompts for information about the server implementation (typical responses are shown in bold as an example).

Implementation alias. Enter a "shorthand" name for conveniently referencing the registered server implementation while using **regimpl:**

```
Enter an alias for new implementation:  myServer
```

Program name. Enter the name of the program that will execute as the server. This may be the name of one of the DSOM generic servers (discussed under the later topic "Running DSOM Servers") or a user-defined name for one of these servers. Note: If the program is located in PATH, only the program name needs to be specified. Otherwise, the pathname must be specified.

```
Enter server program name:(default: somdsvr)  <return>
```

Multi-threading. Specify whether or not the server expects the SOM Object Adapter (**SOMOA**) to run each method in a separate thread or not. Notes: This option is only meaningful on OS/2. It is the responsibility of the class implementor to ensure that the methods being executed by the server are "thread safe".

```
Allow multiple threads in the server? [y/n] (default: no) : n
```

Server class. Enter the name of the **SOMDServer** class or subclass that will manage the objects in the server.

```
Enter server class (default: SOMDServer) : <return>
```

Reference data file name. Enter the full pathname of the file used to store ReferenceData associated with object references created by this server. Note: A file name is required *only* if the server is using the **create** method to generate object references.

```
Enter object reference file name (optional) :  <return>
```

Backup reference data file name. Enter the full pathname of the backup file used to mirror the primary ReferenceData file for this server. **Note:** a file name is required *only* if (1) a primary reference data file has been specified, and (2) the application desires an online backup to be maintained. This file can be used to restore a copy of the primary file should it become corrupted.

```
Enter object reference backup file name (optional) : <return>
```

Host machine name. This is the name of the machine on which the server program code is stored. The same name should be indicated in the HOSTNAME environment variable. (If "localhost" is entered, the contents of the HOSTNAME environment variable will be used.

```
Enter host machine name:(default: localhost)  <return>
```

The **regimpl** system next displays a summary of the information defined thus far, and asks for confirmation before adding it. Enter "y" to save the implementation information in the Implementation Repository.

```
================================================================
Implementation id.........: 2befc82b-13a11e00-7f-00-10005ac9272a
Implementation alias......: myServer
Program name..............: somdsvr
Multithreaded.............: No
Server class..............: SOMDServer
Object reference file.....:
Object reference backup...:
Host Name.................: localhost

The above implementation is about to be added. Add? [y/n]  y

Implementation 'myServer' successfully added
```

Add class. Once the server implementation is added, the complete menu reappears. The next series of prompts and entries will identify the classes associated with this server. To begin, from the CLASS OPERATIONS section, select "7.Add":

```
[ IMPLEMENTATION OPERATIONS ]
 1.Add  2.Delete  3.Change
 4.Show one  5.Show all  6.List aliases
[ CLASS OPERATIONS ]
 7.Add  8.Delete  9.Delete from all  10.List classes
[ SAVE & EXIT OPERATIONS ]
 11.Save data  12.Exit

Enter operation: 7
```

Class name. Enter the name of a class associated with the implementation alias.

```
Enter name of class:  class1
```

Implementation alias. Enter the alias for the server that implements the new class (this should be the same alias as given above).

```
Enter alias of implementation that implements class:  myServer

Class 'class1' now associated with implementation 'myServer'
```

The top-level menu will then reappear. Repeat the previous three steps until all classes have been associated with the server.

Then, from the SAVE & EXIT OPERATIONS section, select "11.Save data" to complete the registration. Finally, select "12.Exit" to exit the **regimpl** utility.

```
[ IMPLEMENTATION OPERATIONS ]
 1.Add  2.Delete  3.Change
 4.Show one  5.Show all  6.List aliases
[ CLASS OPERATIONS ]
 7.Add  8.Delete  9.Delete from all  10.List classes
[ SAVE & EXIT OPERATIONS ]
 11.Save data  12.Exit

Enter operation: 11

Enter operation: 12
```

Command line interface to 'regimpl'

The **regimpl** utility also has a command line interface. The command flags correspond to the interactive commands described above. The syntax of the **regimpl** commands follow.

Note: The **regimpl** command and any optional **regimpl** command flags can be entered at a system prompt, and the command will execute as described below. For OS/2 and Windows users, this text-based interface is particularly useful in batch files.

To enter interactive mode:

```
regimpl
```

To add an implementation:

```
regimpl -A -i <str> [-p <str>] [-v <str>] [-f <str>] [-b <str>]
                    [-h <str>] [-m {on|off}] [-z <str>]
```

To update an implementation:

```
regimpl -U -i <str> [-p <str>] [-v <str>] [-f <str>] [-b <str>]
                    [-h <str>] [-m {on|off}]
```

To delete one or more implementations:

```
regimpl -D -i <str> [-i ...]
```

To list all, or selected, implementations:

```
regimpl -L [-i <str> [-i ...]]
```

To list all implementation aliases:

```
regimpl -S
```

To add class associations to one or more implementations:

```
regimpl -a -c <str> [-c ...] -i <str> [-i ...]
```

To delete class associations from all, or selected, implementations:

```
regimpl -d -c <str> [-c ...] [-i <str> [-i ...]]
```

To list classes associated with all, or selected, implementation:

```
regimpl -l [-i <str> [-i ...]]
```

The following parameters are used in the commands described above:

```
-i <str>     = Implementation alias name (maximum of 16 -i names)
-p <str>     = Server program name  (default: somdsvr)
-v <str>     = Server-class name  (default: SOMDServer)
-f <str>     = Reference data file name  (optional)
-b <str>     = Reference data backup file name  (optional)
-h <str>     = Host machine name  (default: localhost)
-m {on|off} = Enable multi-threaded server  (optional)
-z <str>     = Implementation ID
-c <str>     = Class name (maximum of 16 -c names)
```

Registration steps using 'pregimpl' or 'wregimpl'

The **pregimpl** utility is a Presentation Manager version of **regimpl**, the DSOM implementation definition utility. Similarly, the **wregimpl** utility is a Windows version of **regimpl.** The **pregimpl** and **wregimpl** utilities offer all the functionality of **regimpl** except its command-line arguments (described earlier in "Command line interface to 'regimpl'"). In addition, the **pregimpl** and **wregimpl** utilities provide an intuitive GUI interface in place of **regimpl**'s text-based interface. Before proceeding, you should first familiarize yourself with the basic registration process described earlier in "Registration steps using 'regimpl'."

Note: OS/2 or Windows users can execute the text-interface **regimpl** utility by entering **"regimpl"** at a system prompt, as described in the previous topic, "Command line interface to 'regimpl'." This facility is available primarily for use in batch files.

You can start the **pregimpl** or **wregimpl** utility conveniently in either of two ways:

- From the **Register Impls** icon in the SOMobjects icon group, or
- From the Run option of the File menu.

With **pregimpl** or **wregimpl**, you can view, add, change or delete DSOM implementation definitions, as well as view, add or delete implementation class definitions. These basic functions are accessible from the main menu in the initial window that displays when you start **pregimpl** or **wregimpl**. Thus, the main menu offers the choices:

File
Implementations
Classes

To work with an implementation definition, first click *Implementations* on the main menu (or press Alt–I). The pulldown menu that appears shows the options:

> View
> Add
> Change
> Delete

To add an implementation definition, click *Add* on the pulldown menu (or press A). This will bring up the *Add Implementations* dialog box, where you can define or change fields as necessary. The "Alias" field is the only blank field for which a setting is mandatory. For this, you should enter a "shorthand" name for conveniently referencing the registered server implementation while using **pregimpl** or **wregimpl**; for example: `myServer.`

Except for the "Alias" setting, the remainder of the fields may be left either blank or with the default settings that are provided (see "Registration steps using 'regimpl'" for descriptions of the defaults). If you should clear a field that originally contained a default, then when the implementation is applied, **pregimpl** or **wregimpl** will still use the default setting. [Note: The implementation-ID ("Impl ID") field is also displayed. This is not an editable field but is shown for information purposes.]

Once the definition is complete, click *Apply* (or press Alt–A) to add the definition. Click *Discard* (or Alt–D) to discard a new definition. You can select *Exit* (or Alt–X) at any time to exit the dialog box. (Message boxes or confirmation windows typically appear after you make entries.)

To change an implementation definition, click *Change* from the Implementations pulldown menu (or press C). In the *Change Implementations* dialog box that appears, you can select the desired implementation by scrolling though the list box on the left and highlighting your choice. Each time an implementation is highlighted, details of its definition will appear in the edit fields on the right. You can change any of the fields except the implementation-ID. Once the changes are made, click *Apply* (or Alt–A) or *Discard* (or Alt–D) as desired. Click *Exit* (or Alt–X) to exit the dialog box.

The *View* and *Delete* implementations functions work similarly to the Change Implementations function above. That is, you can highlight a selection in the list box on the left, and details of its definition will appear in the edit fields on the right. In the Delete Implementations dialog box, click *Delete Current Implementation* to delete a selected implementation definition. For either function, click *Exit* (or Alt–X) to exit the dialog box.

To work with class definitions, first click *Classes* from the main menu (or press Alt–C). This produces a pulldown menu with the options:

> View
> Add
> Delete

To add classes, select *Add* on the Classes pulldown menu (or press A). From the resulting *Add Classes* dialog box, you can select an implementation by scrolling through the list box on the left and highlighting your choice. When an implementation is selected, the classes associated with it will display in the list box on the right. To add a new class name to the highlighted implementation, enter the name in the edit field at the bottom right, and click *Add Class* (or press Alt–A). The new class name will then appear in the list box on the right.

The current class name will not be cleared from the edit field, so that more implementations may be selected, if appropriate, and the same class can be added to them. Or, you can enter another new class name and add it to the implementation as described above. *Exit* (or Alt–X) may be selected at any time to exit the dialog box.

To view classes, select *View* from the Classes pulldown menu (or press V). When you highlight an implementation in the list box on the left, the classes associated with the implementation will be shown in the list box on the right.

To switch key order, select the "Classes" radio button under "Display keyed by". This causes all of the defined classes to appear in the list box on the left. When you highlight a class name, all of the implementations associated with that class will appear in the list box on the right. *Exit* (or Alt–X) may be selected at any time to exit the dialog box.

To delete classes, select *Delete* from the Classes pulldown menu (or press D). When you highlight an implementation in the list box on the left, the classes associated with the implementation will be shown in the list box on the right. You can then highlight a class and click *Delete Class* (or Alt–D) to delete the class from the highlighted implementation. A confirmation window will appear next, which will also give you the option of deleting the class from *all* implementations. *Exit* (or Alt–X) may be selected at any time to exit the dialog box.

Important: Please note that any changes are saved internally in memory, but are *not* written to the database until you save the changes by clicking *File* (or Alt–F) from the main menu and then *Save* (or S) from the resulting pulldown menu. Conversely, you can abort all changes and reload the original database by clicking *File* (Alt–F) and then *Abort+Reload* (A). To exit the **pregimpl** or **wregimpl** program, click *File* (Alt–F) and then *Exit* (X).

Programmatic interface to the Implementation Repository

The Implementation Repository can be accessed and updated dynamically using the programmatic interface provided by the **ImplRepository** class (defined in "implrep.idl"). The global variable **SOMD_ImplRepObject** is initialized by **SOMD_Init** to point to the **ImplRepository** object. The following methods are defined on it:

```
void   add_impldef (in ImplementationDef impldef);
```

> — Adds an implementation definition to the Implementation Repository. (Note: The value of the "impl_id" attribute is ignored. A unique **ImplId** will be generated for the newly added **ImplementationDef**.)

```
void   delete_impldef (in ImplId implid);
```

> — Deletes an implementation definition from the Implementation Repository, given the ID of the implementation definition.

```
void   update_impldef (in ImplementationDef impldef);
```

> — Updates the implementation definition (defined by the "impl_id" of the supplied **ImplementationDef**) in the Implementation Repository.

```
ImplementationDef   find_impldef (in ImplId implid);
```

> — Returns a server implementation definition given its ID.

```
ImplementationDef   find_impldef_by_alias (in string alias_name);
```

> — Returns a server implementation definition, given its user-friendly alias.

```
sequence<ImplementationDef>   find_impldef_by_class (
                                         in string classname);
```

> — Returns a sequence of **ImplementationDef**s for those servers that have an association with the specified class. Typically, a server is associated with the classes it knows how to implement, by registering its known classes via the **add_class_to_impldef** method.

```
ORBStatus   find_all_impldefs (
                   out sequence<ImplementationDef> outimpldefs);
```

> — Retrieves all **ImplementationDef** objects in the Implementation Repository.

The following methods maintain an association between server implementations and the names of the classes they implement. These methods effectively maintain a mapping of <className, Implid>.

```
void  add_class_to_impldef (in ImplId implid,
                            in string classname);
```

— Associates a class, identified by name, with a server, iden-
tified by its **ImplId**. This type of association is used to
lookup server implementations via the **find_impl-
def_by_class** method.

```
void  remove_class_from_impldef (
                            in ImplId implid,
                            in string classname);
```

— Removes the association of a particular class with a server.

```
void  remove_class_from_all (
                            in string classname);
```

— Removes the association of a particular class from all serv-
er implementations in the Implementation Repository.

```
sequence<string>  find_classes_by_impldef (in ImplId implid);
```

— Returns a sequence of class names associated with a
server.

With the **ImplRepository** programmatic interface, it is possible for an application to define
additional server implementations at run time.

The 'dsom' server manager utility

The **dsom** utility is a command-line utility program used to manage server processes. At
present, server processes that can be managed are limited to those present in the Implementa-
tion Repository. The choice of Implementation Repository is determined by the environment
variable SOMDDIR. The **dsom** utility can be used to start, restart, stop, list, enable, or disable
server processes. The syntax of the **dsom** command is as follows:

> **dsom** <*cmd*> **{** *impl_alias1* **[** *impl_alias2* **...]** **|** *** }**

where <*cmd*> can be any one of the terms: **start**, **restart**, **stop**, **list**, **disable**, or **enable**. Each
impl_alias is the server-alias name for a server process. All forms of the command take one or
more server-alias names, or a wild card character "*****". The "*****" will be replaced with all of the
server-alias names present in the Implementation Repository.

For example, to start one or more server processes, the command takes the form:

```
dsom start { impl_alias1 [impl_alias2 ...] | * }
```

To restart one or more server processes:

```
dsom restart { impl_alias1 [impl_alias2 ...] | * }
```

To stop one or more server processes:

```
dsom stop { impl_alias1 [impl_alias2 ...] | * }
```

To list the status of one or more server processes:

```
dsom list { impl_alias1 [impl_alias2 ...] | * }
```

To prevent the server processes from starting, use the disable command. To disable one or
more server processes:

```
dsom disable { impl_alias1 [impl_alias2 ...] | * }
```

A previously disabled server process can be enabled by the enable command. To enable one or
more server processes:

```
dsom enable { impl_alias1 [impl_alias2 ...] | * }
```

Interpretation of 'dsom' messages

The messages generated by the **dsom** utility have a one-to-one mapping on the DSOM error codes. Knowing this mapping will aid in a better understanding of the **dsom** return messages. The following six messages are mapped onto the DSOM error code of 0 (success):

'dsom' cmd	Message
start	**Started server process**
restart	**Restarted server process**
list	**Server process currently running**
stop	**Stopped server process**
disable	**Successfully disabled server**
enable	**Successfully enabled server**

The messages generated by the **dsom** utility commands are mapped onto DSOM error codes, as follows:

DSOM error code	Message
SOMDERROR_ServerNotFound	**Server process not running**
SOMDERROR_NotProcessOwner	**Cannot stop server; Not process owner**
SOMDERROR_NoSocketsClass	**Cannot find Sockets class**
SOMDERROR_NoRemoteComm	**Not enabled for Workgroup**
SOMDERROR_CommTimeOut	**Client timed out**
SOMDERROR_SendError	**Send Error**
SOMDERROR_SocketSend	**Send Error**
SOMDERROR_ServerInactive	**Server activation pending**
SOMDERROR_NoSOMDInit	**Unable to create global LocServ Object**
SOMDERROR_UnknownError	**Command not supported by daemon**
SOMDERROR_ServerDisabled	**Server process is currently disabled**
SOMDERROR_CouldNotStartProcess	**Server process cannot be started**
SOMDERROR_ServerToBeDeleted	**Current server process marked for deletion; try again**
Any other error	**Request unsuccessful; Unknown error**

In addition, if the *impl_alias* specified with any **dsom** utility command is not present in the Implementation Repository, DSOM will generate the message: **Server alias not found in Implementation Repository**.

Programmatic interface to manage server processes

Server processes can also be managed by using the programmatic interface provided by the **SOMDServerMgr** class. For descriptions of the **SOMDServerMgr** class and its corresponding methods, see the DSOM section of the *SOMobjects Developer Toolkit Programmers Reference Manual*.

Verifying the DSOM environment with 'somdchk'

The **somdchk** program evaluates the environment to verify whether DSOM can operate correctly. As described in the preceding topics of Sections 6.5 "Implementing Classes" and 6.6 "Configuring DSOM Applications," to operate correctly DSOM must be able to find the appropriate libraries (DLLs), the Interface Repository, and the Implementation Repository. The settings of various environment variables help DSOM find the path to the libraries and repositories.

The **somdchk** program generates messages that evaluate the DSOM environment. It determines whether the necessary SOM DLLs can be located, whether DSOM is enabled for workgroup (cross-machine) communication, whether Interface and Implementation Repositories can be located, and it displays the settings of important environment variables. In its "verbose" mode, **somdchk** gives the default settings for DSOM environment variables and explains how DSOM uses them.

On AIX or OS/2, the program is invoked from the command line using the syntax given below. The optional verbose setting can be turned on by including the **–v** option with the command:

somdchk [–v]

On Windows, the **somdchk** program can be invoked by double clicking on the SOMDCHK icon. The resulting messages will appear in the message window.

The following example shows sample output from the **somdchk –v** command on AIX. Output on other platforms would look similar.

```
      D S O M     E N V I R O N M E N T     E V A L U A T I O N

SOMBASE = /usr/lpp/som
SOMBASE should not be set to the base directory of the SOMObjects
Toolkit Enablers.

Searching for important DLLs.....
/usr/lpp/som/lib/som.dll found.
/usr/lpp/som/lib/somd.dll found.
/usr/lpp/som/lib/soms.dll found.
/usr/lpp/som/lib/somst.dll found.

/usr/lpp/som/lib/somd.dll IS Workgroup Enabled.
Workgroup Enabled DLL permits inter-machine communication.

SOMSOCKETS  = TCPIPSockets
SOMSOCKETS must be set. Valid settings are:
   TCPIPSockets for TCPIP.
   IPXSockets for NetWare.

SOMDDIR    = /u/raviv/impl_rep/
Valid Implementation Repository found in /u/raviv/impl_rep/
SOMDDIR may be set to a valid directory in which the Implementation
Repository resides.
   Default is /usr/lpp/som/etc/dsom


SOMIR    = /u/raviv/raviv.ir
SOMIR may be set to a list of file names which together form the
Interface Repository.
   Default is ./som.ir
/u/raviv/raviv.ir found.

SOMDPORT   = 3001
SOMDPORT may be set to a 'well-known port'. Default value is 9393.

SOMDTIMEOUT  = (null).
SOMDTIMEOUT may be set to the number of seconds to timeout.
   Default value is 600.

SOMDDEBUG    = 2
SOMDDEBUG may be set to 1 to enable runtime error messages.
   Default value is 0.

SOMDMESSAGELOG    = (null).
SOMDMESSAGELOG may be set to the name of a file where messages may
be logged.
   Default is stdout.
```

6.7 Running DSOM Applications

Prior to starting the DSOM processes, the DSOM executables should be installed and the DSOM environment variables should be set appropriately, as discussed in the earlier section, "Configuring DSOM Applications."

Running the DSOM daemon (somdd)

To run a DSOM application, the DSOM daemon, **somdd,** must be started

- On AIX or OS/2, the daemon can be started manually from the command line, or could be started automatically from a start-up script run at boot time. It may be run in the background with the commands **somdd&** on AIX, and **start somdd** on OS/2. The **somdd** command has the following syntax:

    somdd [–q]

    where the optional **–q** flag signifies "quiet" mode. By default, **somdd** will produce a "ready" message when the DSOM daemon is ready to process requests, and it will produce diagnostic messages as errors are encountered if the SOMDDEBUG environment variable is set to 1. In quiet mode, however, the "ready" message will not appear, and diagnostic messages will not appear even if SOMDDEBUG is set. Alternatively, if the SOMDMESSAGELOG environment variable is set, diagnostic error messages will be sent directly to the specified message log file, regardless of whether the **–q** flag is specified.

- On Windows, the daemon can be started with the **DSOM Daemon** icon in the SOM icon group or started in Windows from the Run option of the File menu. The **DSOM Daemon** icon will change colors to indicate that the daemon is ready, after which client and server programs can be started.

The **somdd** daemon is responsible for "binding" a client process to a server process and will activate the desired server if necessary. The binding procedure is such that the client will consult the Implementation Repository to find out which machine contains a desired server, and will then contact the DSOM daemon on the server's machine to retrieve the server's communications address (a port). Servers are activated dynamically as separate processes.

Running DSOM servers

Once the **somdd** daemon is running, application programs can be started. If the application uses the generic SOM server, **somdsvr**, it can be started either from the command line or automatically upon demand. When starting **somdsvr** from the command line, the server's implementation ID or alias must be supplied as an argument. The command syntax for starting a generic SOM server is:

    somdsvr [*impl_id* | **–a** *alias* **]**

For example, the command

```
$   somdsvr  2ad2688fb-00389c00-7f-00-10005ac900d8
```

would start a **somdsvr** for an implementation with the specified ID. Likewise, the command

```
$   somdsvr  -a  myServer
```

would start a **somdsvr** that represents an implementation of "myServer".

6.8 DSOM as a CORBA-compliant Object Request Broker

The Object Management Group (OMG) consortium defines the notion of an *Object Request Broker (ORB)* that supports access to remote objects in a distributed environment. Thus, Distributed SOM is an ORB. SOM and DSOM together comply with the OMG's specification of the Common Object Request Broker Architecture (CORBA).

Since the interfaces of SOM and DSOM are largely determined by the CORBA specification, the CORBA components and interfaces are highlighted in this section.

The CORBA specification defines the components and interfaces that must be present in an ORB, including the:

- Interface Definition Language (IDL) for defining classes (discussed in Chapter 4, "SOM IDL and the SOM Compiler"),

- C usage bindings (procedure-call formats) for invoking methods on remote objects,

- Dynamic Invocation Interface and an Interface Repository, which support the construction of requests (method calls) at run time (for example, for interactive desktop applications), and

- Object Request Broker run-time programming interfaces.

SOM and DSOM were developed to comply with these specifications (with only minor extensions to take advantage of SOM services). Although the capabilities of SOM are integral to the implementation of DSOM, the application programmer need not be aware of SOM as the implementation technology for the ORB.

This section assumes some familiarity with *The Common Object Request Broker: Architecture and Specification, Revision 1.1* (also referred to as "CORBA 1.1"). The specification is published jointly by the Object Management Group and x/Open™. The mapping of some CORBA 1.1 terms and concepts to DSOM terms and concepts is described in the remainder of this section.

Mapping OMG CORBA terminology onto DSOM

This section discusses how various CORBA concepts and terms are defined in terms of DSOM's implementation of the CORBA 1.1 standard.

Object Request Broker run-time interfaces

In the previous sections, the **SOMDObjectMgr** and **SOMDServer** classes were introduced. These are classes defined by DSOM to provide basic support in managing objects in a distributed application. These classes are built upon Object Request Broker interfaces defined by CORBA for building and dispatching requests on objects. The ORB interfaces, **SOMDObjectMgr** and **SOMDServer,** together provide the support for implementing distributed applications in DSOM.

CORBA 1.1 defines the interfaces to the ORB components in IDL. In DSOM, the ORB components are implemented as SOM classes whose interfaces are expressed using the same CORBA 1.1 IDL. Thus, an application can make calls to the DSOM run time using the SOM language bindings of its choice.

Interfaces for the following ORB run-time components are defined in CORBA 1.1, and are implemented in DSOM. They are introduced briefly here, and discussed in more detail throughout this chapter. (See the *SOMobjects Developer Toolkit Programmers Reference Manual* for the complete interface definitions.)

Object — The **Object** interface defines operations on an "object reference", which is the information needed to specify an object within the ORB.

In DSOM, the class **SOMDObject** implements the CORBA 1.1 **Object** interface. (The **"SOMD"** prefix was added to distinguish this class from **SOMObject**.) The subclass **SOMDClientProxy** extends **SOMDObject** with support for proxy objects.

ORB	— (Object Request Broker) The **ORB** interface defines utility routines for building requests and saving references to distributed objects. The global variable **SOMD_ORBObject** is initialized by **SOMD_Init** and provides the reference to the ORB object.
ImplementationDef	— An **ImplementationDef** object is used to describe an object's implementation. Typically, the **ImplementationDef** describes the program that implements an object's server, how the program is activated, and so on.
	(CORBA 1.1 introduces **ImplementationDef** as the name of the interface, but leaves the remainder of the IDL specification to the particular ORB. DSOM defines an interface for **ImplementationDef**.)
	ImplementationDef objects are stored in the *Implementation Repository* (defined in DSOM by the **ImplRepository** class).
InterfaceDef	— An **InterfaceDef** object is used to describe an IDL interface in a manner that can be queried and manipulated at run time when building requests dynamically, for example.
	InterfaceDef objects are stored in the *Interface Repository* (described fully in Chapter 7, "The Interface Repository Framework").
Request	— A **Request** object represents a specific request on an object, constructed at run time. The **Request** object contains the target object reference, operation (method) name, a list of input and output arguments. A **Request** can be invoked synchronously (wait for the response), asynchronously (initiate the call, and later, get the response), or as a "oneway" call (no response expected).
NVList	— An **NVList** is a list of **NamedValue** structures, used primarily in building **Request** objects. A **NamedValue** structure consists of a name, typed value, and some flags indicating how to interpret the value, how to allocate/free the value's memory, and so on.
Context	— A **Context** object contains a list of "properties" that represent information about an application process's environment. Each **Context** property consists of a <name,string_value> pair, and is used by application programs or methods much like the "environment variables" commonly found in operating systems like AIX, OS/2, and Windows. IDL method interfaces can explicitly list which properties are queried by a method, and the ORB will pass those property values to a remote target object when making a request.
Principal	— A **Principal** object identifies the principal ("user") on whose behalf a request is being performed.
	(CORBA 1.1 introduces the name of the interface, **Principal**, but leaves the remainder of the IDL specification to the particular ORB. DSOM defines an interface for **Principal**.)
BOA	— (Basic Object Adapter) An Object Adapter provides the primary interface between an implementation and the ORB "core". An ORB may have a number of Object Adapters, with interfaces that are appropriate for specific kinds of objects.

The **Basic Object Adapter** is intended to be a general-purpose Object Adapter available on all CORBA-compliant Object Request Brokers. The **BOA** interface provides support for generation of object references, identification of the principal making a call, activation and deactivation of objects and implementations, and method invocation on objects.

In DSOM, **BOA** is defined as an abstract class. The **SOMOA** (SOM Object Adapter) class, derived from **BOA**, is DSOM's primary Object Adapter implementation. The **SOMOA** interface extends the **BOA** interface with several of its own methods that are *not* defined by CORBA 1.1.

Object references and proxy objects

CORBA 1.1 defines the notion of an *object reference*, which is the information needed to specify an object in the ORB. An object is defined by its **ImplementationDef**, **InterfaceDef**, and application-specific "reference data" used to identify or describe the object. An object reference is used as a handle to a remote object in method calls. When a server wants to export a reference to an object it implements, it supplies the object's **ImplementationDef**, **InterfaceDef**, and reference data to the Object Adapter, which returns the reference.

The structure of an object reference is opaque to the application, leaving its representation up to the ORB.

In DSOM, an object reference is represented as an object that can simply be used to identify the object on that server. The DSOM class that implements simple object references is called **SOMDObject** (corresponding to **Object** in CORBA 1.1.) However, in a client's address space, DSOM represents the remote object with a *proxy object* in order to allow the client to invoke methods on the target object as if it were local. When an object reference is passed from a server to a client, DSOM *dynamically* and *automatically* creates a proxy in the client for the remote object. Proxies are specialized forms of **SOMDObject**; accordingly, the base proxy class in DSOM, **SOMDClientProxy**, is derived from **SOMDObject**.

In order to create a proxy object, DSOM must first build a proxy class. It does so automatically using SOM facilities for building classes at run time. The proxy class is constructed using multiple inheritance: the proxy object functionality is inherited from **SOMDClientProxy**, while just the *interface* of the target class is inherited. (See the illustration below.)

Construction of a proxy class in DSOM

SOMDObject

SOMDClientProxy

"Stack"

"Stack__Proxy"

abstract inheritance
(interface only)
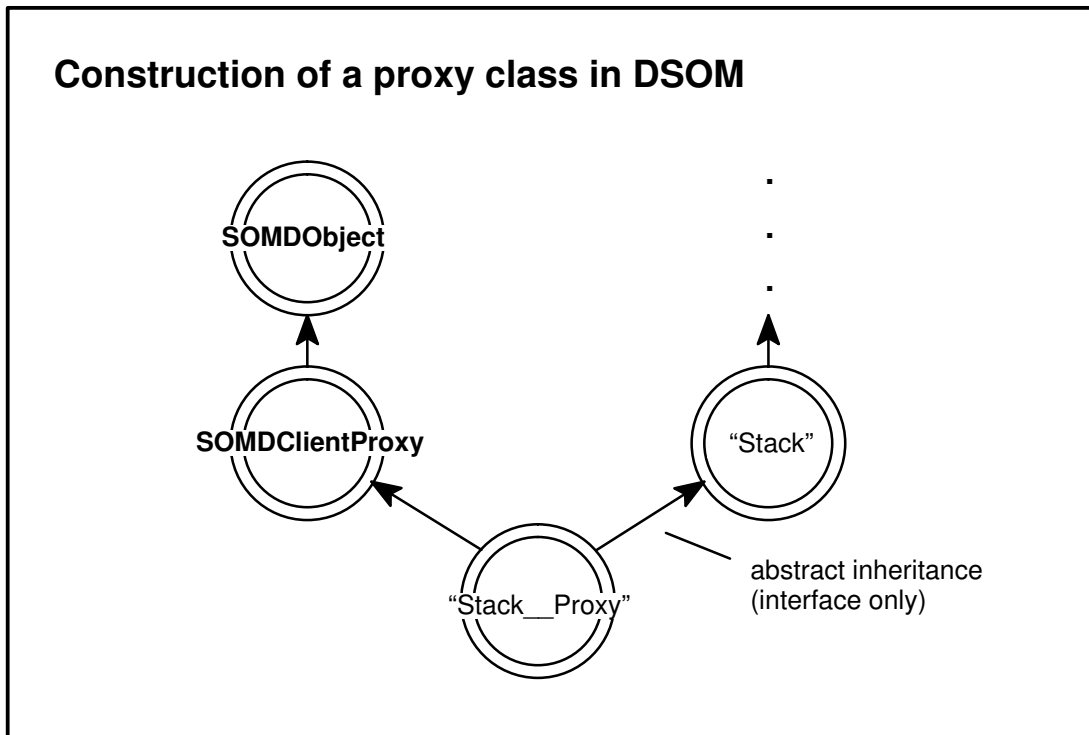
In the newly derived proxy class, DSOM overrides each method inherited from the target class with a "remote dispatch" method that forwards an invocation request to the remote object. Consequently, the proxy object provides location transparency, and the client code invokes operations (methods) on the remote object using the same language bindings as if it were a local target object.

For example, recall the "Stack" class used in the tutorial example given earlier. When a server returns a reference to a remote "Stack" object to the client, DSOM builds a "Stack_ _Proxy" class (note two underscores in the name), derived from **SOMDClientProxy** and "Stack", and creates a proxy object from that class. When the client invokes the "push" method on the proxy,

```
_push(stk, &ev, 100);
```

the method is redispatched using the remote-dispatch method of the **SOMDClientProxy** class, and the method is forwarded to the target object.

CORBA defines several special operations on object references that operate on the local references (proxies) themselves, rather than on the remote objects. These operations are defined by the classes **SOMOA** (SOM Object Adapter), **SOMDObject** (which is DSOM's implementation of CORBA's **Object** "pseudo-class") and **ORB** (Object Request Broker class). Some of these operations are listed below, expressed in terms of their IDL definitions.

SOMOA methods (inherited from **BOA**):

```
sequence <octet,1024> ReferenceData;
SOMDObject  create (in ReferenceData id, in InterfaceDef intf,
                    in ImplementationDef impl);
```

— Creates and returns an object reference.

SOMDObject methods:

```
SOMDObject  duplicate ( );
```

— Creates and returns a duplicate object reference.

```
void  release ( );
```

— Destroys an object reference.

```
boolean  is_nil ( );
```

— Tests to see if the object reference is NULL.

ORB methods:

```
string  object_to_string ( SOMDObject obj );
```

— Converts an object reference to a (storable) string form.

```
SOMDObject  string_to_object ( string str );
```

— Converts a string form back to the original object reference.

Creation of remote objects

The OMG has standardized an "object lifecycle service," built on top of the ORB, for creating and destroying remote objects. Currently, DSOM provides its own interfaces for creating and destroying objects (see "Basic Client Programming"), but a future release may provide an OMG-compliant lifecycle service as well.

Interface definition language

The CORBA specification defines an Interface Definition Language, IDL, for defining object interfaces. The SOM Compiler compiles standard IDL interface specifications, but it also allows the class implementor to include implementation information that will be used in the implementation bindings for a particular language.

Note: Before IDL, SOM (version 1.0) had its own Object Interface Definition Language (OIDL). SOM classes specified using OIDL must be converted to IDL before they can be used with DSOM. The SOMobjects Developer Toolkit provides a migration tool for this purpose. (See Appendix B of the *SOMobjects Developer Toolkit User's Guide*.)

C language mapping

The CORBA specification defines the mapping of method interface definitions to C language procedure prototypes, hence SOM defines the same mapping. This mapping requires passing a reference to the target object and a reference to an implementation-specific **Environment** structure as the first and second parameters, respectively, in any method call.

The **Environment** structure is primarily used for passing error information from a method back to its caller. See also the topic "Exceptions and Error Handling" in Chapter 3, "Using SOM Classes in Client Programs," for a description of how to "get" and "set" error information in the **Environment** structure.

Dynamic Invocation Interface (DII)

The CORBA specification defines a Dynamic Invocation Interface (DII) that can be used to dynamically build requests on remote objects. This interface is described in section 6 (page 105) of the CORBA 1.1 document, and is implemented in DSOM. The DSOM implementation of the DII is described later in this chapter, in the topic entitled "Dynamic Invocation Interface" under Section 6.9, "Advanced Topics." Note that, in DSOM, **somDispatch** is overridden so that method invocations on proxy objects are forwarded to the remote target object. SOM applications can use the SOM **somDispatch** method for dynamic method calls whether the object is local or remote.

Implementations and servers

The CORBA specification defines the term *implementation* as the *code that implements an object*. The implementation usually consists of a program and class libraries.

Servers are processes that *execute* object implementations. CORBA 1.1 defines four activation policies for server implementations: shared, unshared, server-per-method, and persistent, as follows.

- A *shared* server implements multiple objects (of arbitrary classes) at the same time, and allows multiple methods to be invoked at the same time.

- An *unshared* server, conversely, implements only a single object, and handles one request at a time.

- The *server-per-method* policy requires a separate process to be created for each request on an object and, usually, a separate program implements each method.

   Under the shared, unshared, and server-per-method activation policies, servers are activated automatically (on demand).

- A *persistent* server, by contrast, is a shared server that is activated "by hand" (for example, from the command shell or from a startup script), vs. being activated automatically when the first method is dispatched to it.

The term "persistent server" refers to the relative lifetime of the server: it is "always running" when DSOM is running. (CORBA implies that persistent servers are usually started at ORB boot time.) It should not be assumed, however, that a "persistent" server necessarily implements persistent objects (that persist between ORB reboots).

In DSOM, specific process models are implemented by the server program. That is, DSOM simply starts a specified program when a client attempts to connect to a server. The four CORBA activation policies, or any other policies, can be implemented by the application as necessary. For example,

- an object that requires a server-per-method implementation could itself spawn a process at the beginning of each method execution. Alternatively, the server object in the "main" server can spawn a process before each method dispatch.

- a dedicated server could be registered for each object that requires an unshared server implementation (separate process). This may be done dynamically (see the topic "Programmatic interface to the Implementation Repository" earlier in this chapter).

An **ImplementationDef** object, as defined by the CORBA specification, describes the characteristics of a particular implementation. In DSOM, an **ImplementationDef** identifies an implementation's unique ID, the program name, its location, and so forth. The **ImplementationDef** objects are stored in an *Implementation Repository*, which is represented in DSOM by an **ImplRepository** object.

A CORBA-compliant ORB must provide the mechanisms for a server program to register itself with the ORB. To "register itself with the ORB" simply means to tell the ORB enough information about the server process so that the ORB will be able to locate, activate, deactivate, and dispatch methods to the server process. DSOM supports these mechanisms, so that server programs written in arbitrary languages can be used with DSOM. (See also the next topic, "Object Adapters.")

In addition to the generic registration mechanisms provided by all CORBA-compliant ORBs, DSOM provides extra support for using SOM-class DLLs. DSOM provides a *generic server* program that automatically registers itself with DSOM, loads SOM-class DLLs on demand, and dispatches incoming requests on SOM objects. Thus, by using the generic server program (when appropriate), a user may be able to avoid writing any server program code.

Object Adapters

An *Object Adapter* (OA) provides the mechanisms that a server process uses to interact with DSOM, and vice versa. That is, an Object Adapter is responsible for server activation and deactivation, dispatching methods, activation and deactivation of individual objects, and providing the interface for authentication of the principal making a call.

DSOM defines a *Basic Object Adapter* (**BOA**) interface, described in the CORBA specification, as an abstract class (a class having no implementation, only an interface specification). The **BOA** interface represents generic Object Adapter methods that a server written in an arbitrary language can use to register itself and its objects with the ORB. Because it is an abstract class having no implementation, however, the **BOA** class should not be directly instantiated.

DSOM provides a SOM Object Adapter, **SOMOA**, derived from the **BOA** interface, that uses SOM Compiler and run-time support to accomplish dispatching of methods (that is, accepting messages, turning them into method invocations, and routing the invocations to the target object in the server process). **SOMOA** can be used to dispatch methods on either SOM or non-SOM object implementations, as described in the sections "Implementing Classes" and "Basic Server Programming." It is possible to use non-SOM based implementations with **SOMOA**, and often there is no additional programming required to use implementations (class libraries) already developed using SOM.

The **SOMOA** works in conjunction with the application-defined *server object* to map between objects and object references, and to dispatch methods on objects. By partitioning out these mapping and dispatching functions into the server object, the application can more easily customize them, without having to build object adapter subclasses.

SOMOA introduces two methods that handle execution of requests received by the server:

> **execute_request_loop**
> **execute_next_request**

Typically, **execute_request_loop** is used to receive and execute requests, continuously, in the server's main thread. The **execute_next_request** method allows a single request to be executed. Both methods have a non-blocking option: when there are no messages pending, the method call will return instead of wait.

On OS/2, if the server implementation has been registered as "multi-threaded" (via an IMPL-DEF_MULTI_THREAD flag in the **ImplementationDef**), **SOMOA** will *automatically* run each request in a separate thread. If the "multi-thread" flag is not set, the server implementation can still choose to manage its own threads.

The generic server program provided by DSOM (described in the preceding topic) uses **execute_request_loop** to receive and execute requests on SOM objects.

Extensions and limitations

The DSOM implementation has the following extensions and limitations in its implementation of the CORBA specification:

• As just described, the current release of DSOM supports a simple server activation policy, which is equivalent to the "shared" and "persistent" policies defined by CORBA. DSOM does not explicitly support the "unshared" or "server-per-method" server activation policies. Policies other than the basic activation scheme must be implemented by the application.

• DSOM provides null implementations for the **obj_is_ready** or **deactivate_obj** methods, defined by the **BOA** interface for the unshared server activation policy.

• DSOM does not support the **change_implementation** method, defined by the **BOA** interface to allow an application to change the implementation definition associated with an object. In DSOM, the **ImplementationDef** identifies the server which implements an object. In these terms, changing an object's **ImplementationDef** would result in a change in the object's server ID. Any existing object references that have the old server ID would be rendered invalid.

It *is* possible, however, to change the program which implements an object's server, or change the class library which implements an object's class. To modify the program associated with an **ImplementationDef**, use the **update_impldef** method defined on **Impl-Repository.** To change the implementation of an object's class, replace the corresponding class library with a new (upward-compatible) one.

• The OUT_LIST_MEMORY, IN_COPY_VALUE, and DEPENDENT_LIST flags, used with the Dynamic Invocation Interface, are not yet supported.

• The SOM Object Adapter (**SOMOA**) provides a method (**change_id**) to update the **ReferenceData** associated with an object reference created by the **create** call. This is useful if the information which describes the object must be changed without invalidating copies of the existing object reference. CORBA defines no such method; **change_id** is an extension to the standard **BOA** methods.

- The **SOMOA** provides some specialized object reference types which, in certain situations, are more efficient or easier to use than standard object references.

- DSOM supports the SOM extension to IDL that allows method parameters that are pointers. Structure, sequence, and array parameters may only contain pointers to *objects* (not arbitrary types).

6.9 Advanced Topics

Peer vs. client-server processes

The client-server model of distributed computing is appropriate when it is convenient (or necessary) to centralize the implementation and management of a set of shared objects in one or more servers. However, some applications require more flexibility in the distribution of objects among processes. Specifically, it is often useful to allow processes to manage and export some of their objects, as well as access remote objects owned by other processes. In these cases, the application processes do not adhere to a strict client-server relationship — instead, they cooperate as "peers", behaving both as clients and as servers.

Peer applications must be written to respond to incoming asynchronous requests, in addition to performing their normal processing. In a multi-threaded system (like OS/2), this is best accomplished by dedicating a separate process thread that handles DSOM communications and dispatching. In systems that do not currently support multi-threading (like AIX and Windows), peer applications must be structured as event-driven programs.

Multi-threaded DSOM programs

In a system that supports multi-threading, like OS/2, the easiest way to write a peer DSOM program is to dedicate a separate thread to perform the usual "server" processing. This body of this thread would contain the same code as the simple servers described in section 6.4, "Basic Server Programming."

```
DSOM_thread(void *params)
{
    Environment ev;
    SOM_InitEnvironment(&ev);

/* Initialize the DSOM run-time environment */
    SOMD_Init(&ev);

/* Retrieve its ImplementationDef from the Implementation
   Repository by passing its implementation ID as a key */
    SOMD_ImplDefObject =
        _find_impldef(SOMD_ImplRepObject, &ev, *(ImplId *)params);

/* Create SOM Object Adapter and begin executing requests */
    SOMD_SOMOAObject = SOMOANew();
    _impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
    _execute_request_loop(SOMD_SOMOAObject, &ev, SOMD_WAIT);
}
```

Note: The DSOM run time is "thread safe"; that is, DSOM protects its *own* data structures and objects from race conditions and update conflicts. However, it is the application's responsibility to implement its own concurrency control for concurrent thread access to local shared application objects.

Event-driven DSOM programs using EMan

EMan (see Chapter 9 on "The Event Management Framework") is not a replacement for threads, but it supports processing of asynchronous requests. EMan allows a program to handle events from multiple input sources — but the handlers run on a single thread, under control of EMan's main loop.

DSOM provides a runtime function, **SOMD_RegisterCallback**, which is used by DSOM to associate user-supplied event handlers with DSOM's communications sockets and message queues with EMan. Example code is shown below.

DSOM server programs which use EMan must be very careful not to get into deadlock situations. This is quite easy to do with DSOM, since method calls are synchronous. If two cooperating processes simultaneously make calls on each other, a deadlock could result. Likewise, if a method call on remote object B from A requires a method call back to A, a deadlock cycle will exist. (Of course, the number of processes and objects which create the cyclic dependency could be greater than two.) To illustrate:

Potential deadlocks exist using EMan and DSOM

register(B, A)

(then wait
for response)

A

get_data(A)

B

(then wait
for response)

DEADLOCK!
Synchronous
method calls
block, awaiting
responses ...
for requests
that will never
be serviced.

notify(B)

A

notify(A)

B

OK!
Note: oneway messages may be used to avoid deadlock.
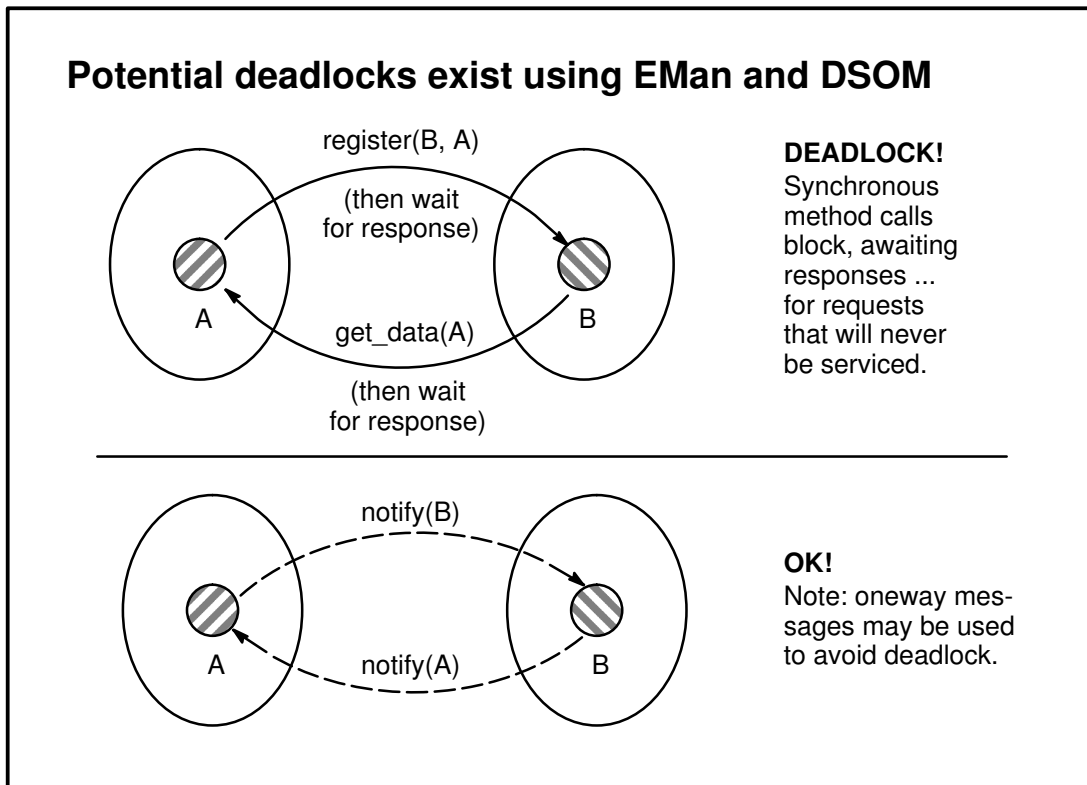
The application developer must be careful to avoid situations where cooperating processes are likely to make calls upon each other, creating a cyclic dependency. Some applications may find it appropriate to use *oneway messages* to avoid deadlock cycles, since oneway messages do not cause a process to block. It may also be possible for an application to *defer* the actual processing of a method that may "call back" an originating process, by scheduling work using EMan client events.

Sample server using EMan

The following server code has been distilled from one of the DSOM sample applications provided with SOMobjects Developer Toolkit. It is an example of a server which has an interval timer that signals another server (via DSOM) whenever its timer "pops". Thus, it is both a client (of the server it signals) and a server (because it can receive timer notifications from other servers).

The IDL for the server object class to be used by this server program is as follows. Note that the "noteTimeout" method is *oneway*, in order to avoid deadlock.

```
interface PeerServer : SOMDServer
{ oneway void noteTimeout(in string serverName);
  // Notification that a timer event occurred in server serverName
};
```

The example server program is outlined as follows. It is assumed that "eman.h" has been included by the program.

• Perform DSOM initialization up to, but not including, asking **SOMOA** to start handling requests.

```
MyEMan = SOMEEManNew();
SOM_InitEnvironment(&ev);
SOM_InitEnvironment(&peerEv);
SOMD_Init(&ev);

somPrintf("What is the alias for this server? ");
gets(thisServer);

SOMD_ImplDefObject = _find_impldef_by_alias(SOMD_ImplRepObject,
                                            &ev, thisServer);
SOMD_SOMOAObject = SOMOANew();
_impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
```

- Register a "DSOM event" with EMan, having EMan callback to a procedure that asks the **SOMOA** to process any pending DSOM requests.

```
void SOMD_RegisterCallback(SOMEEman emanObj, EMRegProc *func);

void DSOMEventCallBack (SOMEEvent event, void *eventData)
{    Environment ev;
     SOM_InitEnvironment(&ev);
     _execute_request_loop(SOMD_SOMOAObject, &ev, SOMD_NO_WAIT);
}

SOMD_RegisterCallback (MyEMan, DSOMEventCallBack);
```

- Ask user to provide "target server's alias", where the target server is that this server will signal when its timer "pops". Then get a proxy for that server.

```
somPrintf("What is the alias for the target server? ");
gets(inbuf);
RemotePeer = _somdFindServerByName(SOMD_ObjectMgr, &ev, inbuf);
```

- Ask user to provide the timer's interval (in milliseconds)

```
somPrintf("What is the timer interval, in millseconds? ");
gets(inbuf);
Interval = atoi(inbuf);
```

- Register a timer event with EMan, having EMan call back a procedure that will invoke the notification method on the target server.

```
void TimerEventCallBack (SOMEEvent event, void *eventData)
{ Environment ev;
  SOM_InitEnvironment(&ev);
  /* call the peer, with a oneway message */
  _noteTimeout(RemotePeer, &ev, thisServer);
}

data = SOMEEMRegisterDataNew();
_someClearRegData(data, &ev);
_someSetRegDataEventMask(data, &ev, EMTimerEvent, NULL);
_someSetRegDataTimerInterval(data, &ev, Interval);
somPrintf("Type <Enter> key when ready to go: ");
gets(inbuf);
regId = _someRegisterProc(MyEMan, &ev, data, TimerEventCallBack,
                          NULL);
```

Important: Do *not* use **someRegister** or **someRegisterEv** to register "callback methods" that would be executed on proxy objects. Instead, write a callback routine that invokes the desired method on the proxy, like the one shown above, and register that routine using the method **someRegisterProc**.

Note: EMan currently uses the methods **someRegister** and **someRegisterEv** to obtain the *address* of a method-procedure to call when a specified event occurs. If EMan directly calls the method-procedure versus **somDispatch**, the method call will not be forwarded to the remote object as desired.

- Start the EMan event processing loop.

```
_someProcessEvents(MyEMan, &ev);
```

Before the sample is run, two server implementations should be registered with **regimpl**. The implementations are identical except for their aliases. One may be called "peerServer1" and the other "peerServer2". The "PeerServer" class should be specified as their server-object class.

Whenever peerServer1's timer pops, the Event Manager causes a method, "noteTimeout", to be sent to the server object in peerServer2. PeerServer2's server object executes this method by displaying a message on its window. Whenever peerServer2's timer pops, a similar sequence occurs with peerServer1. The two servers will run continuously until interrupted.

Dynamic Invocation Interface

DSOM supports the CORBA dynamic invocation interface (DII), which clients can use to dynamically build and invoke requests on objects. This section describes how to use the DSOM DII. Currently, DSOM supports dynamic request invocation only on objects outside the address space of the request initiator, via proxies. The **somDispatch** method (non-CORBA) can be used to invoke methods dynamically on either local or remote objects, however.

To invoke a request on an object using the DII, the client must explicitly construct and initiate the request. A request is comprised of an object reference, an operation, a list of arguments for the operation, and a return value from the operation. A key to proper construction of the request is the correct usage of the **NamedValue** structure and the **NVList** object. The return value for an operation is supplied to the request in the form of a **NamedValue** structure. In addition, it is usually most convenient to supply the arguments for a request in the form of an **NVList** object, which is an ordered set of **NamedValue**s. This section begins with a description of **NamedValue**s and **NVList**s and then details the procedure for building and initiating requests.

The NamedValue structure

The **NamedValue** structure is defined in C as:

```
typedef unsigned long Flags;

struct NamedValue {
        Identifier      name;           // argument name
        any             argument;       // argument
        long            len;            // length/count of arg value
        Flags           arg_modes;      // argument mode flags
};
```

where:

name is an **Identifier** string as defined in the CORBA specification, and
arg is an **any** structure with the following declaration:

```
struct any {
        TypeCode        _type;
        void*           _value;
};
```

_type is a **TypeCode**, which has an opaque representation with operations defined on it to allow access to its constituent parts. Essentially the **Typecode** is composed of a field specifying the CORBA type represented and possibly additional fields needed to fully describe the type. See Chapter 7 of this manual for a complete explanation of **TypeCode**s.

_value is a pointer to the value of the *any* structure. Important: The contents of "_value" should always be a *pointer* to the value, regardless of whether the value is a primitive, a structure, or is itself a pointer (as in the case of object references, strings and arrays). For object references, strings and arrays, *_value* should contain *a pointer to the pointer* that references the value. For example:

```
string          testString;
any             testAny;

testAny._value = &testString;
```

len is the number of bytes that the argument value occupies. The following table gives the length of data values for the C language bindings. The value of *len* must be consistent with the **TypeCode**.

```
Data type               Length
short                   sizeof(short)
unsigned short          sizeof(unsigned short)
long                    sizeof(long)
unsigned long           sizeof(unsigned long)
float                   sizeof(float)
double                  sizeof(double)
char                    sizeof(char)
boolean                 sizeof(boolean)
octet                   sizeof(octet)
string                  strlen(string) - does not include '\0' byte
enum E{}                sizeof(unsigned long)
union U                 sizeof(U)
struct S{}              sizeof(S)
Object                  1
array N of type T1      Length(T1)*N
sequence V of type T2   Length(T2)*V - V is the actual # of elements
```

The *arg_modes* field is a bitmask (unsigned long) and may contain the following flag values:

ARG_IN	the associated value is an input-only argument
ARG_OUT	the associated value is an output-only argument
ARG_INOUT	the associated argument is an in/out argument

These flag values identify the parameter passing mode when the **NamedValue** represents a method parameter. Additional flag values have specific meanings for **Request** and **NVList** methods and are listed with their associated methods.

The NVList class

An **NVList** contains an ordered set of **NamedValue**s. The CORBA specification defines several operations that the **NVList** supports. The IDL prototypes for these methods are as follows:

```
// get the number of elements in the NVList
ORBStatus get_count(
            out  long       count );

// add an element to an NVList
ORBStatus add_item(
            in  Identifier  item_name,
            in  TypeCode    item_type,
            in  void*       value,
            in  Flags       item_flags );

// free the NVList and any associated memory
ORBStatus free();

// free dynamically allocated memory associated with the list
ORBStatus free_memory();
```

In DSOM, the **NVList** is a full-fledged object with methods for getting and setting elements:

```
//set the contents of an element in an NVList
ORBStatus set_item(
            in  long        item_number, /* element # to set */
            in  Identifier  item_name,
            in  TypeCode    item_type,
            in  void*       item_value,
            in  long        value_len,
            in  Flags       item_flags );
```

```
                // get the contents of an element in an NVList
                ORBStatus get_item(
                                in  long        item_number, /* element # to get */
                                out Identifier  item_name,
                                out TypeCode    item_type,
                                out void*       item_value,
                                out long        value_len,
                                out Flags       item_flags );
```

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for a detailed description of the methods defined on the **NVList** object.

Creating argument lists

A very important use of the **NVList** is to pass the argument list for an operation when creating a request. CORBA 1.1 specifies two methods, defined in the **ORB** class, to build an argument list: **create_list** and **create_operation_list**. The IDL prototypes for these methods are as follows:

```
                ORBStatus create_list(
                                in  long        count,    /* # of items */
                                out NVList      new_list );

                ORBStatus create_operation_list(
                                in  OperationDef oper,
                                out NVList       new_list );
```

The **create_list** method returns an **NVList** with the specified number of elements. Each of the elements is empty. It is the client's responsibility to fill the elements in the list with the correct information using the **set_item** method. Elements in the **NVList** must contain the arguments in the same order as they were defined for the operation. Elements are numbered from 0 to count–1.

The **create_operation_list** method returns an **NVList** initialized with the argument descriptions for a given operation (specified by the **OperationDef**). The arguments are returned in the same order as they were defined for the operation. The client only needs to fill in the *item_value* and *value_len* in the elements of the **NVList**.

In addition to these CORBA-defined methods, DSOM provides a third version, defined in the **SOMDObject** class. The IDL prototype for this method is as follows:

```
                ORBStatus create_request_args(
                                in Identifier operation,
                                out NVList arg_list,
                                out NamedValue result );
```

Like **create_operation_list**, the **create_request_args** method creates the appropriate **NVList** for the specified operation. In addition, **create_request_args** initializes the **NamedValue** that will hold the result with the expected return type. The **create_request_args** method is defined as a companion to the **create_request** method, and has the advantage that the **InterfaceDef** for the operation does not have to be retrieved from the Interface Repository.

Note: The **create_request_args** method is *not* defined in CORBA 1.1. Hence, the **create_operation_list** method, defined on the **ORB** class, should be used instead when writing portable CORBA-compliant programs.

Building a Request

There are two ways to build a **Request** object. Both begin by calling the **create_request** method defined by the **SOMDObject** class. The IDL prototype for **create_request** is as follows:

```
                ORBStatus create_request(
                                in   Context     ctx,
                                in   Identifier  operation,
                                in   NVList      arg_list,
                                inout NamedValue result,
                                out  Request     request,
                                in   Flags       req_flags );
```

The *arg_list* can be constructed using the procedures described above and is passed to the **Request** object in the **create_request** call. Alternatively, *arg_list* can be specified as NULL and repetitive calls to **add_arg** can be used to specify the argument list. The **add_arg** method, defined by the **Request** class, has the following IDL prototype:

```
ORBStatus add_arg(
              in  Identifier  name,
              in  TypeCode    arg_type,
              in  void*       value,
              in  long        len,
              in  Flags       arg_flags );
```

The "arg_modes" field of the *result* **NamedValue** parameter to **create_request** is ignored.

Initiating a Request

There are two ways to initiate a request, using either the **invoke** or **send** method defined by the **Request** class. The IDL prototypes for these two operations are as follows:

```
ORBStatus invoke(
              in Flags         invoke_flags );

ORBStatus send(
              in Flags         send_flags );
```

The **invoke** method calls the ORB, which handles the remote method invocation and returns the result. This method will block while awaiting return of the result.

The **send** method calls the ORB but does not wait for the operation to complete before returning. To determine when the operation is complete, the client must call the **get_response** method (also defined by the **Request** class), which has this IDL prototype:

```
ORBStatus get_response(
              in Flags         response_flags );
```

The following flag is defined for **get_response**:

RESP_NO_WAIT   Means that the caller does not want to wait for a response.

get_response determines whether a request has competed. If the RESP_NO_WAIT flag is set, **get_response** returns immediately even if the request is still in progress. If RESP_NO_WAIT is not set, **get_response** waits until the request is done before returning.

Example code

Below is an incomplete example showing how to use the DII to invoke a request having the following method procedure prototype:

```
string _testMethod( testObject        obj,
                    Environment       *ev,
                    long              input_value,
    );
```

```
            main()
            {
                ORBStatus rc;
                Environment ev;
                SOMDObject obj;
                NVList arglist;
                NamedValue result;
                Context ctx;
                Request reqObj;
                OperationDef opdef;
                Description desc;
                OperationDescription opdesc;
                static long input_value = 999;

                SOM_InitEnvironment(&ev);
                SOMD_Init(&ev);

        /* create the argument list */
        /* get the operation description from the interface repository */
            opdef = _lookup_id(SOM_InterfaceRepository, *ev,
                        "testObject::testMethod");
            desc = _describe(opdef, &ev);
            opdesc = (OperationDescription *) desc.value._value;

        /* fill in the TypeCode field for the result */
            result.argument._type = opdesc->result;

        /* Initialize the argument list */
            rc = _create_operation_list(SOMD_ORBObject, &ev, opdef,
                                    &arglist);

        /* get default context */
            rc = _get_default_context(SOMD_ORBObject, &ev, &ctx);

        /* put value and length into the NVList */
            _get_item(arglist, &ev, 0, &name, &tc, &dummy, &dummylen,
                    &flags);

            _set_item(arglist, &ev, 0, name, tc, &input_value,
                    sizeof(input_value),flags);
            ...
        /* create the request – assume the object reference came from
            somewhere -- from a file or returned by a previous request */
            rc = _create_request(obj, &ev, ctx, "testMethod",
                            arglist, &result, &reqObj, (Flags)0);

        /* invoke request */
            rc = invoke(reqObj, &ev, (Flags)0);

        /* print result */
            printf("result: %s\n",*(string*)(result.argument._value));
            return(0);
        }
```

Creating user-supplied proxies

DSOM uses a proxy object in the client's address space to represent the remote object. As mentioned earlier in this chapter, the proxy object encapsulates the operations necessary to forward and invoke methods on the remote object and return the results. By default, proxy generation is done automatically by the DSOM run time. However, if desired, the programmer can cause a user-supplied proxy class to be loaded instead of letting the run time dynamically generate a default proxy class. User-supplied proxies can be useful in specialized circum-stances when local processing or data caching is desired.

To build a user-supplied proxy class, it is necessary to understand a bit about how dynamic proxy classes are constructed by the DSOM run time. The DSOM run time constructs a proxy class by creating an instance of a class that inherits the interface and implementation of **SOMDClientProxy**, and the interface (but not the implementation) of the target class. The methods in the interface of the target object are all overridden to call the **somDispatch** method. (For more details, see "Object references and proxy objects" in section 6.8.)

Every SOM object contains the **somDispatch** method, inherited from **SOMObject**. This method is used to dynamically dispatch a method on an object, and can be overridden with application-specific dispatching mechanisms. In **SOMDClientProxy**, the **somDispatch** method is overridden to forward method calls to the corresponding remote target object.

So, in effect, when a method is called on a default proxy object created by the DSOM run time, it redispatches the method to the remote object using DSOM's version of **somDispatch**.

Below is a simple example of a user-supplied proxy class. In this particular example, the proxy object maintains a local, unshared copy of an attribute ("attribute_long") defined in the remote object ("Foo"), while forwarding method invocations ("method1") on to the remote object. The result is that, when multiple clients are talking to the same remote "Foo" object, each client has a local copy of the attribute but all clients share the "Foo" object's implementation of "method1".

Note: It is important to understand that simply setting the attribute in one client's proxy does *not* affect the value of the attribute in other proxies. Maintaining consistency of the cached data values, if desired, is the responsibility of the user-supplied proxy class.

Following is the IDL file for the "Foo" class:

```
// foo.idl

#include <somdtype.idl>
#include <somobj.idl>

interface Foo : SOMObject
{
        string  method1(out string a, inout long b,
                        in ReferenceData c);
        attribute long attribute_long;

  implementation
  {
    releaseorder: method1, _set_attribute_long,
                  _get_attribute_long;
    dllname="foo.dll";
    somDefaultInit: override;
  };
};
```

The user-supplied proxy class is created by using multiple inheritance between **SOMDClient-Proxy** and the target object (in this case "Foo"). Thus, the IDL file for the user-supplied proxy class "Foo__Proxy" (note the two underscores) is as follows:

```
// fooproxy.idl

#include <somdcprx.idl>
#include <foo.idl>

interface Foo__Proxy : SOMDClientProxy, Foo
{
  implementation
  {
    dllname="fooproxy.dll";
    method1: override;
  };
};
```

When a dynamic proxy class is created by the DSOM run time, the methods inherited from the target class are automatically overridden to use **somDispatch**. When you build a user-supplied proxy, you need to do this explicitly. This is why "method1" is overridden in the implementation section of the "fooproxy.idl" file.

The implementation of "method1", which was added to the template produced by the SOM Compiler, simply calls the **somDispatch** method on "somSelf". Because "Foo__Proxy" has inherited the implementation of **SOMDClientProxy**, calling **somDispatch** within "method1" sends the method to the remote "Foo" object.

```
/* foo.c */

#include <somdtype.h>
#include <fooproxy.ih>

SOM_Scope string  SOMLINK method1(Foo__Proxy somSelf,
                                  Environment *ev,
                                  string* a, long* b,
                                  ReferenceData* c)
{
    string ret_str;
    somId  methodId;

/*  Foo__ProxyData *somThis = Foo__ProxyGetData(somSelf); */
    Foo__ProxyMethodDebug("Foo__Proxy","method1");

    /* redispatch method, remotely */
    methodId = somIdFromString("method1");
    _somDispatch(somSelf, (void**)&ret_str,
                 methodId, somSelf, ev, a, b, c);
    SOMFree(methodId);

    return ret_str;
}
```

In summary, to build a user-supplied proxy class:

- Create the .idl file with the proxy class inheriting from both **SOMDClientProxy** and from the target class.
  Important: The user-supplied proxy class *must be named* "*<targetClassName>_* _**Proxy**" (with two underscores in the name) and **SOMDClientProxy** *must be the first class* in the list of parent classes; for example,

        interface Foo__Proxy : SOMDClientProxy, Foo

  Putting **SOMDClientProxy** first ensures that its version of **somDispatch** will be used to dispatch remote method calls.

  In the implementation section of the .idl file, override all methods that are to be invoked on the target class. Do not override methods that are to be invoked on the local proxy.

- Compile the .idl file. Be sure the Interface Repository gets updated with the .idl file. In the implementation file, for each overridden method, call **somDispatch** with the method name and parameters passed into the overridden method. If the proxy class provides an implementation for the **somInit** or **somDefaultInit** method, then it is important to ensure that calling that method more than once on the same proxy object has no negative effect.

- Build the DLL and place it in one of the directories listed in LIBPATH for AIX and OS/2, or listed in PATH for Windows. Before creating the default proxy, the DSOM run time checks the LIBPATH for a DLL containing the class named "*<targetClassName>_*_**Proxy**". If such a DLL is found, DSOM loads it instead of dynamically generating a proxy class.

Customizing the default base proxy class

Continuing the example from the previous topic, imagine that an application derives 100 subclasses from the "Foo" class. If the application wishes to cache the "Foo::attribute_long" attribute in the proxies for all remote Foo-based objects, the application could supply 100 user-supplied proxy classes, developed in the manner described above. However, this would become a very tedious and repetitive task!

Alternatively, it is possible to provide a customized base proxy class for use in the dynamic generation of DSOM proxy classes. This allows an application to provide a customized base proxy class, from which other dynamic DSOM proxy classes can be derived. This is particularly useful in situations where an application would like to enhance many or all dynamically generated proxy classes with a common feature.

As described in the previous topic, proxy classes are derived from the **SOMDClientProxy** class by default. It is the **SOMDClientProxy** class that overrides **somDispatch** in order to forward method calls to remote objects.

The **SOMDClientProxy** class can be customized by deriving a subclass in the usual way (being careful not to replace **somDispatch** or other methods that are fundamental to implementing the proxy's behavior). To extend the above example further, the application might define a base proxy class called "MyClientProxy" that defines a long attribute called "attribute_long," which will be inherited by Foo-based proxy classes.

The SOM IDL modifier **baseproxyclass** can be used to specify which base proxy class DSOM should use during dynamic proxy-class generation. To continue the example, if the class "MyClientProxy" were used to construct the proxy class for a class "XYZ," then the **baseproxyclass** modifier would be specified as follows:

```
// xyz.idl

#include <somdtype.idl>
#include <foo.idl>

interface XYZ : Foo
{
  ...
  implementation
  {
    ...
    baseproxyclass = MyClientProxy;
  };
};
```

It should be noted that:

- Base proxy classes must be derived from **SOMDClientProxy**.

- If a class "XYZ" specifies a custom base-proxy class, as in the above example, subclasses of "XYZ" do *not* inherit the value of the **baseproxyclass** modifier. If needed, the **baseproxyclass** modifier must be specified explicitly in each class.

Sockets class

To aid in portability, DSOM has been written to use a *common communications interface*, which is implemented by one or more available local protocols.

The common communications interface is represented as an abstract class, called **Sockets**, and is based on the familiar "sockets" interface. Several protocol implementations are supported as **Sockets** subclasses: **TCPIPSockets** (and **TCPIPSockets32** for OS/2) for TCP/IP, the class **NBSockets** for NetBIOS, and the class **IPXSockets** for NetWare IPX/SPX. (The libraries included in a particular SOMobjects run-time package will vary.)

There is one case where a **Sockets** subclass is not required: the DSOM Workstation run-time package uses shared memory to pass messages within a single machine, and bypasses the use of a **Sockets** subclass. (The SOMSOCKETS environment variable is ignored.)

When the Event Management Framework (EMan) is used with DSOM, a **Sockets** subclass will be needed to support EMan, whether or not the application runs completely within a single machine.

Appendix C, "Implementing Sockets Subclasses," describes how an application might provide its own **Sockets** subclass implementation, for special circumstances.

6.10 Error Reporting and Troubleshooting

Error reporting

When the DSOM run-time environment encounters an error during execution of a method or procedure, a SYSTEM_EXCEPTION will be raised. The standard system exceptions are discussed in the topic "Exceptions and Error Handling" in Chapter 3 "Using SOM Classes in Client Programs." The "minor" field of the returned exception value will contain a DSOM error code. The DSOM error codes are listed below.

Although a returned exception value can indicate that a DSOM run-time error occurred, it may be difficult for the user or application to determine what caused the error. Consequently, DSOM has been enabled to report run-time error information, for interpretation by support personnel. These error messages take the following form:

DSOM *<type>* **error:** *<code>* [*<name>*] at *<file>*:*<line>*

where the arguments are as follows:

type	SYSTEM_EXCEPTION type,
code	DSOM error code,
name	symbol for DSOM error code (from "somderr.h"),
file	source-file name where the error occurred, and
line	line number where the error occurred.

For example,

```
DSOM NO_MEMORY error: 30001 [SOMDERROR_NoMemory] at somdobj.c:250
```

indicates that a "NO_MEMORY" error (error code 30001) occurred in file "somdobj.c" at line 250. This information is not usually meaningful to the user; it provides support personnel with a starting point for problem analysis. There will often be a sequence of error messages; together they indicate the context in which the error occurred. It is important to give all reported messages to support personnel for analysis.

There is an environment variable, SOMDDEBUG, which is used to activate error reporting. There is a corresponding global variable that can be set by an application program; it is declared as:

```
extern long SOMD_DebugFlag;
```

Error reporting is normally disabled. To enable error reporting, the environment variable SOMD-DEBUG should be set to a value greater than 0. To disable error reporting, SOMDDEBUG should be set to a value less than or equal to 0.

By default, error messages will display on the standard output device. Error messages can also be redirected to a log file. For this, the environment variable SOMDMESSAGELOG should be set to the pathname of the log file. The **SOMD_Init** procedure opens the file named in SOMD-MESSAGELOG (if any), during process initialization.

Error codes

The error codes that may be encountered when using DSOM are listed in Appendix A, "SOMobjects Error Codes," which contains the codes for the entire SOMobjects Toolkit.

Troubleshooting hints

The following hints may prove helpful as you develop and test your DSOM application.

Checking the DSOM setup

This checklist will help you make certain that the DSOM environment is set up correctly.

1. Use the **somdchk** tool to verify the settings of various DSOM environment variables. Note that if the SOMIR environment variable contains relative pathnames, the client programs, server programs, and **somdd** daemon must be started from the same directory. (Instead, it is recommended that SOMIR contain full pathnames.)

2. For all application classes, IDL must be compiled into the Interface Repository. You can verify that a class exists in the Interface Repository by executing "**irdump** <*class*>" on AIX or OS/2, or, on Windows, click on the IRDUMP icon and browse for the appropriate class. See "Registering class interfaces" for more information.

3. An implementation (a server program and class libraries) must be registered with DSOM by running the **regimpl** utility. See "Registering servers and classes" for more information.

4. Verify that all class libraries and networking libraries are in directories specified in LIBPATH.

Analyzing problem conditions

The DSOM error codes mentioned below can be obtained directly by the application from the "minor" field of the exception data returned in a system exception, or from an error report message when SOMDDEBUG is set to a positive integer value (see the earlier topic, "Error reporting").

Symptom: When running **regimpl**, a "PERSIST_STORE" exception is returned. The DSOM error code is SOMDERROR_IO or SOMDERROR_NoImplDatabase.

• This may indicate that the Implementation Repository files are not found or cannot be accessed. Verify that SOMDDIR is set correctly, to a directory that has granted read/write permission to the DSOM user. (It is best if the directory name is fully qualified.) If the SOMDDIR variable is not set, verify that the default directory ($SOMBASE/etc/dsom on AIX, and %SOMBASE%\etc\dsom on OS/2 or Windows) has been set up with the correct permissions. Ensure that the the files contained in the directory all have read/write permission granted to the DSOM user.

Symptom: When starting **somdd**, a "COMM_FAILURE" exception is returned.

• If the DSOM error code is SOMDERROR_SocketBind or SOMDERROR_DuplicateQueue, this may indicate that there is already an instance of **somdd** running. If this is true, and the current instance of **somdd** does not seem to be responding properly, delete all instances of **somdd** and restart a new copy of **somdd**.

Otherwise, some other communications error caused the failure. Verify that the DSOM environment variables (including HOSTNAME) are set correctly. Also, if SOMSOCKETS is set, verify that the IR contains the appropriate **Sockets** interface definition.

Symptom: When starting up a server program, an exception is returned with a DSOM error code of SOMDERROR_ServerAlreadyExists.

• This may indicate that a server process that is already running has already registered itself with the DSOM daemon, **somdd**, using the implementation ID of the desired server program.

Symptom: On OS/2, an operating system error occurs indicating a "stack overflow" condition soon after a the first call to a class DLL. Rebuilding the DLL with a larger stack size does not help.

• This sometimes occurs when a DLL uses, but does not initialize, the C run time for OS/2. To perform the proper initialization, a function named "_DLL_InitTerm" must be included in the DLL, and it must invoke the C Set/2 initialization function, "_CRT_Init." The _DLL_InitTerm function is invoked automatically when the DLL is loaded.

An example _DLL_InitTerm function is included in the DSOM sample code shipped with the SOMobjects Developer Toolkit for OS/2, in the file "initterm.c".

Symptom: When running a DSOM application that uses EMan, an error message is displayed asking that the SOMSOCKETS be set.

• This may indicate a need to specify the **Sockets** subclass to be used with the application. Current choices are **TCPIPSockets** for TCP/IP (and **TCPIPSockets32** on OS/2), **NBSockets** for NetBIOS, and **IPXSockets** for NetWare IPX/SPX. Note: Each SOMobjects package contains an appropriate subset of **Sockets** subclasses. Workstation DSOM includes **TCPIP-Sockets** (and **TCPIPSockets32** on OS/2) for TCP/IP. Workgroup DSOM also includes **NBSockets** for NetBIOS and **IPXSockets** for NetWare.

Symptom: A remote method invocation fails and a "DISPATCH" exception is returned. The DSOM error code is SOMDERROR_BadDescriptor.

- This may indicate that the interface describing the method cannot be found in the Interface Repository. Verify that SOMIR is set correctly, and that the IR contains all interfaces used by your application.

    If the default SOM IR (supplied with the SOMobjects Toolkit and Runtimes) is not used by the application, the user's IR must include the interface definitions for the appropriate **Sockets** class, server class (derived from **SOMDServer**), and the definitions of the standard DSOM exceptions (found in file "stexcep.idl") that may be returned by a method call.

Symptom: A SOMDERROR_ClassNotFound error is returned by a client either when creating a remote object using **somdNewObject**, or when finding a server object using **somdFindAnyServerByClass**. (These methods are defined on the **SOMDObjectMgr** class.)

- This sometimes occurs when the class name specified in calls to **somdNewObject** or **somdFindAnyServerByClass** cannot be found in the Implementation Repository. Make sure that the class name has been associated with at least one of the server implementations.

Symptom: A SOMDERROR_ClassNotFound error is returned by a server when creating a new object using **somdNewObject**, **somdCreateObj**, or **somdGetClassObj**.

- This error may result if the DLL for the class cannot be found. Verify that:

    — the interface of the object can be found in the IR;

    — the class name is spelled correctly and is appropriately scoped (for example, the "Printer" class in the "PrintServer" module must have the identifier "PrintServer::Printer").

- This error can also result when the shared library is statically linked to the server program, but the *<className>***NewClass** procedures have not been called to initialize the classes.

Symptom: When invoking a method returns a proxy for a remote object in the client, a SOMDERROR_NoParentClass error occurs.

- This error may result when the class libraries used to build the proxy class are statically linked to the program, but the *<className>***NewClass** procedures have not been called to initialize the classes.

Symptom: Following a method call, the SOM run-time error message, "A target object failed basic validity checks during method resolution" is displayed.

- Usually this means that the method call was invoked using a bad object pointer, or the object has been corrupted.

Symptom: A remote object has an attribute or instance variable that is, or contains, a pointer to a value in memory (for example, a string, a sequence, an "any"). The attribute or instance variable value is set by the client with one method call. When the attribute or instance variable is queried in a subsequent method call, the value referenced by the pointer is "garbage".

- This may occur because DSOM makes *a copy* of argument values in a client call, for use in the remote call. The argument values are valid for the duration of that call. When the remote call is completed, the copies of the argument values are freed.

    In a DSOM application, a class cannot assume that the client has allocated and will manage the space for object instance values, because the client may be in a different address space. If a data value is meant to persist between requests, the object is responsible for making its own copy of the value.

Symptom: A method defines a (char *) parameter that is used to pass a string input value to an object. The object attempts to print the string value, but it appears to be "garbage".

- DSOM *will* support method arguments that are of type "pointer-to-X" (pointer types are a SOM extension), by deferencing the pointer in the call, and copying the base value. The pointer-to-value is reconstructed on the server before the actual method call is made.

    While (char *) is commonly used to refer to NULL-terminated strings in C programs, (char *) could also be a pointer to a single character or to an array of characters. Thus, DSOM interprets the argument type literally as a pointer-to-one-character.

To correctly pass strings or array arguments, the appropriate CORBA type should be used (for example, "string" or "char foo[4]").

Symptom: A segmentation violation occurs when passing an "any" argument to a method call, where the "any" value is a string, array, or object reference. Note: The **NamedValue**s used in DII calls use "any" fields for the argument values.

- This error may occur because the "_value" field of the "any" structure does not contain the address of a *pointer* to the target string, array, or object reference, as it should. (A common mistake is to set the "_value" field to the address of the string, array, or object reference itself.)

Symptom: When a server program or a server object makes a call to **get_id** or to **get_SOM_object** on a **SOMDObject**, an "OBJ_ADAPTER" exception is returned with an error code of SOMDERROR_WrongRefType.

- This error may occur when the operation **get_id** is called on a **SOMDObject** that does not have any user-supplied **ReferenceData** (that is, the **SOMDObject** is a proxy, is nil, or is a simple "SOM ref" created by **create_SOM_ref**). Likewise, this error may occur when the operation **get_SOM_object** is called on a **SOMDObject** that was not created by the **create_SOM_ref** method.

Symptom: A segmentation fault occurs when a **SOMD_Uninit** call is executed.

- This error could occur if the application has already freed any of the DSOM run-time objects that were allocated by the **SOMD_Init** call, including **SOMD_ObjectMgr**, **SOMD_ImplRepObject** and **SOMD_ORBObject**.

Symptom: Unexplained program crashes.

- Verify that all DSOM environment variables are set, as described in the earlier section "Configuring DSOM Applications". Verify that all class libraries are in directories specified in LIBPATH for AIX and OS/2, or specified in PATH for Windows. Verify that the contents of the Interface Repository, specified by SOMIR, are correct. Verify that the contents of the Implementation Repository, specified by SOMDDIR, are correct. Verify that **somdd** is running. Set SOMDDEBUG to 1 to obtain additional DSOM error messages.

6.11 Limitations

The following list indicates known limitations of Distributed SOM at the time of this release.

1. Currently, objects cannot be moved from one server to another without changing the object references (that is, deleting the object, and creating it anew in another server). This yields all copies of the previous reference invalid.

2. The **change_implementation** method is not supported. This method, defined by the **BOA** interface, is intended to allow an application to change the implementation definition associated with an object. However, in DSOM, changing the server implementation definition may render existing object references (which contain the old server ID) invalid.

3. Currently, DSOM has a single server activation policy, which corresponds to CORBA's "shared" activation policy for dynamic activation, and "persistent" activation policy for manual activation. Other activation policies, such as "server-per-method" and "unshared" are not directly supported, and must be implemented by the application.

   Since the unshared server policy is not directly supported, the **obj_is_ready** and **deactivate_obj** methods, defined in the **BOA** interface, have null implementations.

4. If a server program terminates without calling **deactivate_impl**, subsequent attempts to start that server may fail. The DSOM daemon believes the server is still running until it is told it has stopped. Attempts to start a server that is believed to exist results in an error (SOMDERROR_ServerAlreadyExists).

5. Currently, file names used in **ImplementationDef**s are limited to 255 bytes. Implementations aliases used in **ImplementationDef**s are limited to 50 bytes. Class names used in the Implementation Repository are limited to 50 bytes. Hostnames are limited to 32 bytes.

6. The OUT_LIST_MEMORY, IN_COPY_VALUE, and DEPENDENT_LIST flags, used with the Dynamic Invocation Interface, are not yet supported.

Other important notes concerning DSOM are documented in the "README" file in the SOMBASE root directory ($SOMBASE on AIX, and %SOMBASE% on OS/2 or Windows).

Chapter 7. The Interface Repository Framework

7.1 Introduction

The SOM Interface Repository (IR) is a <u>database</u> that the SOM Compiler optionally creates and maintains from the information supplied in IDL source files. The Interface Repository contains persistent objects that correspond to the major elements in IDL descriptions. The SOM Interface Repository <u>Framework</u> is a <u>set of classes</u> that provide methods whereby executing programs can access these objects to discover everything known about the programming interfaces of SOM classes.

The programming interfaces used to interact with Interface Repository objects, as well as the format and contents of the information they return, are architected and defined as part of the Object Management Group's CORBA standard. The classes composing the SOM Interface Repository Framework implement the programming interface to the CORBA Interface Repository. Accordingly, the SOM Interface Repository Framework supports all of the interfaces described in *The Common Object Request Broker: Architecture and Specification* (OMG Document Number 91.12.1, Revision 1.1, chapter 7).

As an extension to the CORBA standard, the SOM Interface Repository Framework permits storage in the Interface Repository of arbitrary information in the form of SOM IDL **modifiers**. That is, within the SOM-unique **implementation** section of an IDL source file or through the use of the **#pragma modifier** statement, user-defined modifiers can be associated with any element of an IDL specification. (See the section entitled "SOM Interface Definition Language" in Chapter 4, "Implementing Classes in SOM.") When the SOM Compiler creates the Interface Repository from an IDL specification, these potentially arbitrary modifiers are stored in the IR and can then be accessed via the methods provided by the Interface Repository Framework.

This chapter describes, first, how to build and manage interface repositories, and second, the programming interfaces embodied in the SOM Interface Repository Framework.

7.2 Using the SOM Compiler to Build an Interface Repository

The SOMobjects Toolkit includes an Interface Repository emitter that is invoked whenever the SOM Compiler is run using an **sc** command with the **–u** option (which "updates" the interface repository). The IR emitter can be used to create or update an Interface Repository file. The IR emitter expects that an environment variable, SOMIR, was first set to designate a file name for the Interface Repository. For example, to compile an IDL source file named "newcls.idl" and create an Interface Repository named "newcls.ir", use a command sequence similar to the following:

For OS/2:

```
set SOMIR=c:\myfiles\newcls.ir
sc –u newcls
```

For AIX:

```
export SOMIR=~/newcls.ir
sc –u newcls
```

For Windows:
Note: Ensure that no spaces separate the environment variable "SOMIR", the equals sign "=", and the value being set.

```
set SOMIR=c:\myfiles\newcls.ir
somc –u newcls
```

If the SOMIR environment variable is not set, the Interface Repository emitter creates a file named "som.ir" in the current directory.

The **sc** or **somc** command runs the Interface Repository emitter plus any other emitters indicated by the environment variable SMEMIT (described in the topic "Running the SOM Compiler" in Chapter 4, "Implementing Classes in SOM"). To run the Interface Repository emitter by itself, issue the **sc** or **somc** command with the **–s** option (which overrides SMEMIT) set to "ir". For example:

```
sc –u –sir newcls          (On OS/2 or AIX)
somc –u –sir newcls        (On Windows)
```

or equivalently,

```
sc –usir newcls            (On OS/2 or AIX)
somc –usir newcls          (On Windows)
```

The Interface Repository emitter uses the SOMIR environment variable to locate the designated IR file. If the file does not exist, the IR emitter creates it. If the named interface repository already exists, the IR emitter checks all of the "type" information in the IDL source file being compiled for internal consistency, and then changes the contents of the interface repository file to agree with with the new IDL definition. For this reason, the use of the –u compiler flag requires that all of the types mentioned in the IDL source file must be fully defined within the scope of the compilation. Warning messages from the SOM Compiler about undefined types result in actual error messages when using the –u flag.

The additional type checking and file updating activity implied by the –u flag increases the time it takes to run the SOM Compiler. Thus, when developing an IDL class description from scratch, where iterative changes are to be expected, it may be preferable *not* to use the –u compiler option until the class definition has stabilized.

7.3 Managing Interface Repository files

Just as the number of interface definitions contained in a single IDL source file is optional, similarly, the number of IDL files compiled into one interface repository file is also at the programmer's discretion. Commonly, however, all interfaces needed for a single project or class framework are kept in one interface repository.

The SOM IR file "som.ir"

The SOMobjects Toolkit includes an Interface Repository file ("som.ir") that contains objects describing all of the types, classes, and methods provided by the various frameworks of the SOMobjects Toolkit. Since all new classes will ultimately be derived from these predefined SOM classes, some of this information also needs to be included in a programmer's own interface repository files.

For example, suppose a new class, called "MyClass", is derived from **SOMObject**. When the SOM Compiler builds an Interface Repository for "MyClass", that IR will also include all of the information associated with the **SOMObject** class. This happens because the **SOMObject** class definition is inherited by each new class; thus, all of the **SOMObject** methods and typedefs are implicitly contained in the new class as well.

Eventually, the process of deriving new classes from existing ones would lead to a great deal of duplication of information in separate interface repository files. This would be inefficient, wasteful of space, and extremely difficult to manage. For example, to make an evolutionary change to some class interface, a programmer would need to know about and subsequently update all of the interface repository files where information about that interface occurred.

One way to avoid this dilemma would be to keep all interface definitions in a single interface repository (such as "som.ir"). This is not recommended, however. A single interface repository would soon grow to be unwieldy in size and become a source of frequent access contention. Everyone involved in developing class definitions would need update access to this one file, and simultaneous uses might result in longer compile times.

Managing IRs via the SOMIR environment variable

The SOMobjects Toolkit offers a more flexible approach to managing interface repositories. The SOMIR environment variable can reference an ordered list of separate IR files, which process from left to right. Taken as a whole, however, this gives the appearance of a single, logical interface repository. A programmer accessing the contents of "the interface repository" through the SOM Interface Repository framework would not be aware of the division of information across separate files. It would seem as though all of the objects resided in a single interface repository file.

A typical way to utilize this capability is as follows:

- The first (leftmost) Interface Repository in the SOMIR list would be "som.ir". This file contains the basic interfaces and types needed in all SOM classes.

- The second file in the list might contain interface definitions that are used globally across a particular enterprise.

- A third interface repository file would contain definitions that are unique to a particular department, and so on.

- The final interface repository in the list should be set aside to hold the interfaces needed for the project currently under development.

Developers working on different projects would each set their SOMIR environment variables to hold slightly different lists. For the most part, the leftmost portions of these lists would be the same, but the rightmost interface repositories would differ. When any given developer is ready to share his/her interface definitions with other people outside of the immediate work group, that person's interface repository can be promoted to inclusion in the master list.

With this arrangement of IR files, the more stable repositories are found at the left end of the list. For example, a developer should never need to make any significant changes to "som.ir", because these interfaces are defined by IBM and would only change with a new release of the SOMobjects Toolkit.

The Interface Repository Framework only permits updates in the <u>rightmost file</u> of the SOMIR interface repository list. That is, when the SOM Compiler –u flag is used to update the Interface Repository, only the final file on the IR list will be affected. The information in all preceding interface repository files is treated as "read only". Therefore, to change the definition of an interface in one of the more global interface repository files, a developer must overtly construct a special SOMIR list that omits all subsequent (that is, further to the right) interface repository files, or else petition the owner of that interface to make the change.

It is important that the rightmost filename in the SOMIR interface repository list *not* appear elsewhere in the list. For example, the following setting for SOMIR:

```
%SOMBASE%\ETC\SOM.IR;SOM.IR;C:\IR\COMPANY.IR;SOM.IR
```

would cause problems when attempting to update the SOM.IR file, because SOM.IR appears twice in the list.

Here is an example that illustrates the use of multiple IR files with the SOMIR environment variable. In this example, the SOMBASE environment variable represents the directory in which the SOMobjects Toolkit files have been installed. Only the "myown.ir" interface repository file will be updated with the interfaces found in files "myclass1.idl", "myclass2.idl", and "myclass3.idl".

For OS/2:

```
set BASE_IRLIST=%SOMBASE%\IR\SOM.IR;C:\IR\COMPANY.IR;C:\IR\DEPT10.IR
set SOMIR=%BASE_IRLIST%;D:\MYOWN.IR
set
SMINCLUDE=.;%SOMBASE%\INCLUDE;C:\COMPANY\INCLUDE;C:\DEPT10\INCLUDE
sc -usir myclass1
sc -usir myclass2
sc -usir myclass3
```

For AIX:

```
export BASE_IRLIST=$SOMBASE/ir/som.ir:/usr/local/ir/company.ir:\
     /usr/local/ir/dept10.ir
export SOMIR=$BASE_IRLIST:~/myown.ir
export SMINCLUDE=.:$SOMBASE/INCLUDE:/usr/local/company/include:\
     /usr/local/dept10/include
sc -usir myclass1
sc -usir myclass2
sc -usir myclass3
```

For Windows:
The following example (of multiple IR files with the SOMIR environment variable) will work correctly only if it is executed from within a .BAT file. (Otherwise, the %BASE_IRLIST% is not interpreted but is taken literally.)

```
set BASE_IRLIST=%SOMBASE%\IR\SOM.IR;C:\IR\COMPANY.IR;C:\IR\DEPT10.IR
set SOMIR=%BASE_IRLIST%;D:\MYOWN.IR
set
SMINCLUDE=.;%SOMBASE%\INCLUDE;C:\COMPANY\INCLUDE;C:\DEPT10\INCLUDE
somc -usir myclass1
somc -usir myclass2
somc -usir myclass3
```

Placing 'private' information in the Interface Repository

When the SOM Compiler updates the Interface Repository in response to the **–u** flag, it uses all of the information available from the IDL source file. However, if the __PRIVATE__ preprocessor variable is used to designate certain portions of the IDL file as private, the preprocessor actually removes that information before the SOM Compiler sees it. Consequently, private information will not appear in the Interface Repository unless the **–p** compiler option is also used in conjunction with **–u.** For example:

```
sc -up myclass1          (On AIX or OS/2)
somc -up myclass1        (On Windows)
```

This command will place all of the information in the "myclass1.idl" file, including the private portions, in the Interface Repository.

If you are using tools that understand SOM and rely on the Interface Repository to describe the types and instance data in your classes, you may need to include the private sections from your IDL source files when building the Interface Repository.

7.4 Programming with the Interface Repository Objects

The SOM Interface Repository Framework provides an object-oriented programming interface to the IDL information processed by the SOM Compiler. Unlike many frameworks that require you to inherit their behavior in order to use it, the Interface Repository Framework is useful in its own right as a set of predefined objects that you can access to obtain information. Of course, if you need to subclass a class to modify its behavior, you can certainly do so; but typically this is not necessary.

The SOM Interface Repository contains the fully-analyzed (compiled) contents of all information in an IDL source file. This information takes the the form of persistent objects that can be accessed from a running program. There are ten classes of objects in the Interface Repository that correspond directly to the major elements in IDL source files; in addition, one instance of another class exists outside of the IR itself, as follows:

Contained — All objects in the Interface Repository are instances of classes derived from this class and exhibit the common behavior defined in this interface.

Container — Some objects in the Interface Repository hold (or contain) other objects. (For example, a module [**ModuleDef**] can contain an interface [**InterfaceDef**].) All Interface Repository objects that hold other objects are instances of classes derived from this class and exhibit the common behavior defined by this class.

ModuleDef — An instance of this class exists for each **module** defined in an IDL source file. **ModuleDef**s are **Container**s, and they can hold **ConstantDef**s, **TypeDef**s, **ExceptionDef**s, **InterfaceDef**s, and other **ModuleDef**s.

InterfaceDef — An instance of this class exists for each **interface** named in an IDL source file. (One **InterfaceDef** corresponds to one SOM class.) **InterfaceDef**s are **Container**s, and they can hold **ConstantDef**s, **TypeDef**s, **ExceptionDef**s, **AttributeDef**s, and **OperationDef**s.

AttributeDef — An instance of this class exists for each **attribute** defined in an IDL source file. **AttributeDef**s are found only inside of (contained by) **InterfaceDef**s.

OperationDef — An instance of this class exists for each **operation** (method) defined in an IDL source file. **OperationDef**s are **Container**s that can hold **ParameterDef**s. **OperationDef**s are found only inside of (contained by) **InterfaceDef**s.

ParameterDef — An instance of this class exists for each **parameter** of each operation (method) defined in an IDL source file. **ParameterDef**s are found only inside of (contained by) **OperationDef**s.

TypeDef — An instance of this class exists for each **typedef**, **struct**, **union**, or **enum** defined in an IDL source file. **TypeDef**s may be found inside of (contained by) any Interface Repository **Container** except an **OperationDef**.

ConstantDef — An instance of this class exists for each **constant** defined in an IDL source file. **ConstantDef**s may be found inside (contained by) of any Interface Repository **Container** except an **OperationDef**.

ExceptionDef — An instance of this class exists for each **exception** defined in an IDL source file. **ExceptionDef**s may be found inside of (contained by) any Interface Repository **Container** except an **OperationDef**.

Repository	— One instance of this class exists for the entire SOM Interface Repository, to hold IDL elements that are global in scope. The instance of this class does not, however, reside within the IR itself.

Methods introduced by Interface Repository classes

The Interface Repository classes introduce nine new methods, which are briefly described below. Many of the classes simply override methods to customize them for the corresponding IDL element; this is particularly true for classes representing IDL elements that are only contained within another syntactic element. Full descriptions of each method are found in the *SOMobjects Developer Toolkit: Programmers Reference Manual.*

• **Contained** class methods (*all* IR objects are instances of this class and exhibit this behavior):

describe	— Returns a structure of type **Description** containing all information defined in the IDL specification of the syntactic element corresponding to the target **Contained** object. For example, for a target **InterfaceDef** object, the **describe** method returns information about the IDL interface declaration. The **Description** structure contains a "name" field with an identifier that categorizes the description (such as, "InterfaceDescription") and a "value" field holding an "any" structure that points to another structure containing the IDL information for that particular element (in this example, the interface's IDL specifications).
within	— Returns a sequence designating the object(s) of the IR within which the target **Contained** object is contained. For example, for a target **TypeDef** object, it might be contained within any other IR object(s) except an **OperationDef** object.

• **Container** class methods (*some* IR objects contain other objects and exhibit this behavior):

contents	— Returns a sequence of pointers to the object(s) of the IR that the target **Container** object contains. (For example, for a target **InterfaceDef** object, the **contents** method returns a pointer to each IR object that corresponds to a part of the IDL interface declaration.) The method provides options for excluding inherited objects or for limiting the search to only a specified kind of object (such as **Attribute-Def**s).
describe_contents	— Combines the **describe** and **contents** methods; returns a sequence of **ContainerDescription** structures, one for each object contained by the target **Container** object. Each structure has a pointer to the related object, as well as "name" and "value" fields resulting from the **describe** method.
lookup_name	— Returns a sequence of pointers to objects of a given name contained within a specified **Container** object, or within (sub)objects contained in the specified **Container** object.

• **ModuleDef** class methods:

	— Override **describe** and **within.**

• **InterfaceDef** class methods:

describe_interface	— Returns a description of all methods and attributes of a given interface definition object that are held in the Interface Repository.
	— Also overrides **describe** and **within.**

- **AttributeDef** class method:

  — Overrides **describe.**

- **OperationDef** class method:

  — Overrides **describe.**

- **ParameterDef** class method:

  — Overrides **describe.**

- **TypeDef** class method:

  — Overrides **describe.**

- **ConstantDef** class method:

  — Overrides **describe.**

- **ExceptionDef** class method:

  — Overrides **describe.**

- **Repository** class methods:

lookup_id	— Returns the **Contained** object that has a specified **RepositoryId.**
lookup_modifier	— Returns the string value held by a SOM or user-defined **modifier**, given the name and type of the modifier, and the name of the object that contains the **modifier.**
release_cache	— Releases, from the internal object cache, the storage used by all currently unreferenced Interface Repository objects.

Accessing objects in the Interface Repository

As mentioned above, one instance of the **Repository** class exists for the entire SOM Interface Repository. This object does not, itself, reside in the Interface Repository (hence it does not exhibit any of the behavior defined by the **Contained** class). It is, however, a **Container**, and it holds all **ConstantDef**s, **TypeDef**s, **ExceptionDef**s, **InterfaceDef**s, and **ModuleDef**s that are global in scope (that is, not contained inside of any other **Container**s).

When any method provided by the **Repository** class is used to locate other objects in the Interface Repository, those objects are automatically instantiated and activated. Consequently, when the program is finished using an object from the Interface Repository, the client code should release the object using the **somFree** method.

All objects contained in the Interface Repository have both a "name" and a "Repository ID" associated with them. The name is not guaranteed to be unique, but it does uniquely identify an object within the context of the object that contains it. The Repository ID of each object is guaranteed to uniquely identify that object, regardless of its context.

For example, two **TypeDef** objects may have the same name, provided they occur in separate name scopes (**ModuleDef**s or **InterfaceDef**s). In this case, asking the Interface Repository to locate the **TypeDef** object based on its name would result in both **TypeDef** objects being returned. On the other hand, if the name is looked up from a particular **ModuleDef** or **InterfaceDef** object, only the **TypeDef** object within the scope of that **ModuleDef** or **InterfaceDef** would be returned. By contrast, once the Repository ID of an object is known, that object can always be directly obtained from the **Repository** object via its Repository ID.

C or C++ programmers can obtain an instance of the **Repository** class using the **RepositoryNew** macro. Programmers using other languages (and C/C++ programmers without static linkage to the **Repository** class) should invoke the method **somGetInterfaceRepository** on the **SOMClassMgrObject**. For example,

For C or C++ (static linkage):

```
#include <repostry.h>
Repository repo;

...
repo = RepositoryNew();
```

From other languages (and for dynamic linkage in C/C++):

1. Use the **somEnvironmentNew** function to obtain a pointer to the **SOMClassMgrObject**, as described in Chapter 3, "Using SOM Classes in Client Programs."

2. Use the **somResolve** or **somResolveByName** function to obtain a pointer to the **somGetInterfaceRepository** method procedure.

3. Invoke the method procedure on the **SOMClassMgrObject**, with no additional arguments, to obtain a pointer to the **Repository** object.

After obtaining a pointer to the **Repository** object, use the methods it inherits from **Container** or its own **lookup_id** method to instantiate objects in the Interface Repository. As an example, the **contents** method shown in the C fragment below activates every object with global scope in the Interface Repository and returns a sequence containing a pointer to every global object:

```
#include <containd.h>        /* Behavior common to all IR objects */
Environment *ev;
int i;
sequence(Contained) everyGlobalObject;

ev = SOM_CreateLocalEnvironment(); /* Get an environment to use */
printf ("Every global object in the Interface Repository:\n");

everyGlobalObject = Container_contents (repo, ev, "all", TRUE);

for (i=0; i < everyGlobalObject._length; i++) {
    Contained aContained;

    aContained = (Contained) everyGlobalObject._buffer[i];
    printf ("Name: %s, Id: %s\n",
        Contained__get_name (aContained, ev),
        Contained__get_id (aContained, ev));
    SOMObject_somFree (aContained);
}
```

Taking this example one step further, here is a complete program that accesses every object in the entire Interface Repository. It, too, uses the **contents** method, but this time recursively calls the **contents** method until every object in every container has been found:

```
#include <stdio.h>
#include <containd.h>
#include <repostry.h>

void showContainer (Container c, int *next);

main ()
{
    int count = 0;
    Repository repo;

    repo = RepositoryNew ();
    printf ("Every object in the Interface Repository:\n\n");
    showContainer ((Container) repo, &count);
    SOMObject_somFree (repo);
    printf ("%d objects found\n", count);
    exit (0);
}

void showContainer (Container c, int *next)
{
    Environment *ev;
    int i;
    sequence(Contained) everyObject;

    ev = SOM_CreateLocalEnvironment (); /* Get an environment */
    everyObject = Container_contents (c, ev, "all", TRUE);
```

```
        for (i=0; i<everyObject._length; i++) {
            Contained aContained;

            (*next)++;
            aContained = (Contained) everyObject._buffer[i];
            printf ("%6d. Type: %-12s id: %s\n", *next,
                SOMObject_somGetClassName (aContained),
                Contained__get_id (aContained, ev));
            if (SOMObject_somIsA (aContained, _Container))
                showContainer ((Container) aContained, next);
            SOMObject_somFree (aContained);
        }
    }
```

Once an object has been retrieved, the methods and attributes appropriate for that particular object can then be used to access the information contained in the object. The methods supported by each class of object in the Interface Repository, as well as the classes themselves, are documented in the *SOMobjects Developer Toolkit: Programmers Reference Manual.*

A word about memory management

Several conventions are built into the SOM Interface Repository with regard to memory management. You will need to understand these conventions to know when it is safe and appropriate to free memory references and also when it is your responsibility to do so.

All methods that access <u>attributes</u> (such as, the `_get_<attribute>` methods) always return either simple values or direct references to data within the target object. This is necessary because these methods are heavily used and must be fast and efficient. Consequently, you should never free any of the memory references obtained through attributes. This <u>memory will be released automatically</u> when the object that contains it is freed.

For all <u>methods that give out object references</u> (there are five: **within, contents, lookup_name, lookup_id,** and **describe_contents**), when finished with the object, you are expected to <u>release the object reference by invoking the **somFree**</u> method. (This is illustrated in the sample program that accesses all Interface Repository objects.) Do not release the object reference until you have either copied or finished using all of the information obtained from the object.

The **describe** methods (**describe, describe_contents,** and **describe_interface**) return <u>structures and sequences</u> that contain information. The actual structures returned by these methods are passed by value (and hence should only be freed if you have allocated the memory used to receive them). However, you may be required to <u>free some of the information contained in the returned structures</u> when you are finished. Consult the specific method in the *SOMobjects Developer Toolkit: Programmers Reference Manual* for more details about what to free.

During execution of the **describe** and **lookup** methods, sometimes <u>intermediate objects</u> are activated automatically. These objects are kept in an internal cache of objects that are in use, but for which no explicit object references have been returned as results. Consequently, there is no way to identify or free these objects individually. However, whenever your program is finished using all of the information obtained thus far from the Interface Repository, <u>invoking the</u> **release_cache** <u>method</u> causes the Interface Repository to purge its internal cache of these implicitly referenced objects. This cache will replenish itself automatically if the need to do so subsequently arises.

Using TypeCode pseudo-objects

Much of the detailed information contained in Interface Repository objects is represented in the form of **TypeCode**s. **TypeCode**s are complex data structures whose actual representation is hidden. A **TypeCode** is <u>an architected way of describing in complete detail everything that is known about a particular data type</u> in the IDL language, regardless of whether it is a (built-in) *basic* type or a (user-defined) *aggregate* type.

Conceptually, every **TypeCode** contains a "kind" field (which classifies it), and one or more parameters that carry descriptive information appropriate for that particular category of **Type-Code**. For example, if the data type is **long**, its **TypeCode** would contain a "kind" field with the value **tk_long**. No additional parameters are needed to completely describe this particular data type, since **long** is a basic type in the IDL language.

By contrast, if the **TypeCode** describes an IDL **struct**, its "kind" field would contain the value **tk_struct**, and it would possess the following parameters: a string giving the name of the struct, and two additional parameters for each member of the struct: a string giving the member name and another (inner) **TypeCode** representing the member's type. This example illustrates the fact that **TypeCode**s can be nested and arbitrarily complex, as appropriate to express the type of data they describe. Thus, a structure that has N members will have a **TypeCode** of **tk_struct** with 2N+1 parameters (a name and **TypeCode** parameter for each member, plus a name for the struct itself).

A **tk_union TypeCode** representing a union with N members has 3N+2 parameters: the type name of the union, the **switch TypeCode**, and a label value, member name and associated **TypeCode** for each member. (The label values all have the same type as the switch, except that the default member, if present, has a label value of zero **octet.)**

A **tk_enum TypeCode** (which represents an enum) has N+1 parameters: the name of the enum followed by a string for each enumeration identifier. A **tk_string TypeCode** has a single parameter: the maximum string length, as an integer. (A maximum length of zero signifies an unbounded string.)

A **tk_sequence TypeCode** has two parameters: a **TypeCode** for the sequence elements, and the maximum size, as an integer. (Again, zero signifies unbounded.)

A **tk_array TypeCode** has two parameters: a **TypeCode** for the array elements, and the array length, as an integer. (Arrays must be bounded.)

The **tk_objref TypeCode** represents an object reference; its parameter is a repository ID that identifies its interface.

A complete table showing the parameters of all possible **TypeCodes** is given in the *SOMobjects Developer Toolkit Programmers Reference Manual;* see the **TypeCode_kind** function of the Interface Repository Framework.

TypeCodes are not actually "objects" in the formal sense. **TypeCode**s are referred to in the CORBA standard as *pseudo-objects* and described as "opaque". This means that, in reality, **TypeCode**s are special data structures whose precise definition is not fully exposed. Their implementation can vary from one platform to another, but all implementations must exhibit a minimal set of architected behavior. SOM **TypeCode**s support the architected behavior and have additional capability as well (for example, they can be copied and freed).

Although **TypeCode**s are not objects, the programming interfaces that support them adhere to the same conventions used for IDL method invocations in SOM. That is, the first argument is always a **TypeCode** pseudo-object, and the second argument is a pointer to an **Environment** structure. Similarly, the names of the **TypeCode** functions are constructed like SOM's C-language method–invocation macros (all functions that operate on **TypeCode**s are named **Type-Code_**<*function–name*>). Because of this ostensible similarity to an IDL class, the **TypeCode** programming interfaces can be conveniently defined in IDL as shown below.

```
interface TypeCode {

enum TCKind {
    tk_null, tk_void,
    tk_short, tk_long, tk_ushort, tk_ulong,
    tk_float, tk_double, tk_boolean, tk_char,
    tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
    tk_struct, tk_union, tk_enum, tk_string,
    tk_sequence, tk_array,

    // The remaining enumerators are SOM-unique extensions
    // to the CORBA standard.
    //
    tk_pointer, tk_self, tk_foreign
};
```

```
exception Bounds {};
// This exception is returned if an attempt is made
// by the parameter() operation (described below) to
// access more parameters than exist in the receiving
// TypeCode.

boolean equal (in TypeCode tc);
// Compares the argument with the receiver and returns TRUE
// if both TypeCodes are equivalent.  This is NOT a test for
// identity.

TCKind kind ();
// Returns the type of the receiver as a TCKind.

long param_count ();
// Returns the number of parameters that make up the
// receiving TypeCode.

any parameter (in long index) raises (Bounds);
// Returns the indexed parameter from the receiving TypeCode.
// Parameters are indexed from 0 to param_count()-1.

//
//   The remaining operations are SOM-unique extensions.
//

short alignment ();
// This operation returns the alignment required for an instance
// of the type described by the receiving TypeCode.

TypeCode copy (in TypeCode tc);
// This operation returns a copy of the receiving TypeCode.

void free (in TypeCode tc);
// This operation frees the memory associated with the
// receiving TypeCode.  Subsequently, no further use can be
// made of the receiver, which, in effect, ceases to exist.

void print (in TypeCode tc);
// This operation writes a readable representation of the
// receiving TypeCode to stdout.  Useful for examining
// TypeCodes when debugging.

void setAlignment (in short align);
// This operation overrides the required alignment for an
// instance of the type described by the receiving TypeCode.

long size (in TypeCode tc);
// This operation returns the size of an instance of the
// type represented by the receiving TypeCode.
};
```

A detailed description of the programming interfaces for **TypeCode**s is given in the *SOMobjects Developer Toolkit: Programmers Reference Manual*.

Providing 'alignment' information

In addition to the parameters in the **TypeCode**s that describe each type, a SOM-unique extension to the **TypeCode** functionality allows each **TypeCode** to carry alignment information as a "hidden" parameter. Use the **TypeCode_alignment** function to access the alignment value. The alignment value is a short integer that should evenly divide any memory address where an instance of the type will occur.

If no alignment information is provided in your IDL source files, all **TypeCode**s carry default alignment information. The default alignment for a type is the natural boundary for the type, based on the natural boundary for the basic types of which it may be composed. This information can vary from one hardware platform to another. The **TypeCode** will contain the default alignment information appropriate to the platform where it was defined.

To provide alignment information for the types and instances of types in your IDL source file, use the "align=N" modifier, where N is your specified alignment. Use standard modifier syntax of the SOM Compiler to attach the alignment information to a particular element in the IDL source file. In the following example, `align=1` (that is, unaligned or no alignment) is attached to the struct "abc" and to one particular instance of struct "def" (the instance data item "y").

```
interface i {
    struct abc {
        long a;
        char b;
        long c;
    };
    struct def {
        char l;
        long m;
    };
    void foo ();

    implementation {
    //# instance data
        abc x;
        def y;
        def z;


    //# alignment modifiers
        abc: align=1;
        y: align=1;
    };
};
```

Be aware that assigning the required alignment information to a type does *not* guarantee that instances of that type will actually be aligned as indicated. To ensure that, you must find a way to instruct your compiler to provide the desired alignment. In practice, this can be difficult except in simple cases. Most compilers can be instructed to treat all data as aligned (that is, default alignment) or as unaligned, by using a compile-time option or #pragma. The more important consideration is to make certain that the **TypeCode**s going into the Interface Repository actually reflect the alignment that your compiler provides. This way, when programs (such as the DSOM Framework) need to interpret the layout of data during their execution, they will be able to accurately map your data structures. This happens automatically when using the normal default alignment.

If you wish to use unaligned instance data when implementing a class, place an "unattached" `align=1` modifier in the implementation section. An unattached `align=N` modifier is presumed to pertain to the class's instance data structure, and will by implication be attached to all of the instance data items.

When designing your own public types, be aware that the best practice of all (and the one that offers the best opportunity for language neutrality) is to lay out your types carefully so that it will make no difference whether they are compiled as aligned or unaligned!

Using the 'tk_foreign' TypeCode

TypeCodes can be used to partially describe types that cannot be described in IDL (for example, a FILE type in C, or a specific class type in C++). The SOM-unique extension **tk_foreign** is used for this purpose. A **tk_foreign TypeCode** contains three parameters:

1. The name of the type,
2. An implementation context string, and
3. A length.

The implementation context string can be used to carry an arbitrarily long description that identifies the context where the foreign type can be used and understood. If the length of the type is also known, it can be provided with the length parameter. If the length is not known or is not constant, it should be specified as zero. If the length is not specified, it will default to the size of a pointer. A **tk_foreign TypeCode** can also have alignment information specified, just like any other **TypeCode**.

Using the following steps causes the SOM Compiler to create a foreign **TypeCode** in the Interface Repository:

1. Define the foreign type as a **typedef** SOMFOREIGN in the IDL source file.

2. Use the **#pragma modifier** statement to supply the additional information for the **TypeCode** as modifiers. The implementation context information is supplied using the "impctx" modifier.

3. Compile the IDL file using the **–u** option to place the information in the Interface Repository.

For example:

```
typedef SOMFOREIGN Point;
#pragma modifier Point: impctx="C++ Point class",length=12,align=4;
```

If a foreign type is used to define instance data, structs, unions, attributes, or methods in an IDL source file, it is your responsibility to ensure that the implementation and/or usage bindings contain an appropriate definition of the type that will satisfy your compiler. You can use the **passthru** statement in your IDL file to supply this definition. However, it is *not* recommended that you expose foreign data in attributes, methods, or any of the public types, if this can be avoided, because there is no guarantee that appropriate usage binding information can be provided for all languages. If you know that all users of the class will be using the same implementation language that your class uses, you may be able to disregard this recommendation.

TypeCode constants

TypeCodes are actually available in two forms: In addition to the **TypeCode** information provided by the methods of the Interface Repository, **TypeCode** constants can be generated by the SOM Compiler in your C or C++ usage bindings upon request. A **TypeCode** constant contains the same information found in the corresponding IR **TypeCode**, but has the advantage that it can be used as a literal in a C or C++ program anywhere a normal **TypeCode** would be acceptable.

TypeCode constants have the form **TC_**<*typename*>, where <*typename*> is the name of a type (that is, a typedef, union, struct, or enum) that you have defined in an IDL source file. In addition, all IDL basic types and certain types dictated by the OMG CORBA standard come with pre-defined **TypeCode** constants (such as **TC_long**, **TC_short**, **TC_char**, and so forth). A full list of the pre-defined **TypeCode** constants can be found in the file "somtcnst.h". You must explicitly include this file in your source program to use the pre-defined **TypeCode** constants.

Since the generation of **TypeCode** constants can increase the time required by the SOM Compiler to process your IDL files, you must explicitly request the production of **TypeCode** constants if you need them. To do so, use the "tcconsts" modifier with the **–m** option of the **sc** or **somc** command. For example, the command

```
sc –sh –mtcconsts myclass.idl          (On AIX or Windows)
somc –sh –mtcconsts myclass.idl        (On Windows)
```

will cause the SOM Compiler to generate a "myclass.h" file that contains **TypeCode** constants for the types defined in "myclass.idl".

Using the IDL basic type 'any'

Some Interface Repository methods and **TypeCode** functions return information typed as the IDL basic type **any**. Usually this is done when a wide variety of different types of data may need to be returned through a common interface. The type **any** actually consists of a structure with two fields: a **_type** field and a **_value** field. The **_value** field is a pointer to the actual datum that was returned, while the **_type** field holds a **TypeCode** that describes the datum.

In many cases, the context in which an operation occurs makes the type of the datum apparent. If so, there is no need to examine the **TypeCode** unless it is simply as a consistency check. For example, when accessing the first parameter of a **tk_struct TypeCode**, the type of the result will always be the name of the structure (a string). Because this is known ahead of time, there is no need to examine the returned **TypeCode** in the **any _type** field to verify that it is a **tk_string TypeCode**. You can just rely on the fact that it is a string; or, you can check the **TypeCode** in the **_type** field to verify it, if you so choose.

An IDL **any** type can be used in an interface as a way of bypassing the strong type checking that occurs in languages like ANSI C and C++. Your compiler can only check that the interface returns the **any** structure; it has no way of knowing what type of data will be carried by the **any** during execution of the program. Consequently, in order to write C or C++ code that accesses the contents of the **any** correctly, you must always cast the **_value** field to reflect the actual type of the datum at the time of the access.

Here is an example of a code fragment written in C that illustrates how the casting must be done to extract various values from an **any**:

```
#include <som.h>    /* For "any" & "Environment" typedefs */
#include <somtc.h>  /* For TypeCode_kind prototype        */

any result;
Environment *ev;

printf ("result._value = ");
switch (TypeCode_kind (result._type, ev)) {

    case tk_string:
        printf ("%s\n", *((string *) result._value));
        break;

    case tk_long:
        printf ("%ld\n", *((long *) result._value));
        break;

    case tk_boolean:
        printf ("%d\n", *((boolean *) result._value));
        break;

    case tk_float:
        printf ("%f\n", *((float *) result._value));
        break;

    case tk_double:
        printf ("%f\n", *((double *) result._value));
        break;

    default:
        printf ("something else!\n");
}
```

Note: Of course, an **any** has no restriction, per se, on the type of datum that it can carry. Frequently, however, methods that return an **any** or that accept an **any** as an argument do place semantic restrictions on the actual type of data they can accept or return. Always consult the reference page for a method that uses an **any** to determine whether it limits the range of types that may be acceptable.

Chapter 8. The Metaclass Framework

In SOM, classes are objects. Metaclasses are classes and thus are objects, too. Figure 1 depicts the relationship of these sets of objects. Included are the three primitive class objects of the SOM run time: **SOMClass**, **SOMObject**, and **SOMClassMgr**.
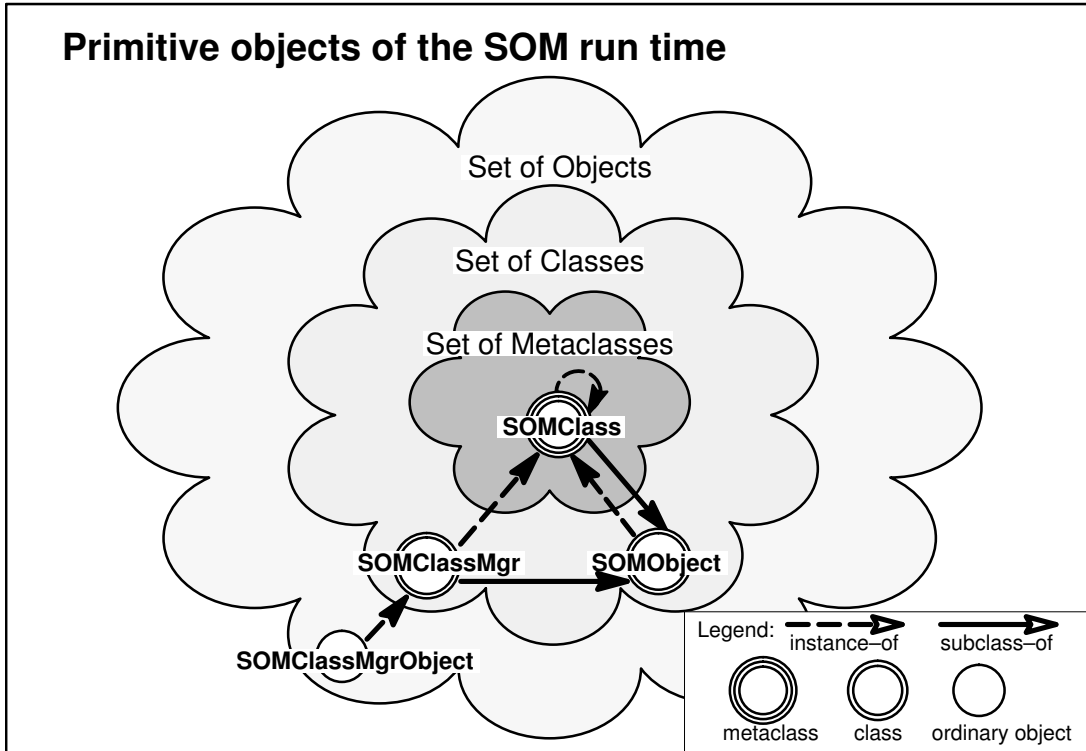
Figure 1.   The primitive objects of the SOM run time.

The important point to observe here is that any class that is a *subclass* of **SOMClass** is a *metaclass*. This chapter describes metaclasses that are available in SOMobjects Toolkit. There are two kinds of metaclasses:

> *Framework metaclasses* — metaclasses for building new metaclasses, and
> *Utility metaclasses* — metaclasses to help you write applications.

Briefly, the SOMobjects Toolkit provides the following metaclasses of each category for use by programmers:

- *Framework metaclasses:*

  SOMMBeforeAfter   — Used to create a metaclass that has "before" and "after" methods for *all* methods (inherited or introduced) invoked on instances of its classes.

- *Utility metaclasses:*

  SOMMSingleInstance   — Used to create a class that may have at most one instance.

  SOMMTraced   — Provides tracing for every invocation of all methods on instances of its classes.

The diagram in Figure 2 depicts the relationship of these metaclasses to **SOMClass** (for completeness, the figure includes the metaclasses that are derived). The following sections describe each metaclass more fully. The ellipses indicate that there are additional metaclasses being used that are not part of the public interface.
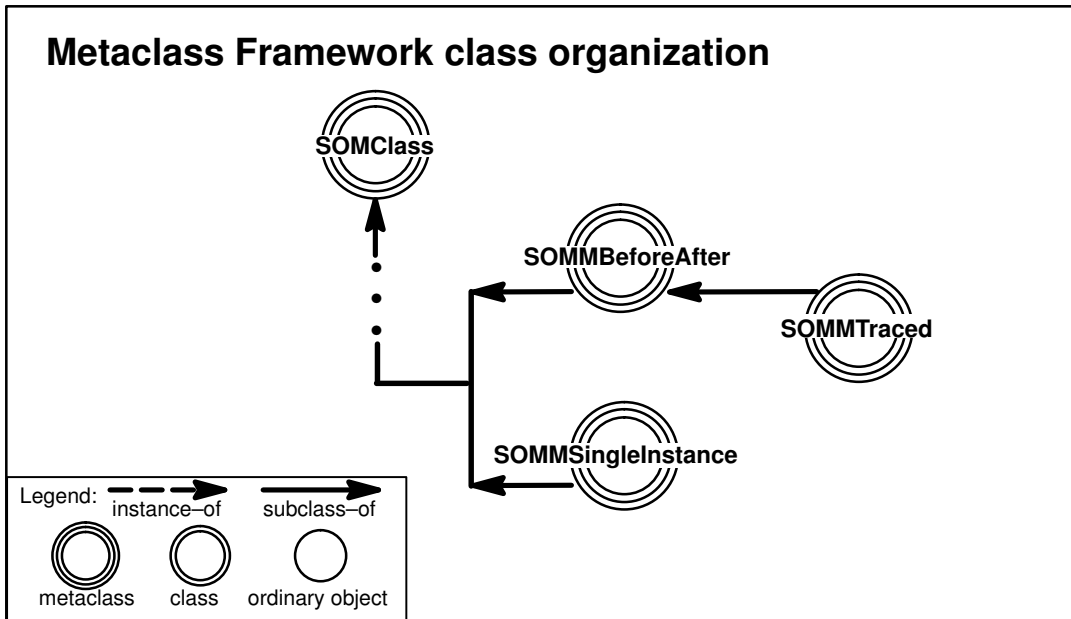
Figure 2.   Class organization of the Metaclass Framework.

A note about metaclass programming

SOM metaclasses are carefully constructed so that they compose (see Section 8.1 below). If you need to create a metaclass, you can introduce new class methods, and new class variables, but you should not override any of the methods introduced by **SOMClass**. If you need more than this, request access to the experimental Cooperative metaclass used to implement the Metaclass Framework metaclasses described in this chapter.

8.1 Framework Metaclasses for "Before/After" Behavior

The 'SOMMBeforeAfter' metaclass

SOMMBeforeAfter is a metaclass that allows the user to create a class for which a particular method is invoked *before* each invocation of every method, and for which a second method is invoked *after* each invocation. **SOMMBeforeAfter** defines two methods: **sommBeforeMethod** and **sommAfterMethod**. These two methods are intended to be overridden in the child of **SOMMBeforeAfter** to define the particular "before" and "after" methods needed for the client application.

As further depicted in Figure 3, the "Barking" metaclass overrides the **sommBeforeMethod** and **sommAfterMethod** with a method that emits one bark when invoked. Thus, one can create the "BarkingDog" class, whose instances (such as "Lassie") bark twice when "disturbed" by a method invocation.
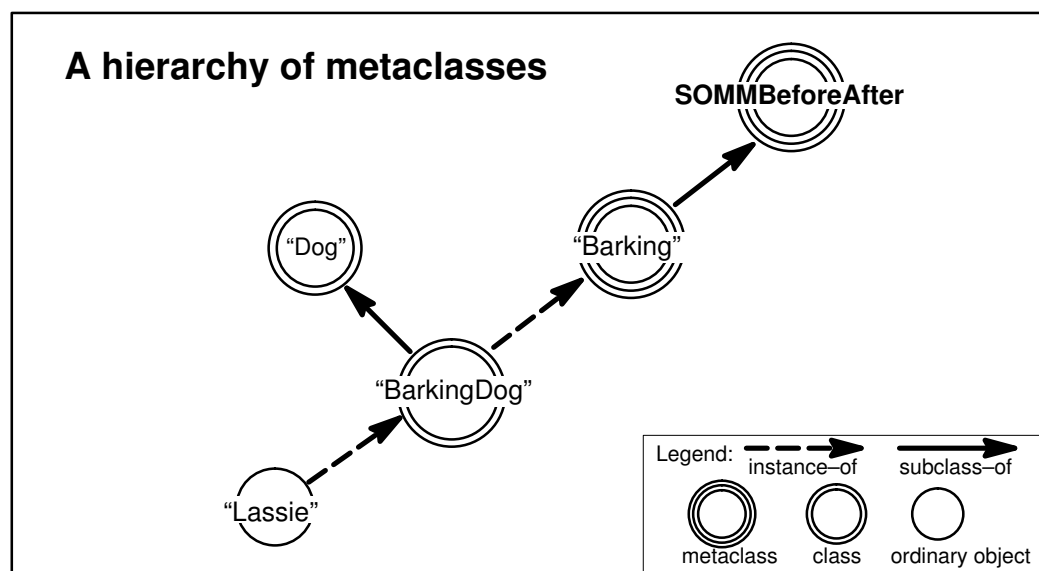
Figure 3.   A hierarchy of metaclasses

The **SOMMBeforeAfter** metaclass is designed to be subclassed; a subclass (or child) of **SOMMBeforeAfter** is also a metaclass. The subclass overrides **sommBeforeMethod** or **sommAfterMethod** or both. These (redefined) methods are invoked before and after any method supported by instances of the subclass (these methods are called *primary* methods). That is, they are invoked before and after methods invoked on the ordinary objects that are instances of the class objects that are instances of the subclass of **SOMMBeforeAfter**.

The **sommBeforeMethod** returns a **boolean** value. This allows the "before" method to control whether the "after" method and the primary method get invoked. If **sommBeforeMethod** returns TRUE, normal processing occurs. If FALSE is returned, neither the primary method nor the corresponding **sommAfterMethod** is invoked. In addition, no more deeply nested before/after methods are invoked (see "Composition of before/after metaclasses" below). This facility can be used, for example, to allow a before/after metaclass to provide secure access to an object. The implication of this convention is that, if **sommBeforeMethod** is going to return FALSE, it must do any post-processing that might otherwise be done in the "after" method.

Caution: **somInit** and **somFree** are among the methods that get before/after behavior. This implies the following two obligations are imposed on the programmer of a **SOMMBeforeAfter** class. First, the implementation must guard against **sommBeforeMethod** being called before **somInit** has executed, and the object is not yet fully initialized. Second, the implementation must guard against **sommAfterMethod** being called after **somFree**, at which time the object no longer exists (see the example "C implementation for 'Barking' metaclass" below).

The following example shows the IDL needed to create a Barking metaclass. Just run the appropriate emitter to get an implementation binding, and then provide the appropriate "before" behavior and "after" behavior.

SOM IDL for 'Barking' metaclass

```
#ifndef Barking_idl
#define Barking_idl

#include <sombacls.idl>
interface  Barking : SOMMBeforeAfter
{
#ifdef __SOMIDL__
implementation
{
  //# Class Modifiers
  filestem = barking;
  callstyle = idl;

  //# Method Modifiers
  sommBeforeMethod : override;
  sommAfterMethod : override;
};
#endif /* __SOMIDL__ */
};
#endif  /* Barking_idl */
```

The next example shows an implementation of the Barking metaclass in which *no* barking occurs when **somFree** is invoked.

C implementation for 'Barking' metaclass

```
#define Barking_Class_Source
#include <barking.ih>

static char *somMN_somFree = "somFree";
static somId somId_somFree = &somMN_somFree;

SOM_Scope boolean  SOMLINK sommBeforeMethod(Barking somSelf,
                                            Environment *ev,
                                            SOMObject object,
                                            somId methodId,
                                            va_list ap)
{
    if ( !somCompareIds( methodId, somId_somFree )
       printf( "WOOF" );
}


SOM_Scope void  SOMLINK sommAfterMethod(Barking somSelf,
                                        Environment *ev,
                                        SOMObject object,
                                        somId methodId,
                                        somId descriptor,
                                        somToken returnedvalue,
                                        va_list ap)
{
    if ( !somCompareIds( methodId, somId_somFree )
       printf( "WOOF" );
}
```

Composition of before/after metaclasses

Consider Figure 4 in which there are two before/after metaclasses — "Barking" (as before) and "Fierce", which has a **sommBeforeMethod** and **sommAfterMethod** that both growl (that is, both make a "grrrr" sound when executed). The preceding discussion demonstrated how to create a "FierceDog" or a "BarkingDog", but has not yet addressed the question of how to compose these properties of fierce and barking. *Composability* means having the ability to easily create either a "FierceBarkingDog" that goes "grrr woof woof grrr" when it responds to a method call or a "BarkingFierceDog" that goes "woof grrr grrr woof" when it responds to a method call.

Example for composition of before/after metaclasses
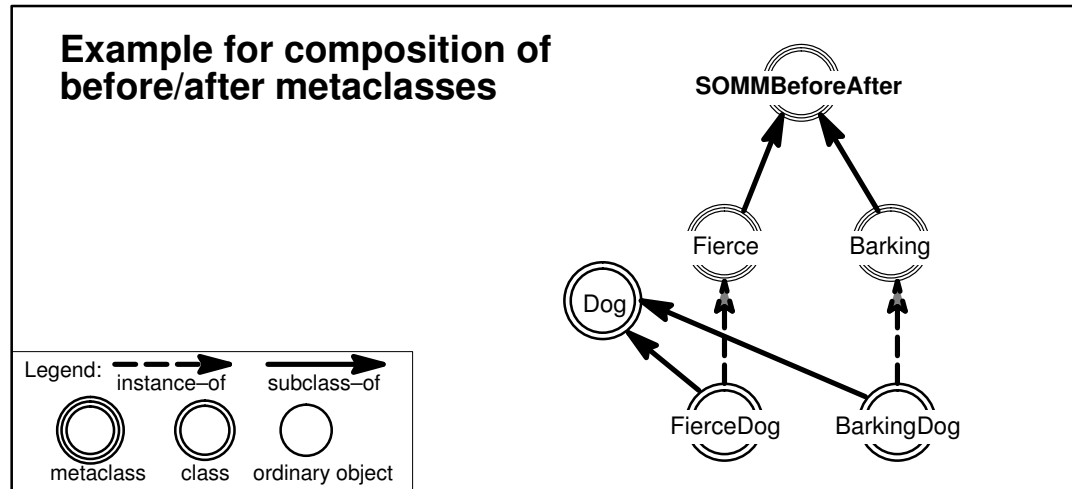
Figure 4.   Example for composition of before/after metaclasses.

There are several ways to express such compositions. Figure 5 depicts SOM IDL fragments for three techniques in which composition can be indicated by a programmer. These are denoted as Technique 1, Technique 2, and Technique 3, each of which creates a "FierceBarkingDog" class, named "FB-1", "FB-2", and "FB-3", respectively, as follows:

- In Technique 1, a new metaclass ("FierceBarking") is created with both the "Fierce" and "Barking" metaclasses as parents. An instance of this new metaclass (that is, "FB-1") is a "FierceBarkingDog" (assuming "Dog" is a parent).

- In Technique 2, a new class is created which has parents that are instances of "Fierce" and "Barking" respectively. That is, "FB-2" is a "FierceBarkingDog" also (assuming "FierceDog" and "BarkingDog" do not further specialize "Dog").

- In Technique 3, "FB-3", which also is a "FierceBarkingDog", is created by declaring that its parent is a "BarkingDog" and that its explicit (syntactically declared) metaclass is "Fierce".

Technique 1	Technique 2	Technique 3

interface FB-1 : Dog
{
  ...
  implementation
  {
    metaclass = FierceBarking;
    ...
  };
};
	interface FB-2 : FierceDog,
	                   BarkingDog
	  {
	    ...
	  implementation
	  {
	    ...
	  };
	};
		interface FB-3 : BarkingDog
		{
		  ...
		  implementation
		  {
		    metaclass = Fierce;
		    ...
		  };
		};
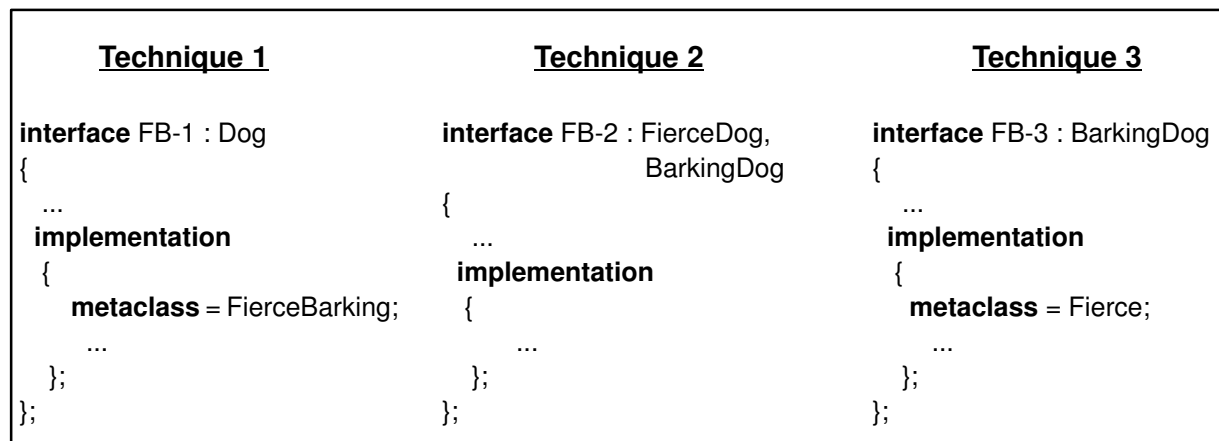
Figure 5.   Three techniques for composing before/after metaclasses.

Figure 6 combines the diagrams for the techniques in Figure 5 and shows the actual class relationships. Note that the explicit metaclass in the SOM IDL of "FB-1" is its derived class, "FierceBarking". The derived metaclass of "FB-2" is also "FierceBarking".   Lastly, the derived metaclass of "FB-3" is not the metaclass explicitly specified in the SOM IDL; rather, it too is "FierceBarking."
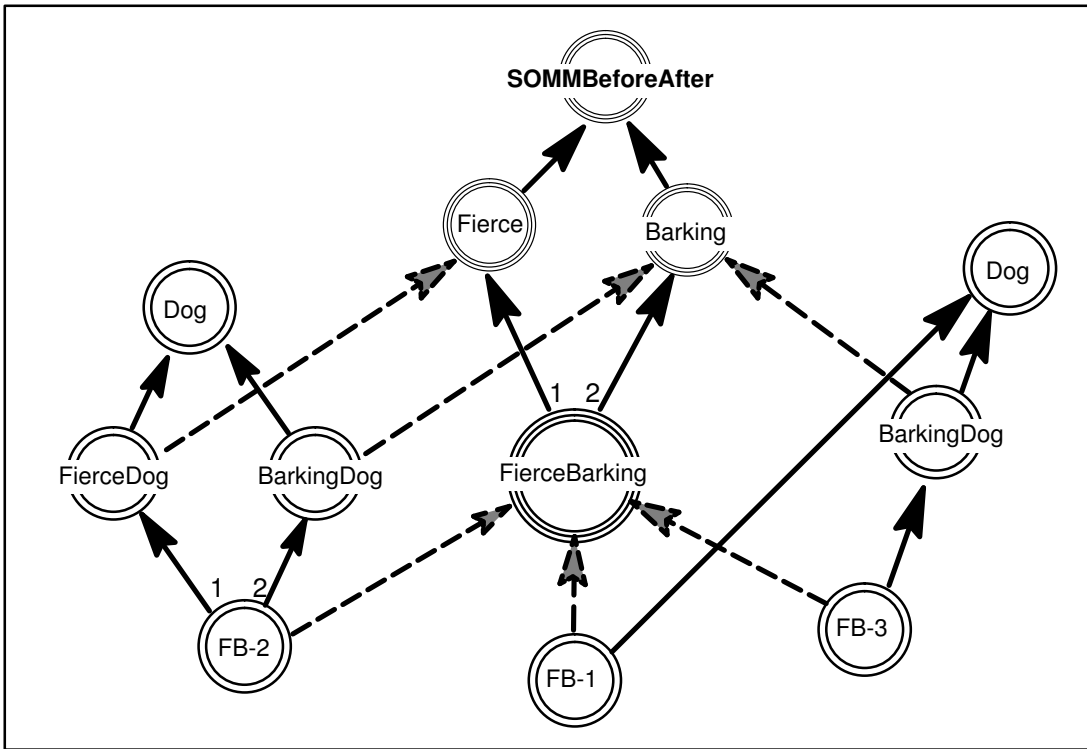
Figure 6. The combined diagram depicting the three techniques for creating a "FierceBarkingDog".

Notes and advantages of 'before/after' usage

Notes on the dispatching of before/after methods:

• A before (after) method is invoked just once per primary method invocation.

• The dispatching of before/after methods is thread-safe.

• The dispatching of before/after methods is fast. The time overhead for dispatching a primary method is on the order of N times the time to invoke a before/after method as a procedure, where N is the total number of before/after methods to be applied.

In conclusion, consider an example that clearly demonstrates the power of the composition of before/after metaclasses. Suppose you are creating a class library that will have n classes. Further suppose there are p properties that must be included in all combinations for all classes. Potentially, the library must have $n2^p$ classes. Let us hypothesize that (fortunately) all these properties can be captured by before/after metaclasses. In this case, the size of the library is $n+p$.

The user of such a library need only produce those combinations necessary for a given application. In addition, note that there is none of the usual programming. Given the IDL for a combination of before/after metaclasses, the SOM compiler generates the implementation of the combination (in either C or C++) with no further manual intervention.

8.2 The 'SOMMSingleInstance' Metaclass

Sometimes it is necessary to define a class for which only one instance can be created. This is easily accomplished with the **SOMMSingleInstance** metaclass. Suppose the class "Collie" is an instance of **SOMMSingleInstance**. The first call to **CollieNew** creates the one possible instance of "Collie"; hence, subsequent calls to **CollieNew** return the first (and only) instance.

Any class whose metaclass is **SOMMSingleInstance** gets this requisite behavior; nothing further needs to be done. The first instance created is always returned by the *<className>***New** macro.

Alternatively, the *method* **sommGetSingleInstance** does the same thing as the *<class-Name>***New** macro. This method invoked on a class object (for example, "Collie") is useful because the call site explicitly shows that something special is occurring and that a new object is not necessarily being created. For this reason, one might prefer the second form of creating a single-instance object to the first.

Instances of **SOMMSingleInstance** keep a count of the number of times **somNew** and **somm-GetSingleInstance** are invoked. Each invocation of **somFree** decrements this count. An invocation of **somFree** does not actually free the single instance until the count reaches zero.

SOMMSingleInstance overrides **somRenew, somRenewNoInit, somRenewNoInitNoZero,** and **somRenewNoZero** so that a proxy is created in the space indicated in the **somRenew*** call. This proxy redispatches all methods to the single instance, which is always allocated in heap storage. Note that all of these methods (**somRenew***) increment the reference count; therefore, **somFree** should be called on these objects, too. In this case, **somFree** decrements the reference and frees the single instance (and, of course, takes no action with respect to the storage indicated in the original **somRenew*** call).

If a class is an instance of **SOMMSingleInstance**, all of its subclasses are also instances of **SOMMSingleInstance**. Be aware that this also means that each *subclass* is allowed to have only a single instance. (This may seem obvious. However, it is a common mistake to create a framework class that must have a single instance, while at the same time expecting users of the framework to subclass the single instance class. The result is that two single-instance objects are created: one for the framework class and one for the subclass. One technique that can mitigate this scenario is based on the use of **somSubstituteClass**. In this case, the creator of the subclass must substitute the subclass for the framework class — before the instance of the framework class is created.)

8.3 The 'SOMMTraced' Metaclass

SOMMTraced is a metaclass that facilitates tracing of method invocations. If class "Collie" is an instance of **SOMMTraced** (if **SOMMTraced** is the metaclass of "Collie"), any method invoked on an instance of "Collie" is traced. That is, before the method begins execution, a message prints (to standard output) giving the actual parameters. Then, after the method completes execution, a second message prints giving the returned value. This behavior is attained merely by being an instance of the **SOMMTraced** metaclass.

If the class being traced is contained in the Interface Repository, actual parameters are printed as part of the trace. If the class is not contained in the Interface Repository, an ellipsis is printed.

To be more concrete, consider Figure 7. Here, the class "Collie" is a child of "Dog" and is an instance of **SOMMTraced**. Because **SOMMTraced** is the metaclass of "Collie," any method invoked on "Lassie" (an instance of "Collie") is traced.
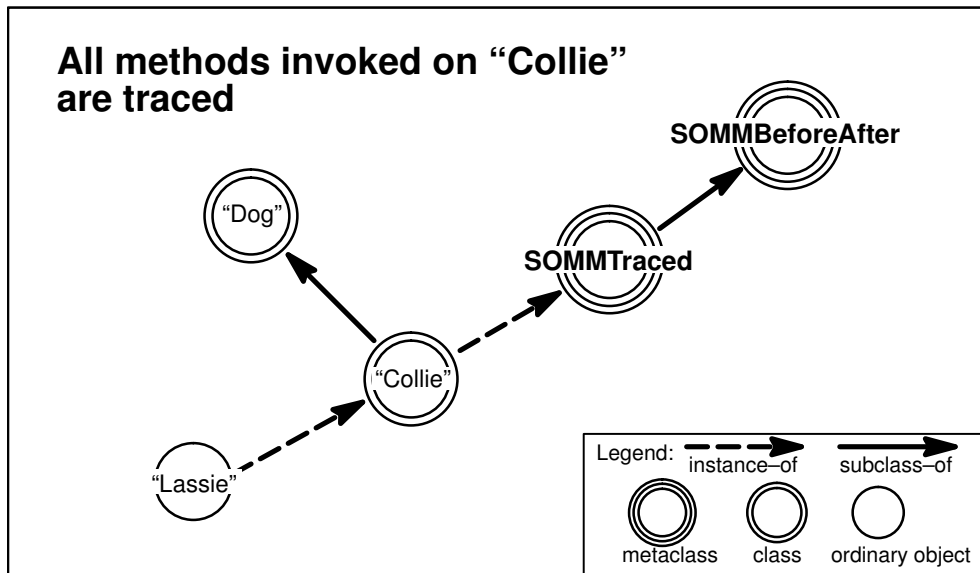
Figure 7.   All methods (inherited or introduced) that are invoked on "Collie" are traced.

It is easy to use **SOMMTraced**: Just make a class an instance of **SOMMTraced** in order to get tracing.

There is one more step for using **SOMMTraced**: Nothing prints unless the environment variable SOMM_TRACED is set. If it is set to the empty string, *all* traced classes print. If SOMM_TRACED is not the empty string, it should be set to the list of <u>names of classes</u> that should be traced. For example, the following command turns on printing of the trace for "Collie", but not for any other traced class:

```
export  SOMM_TRACED=Collie       (on AIX)
SET     SOMM_TRACED=Collie       (on OS/2 or Windows)
```

The example below shows the IDL needed to create a traced dog class: Just run the appropriate emitter to get an implementation binding.

SOM IDL for 'TracedDog' class

```
#include "dog.idl"
#include <somtrcls.idl>
interface TracedDog : Dog
{
#ifdef __SOMIDL__
implementation
{
  //# Class Modifiers
  filestem = trdog;
  metaclass = SOMMTraced;
};
#endif /* __SOMIDL__ */
};
```

8.4 Error Codes

It is possible to receive the following messages from the Metaclass Framework while an application is running.

60001 An attempt was made to construct a class with **SOMMSingleInstance** as a metaclass constraint. (This may occur indirectly because of the construction of a derived metaclass). The initialization of the class failed because **somInitMI-Class** defined by **SOMMSingleInstance** is in conflict with another metaclass that has overridden **somNew**. That is, some other metaclass has already claimed the right to return the value for **somNew**.

60002 An attempt was made to construct a class with **SOMMSingleInstance** as a metaclass constraint. (This may occur indirectly because of the construction of a derived metaclass). The initialization of the class failed because **somInitMI-Class** defined by **SOMMSingleInstance** is in conflict with another metaclass that has overridden **somFree**. That is, some other metaclass has already claimed this right to override **somFree**.

60004 An invocation of **somrRepInit** was made with a logging type other than '**o**' or '**v**'.

60005 The **sommBeforeMethod** or the **sommAfterMethod** was invoked on a **SOMRReplicableObject** whose logging type is other than '**o**' or '**v**'. This error cannot occur normally. The likely cause is that some method invoked on another object has overwritten this object's memory.

60006 A Before/After Metaclass must override both **sommBeforeMethod** and **sommAfterMethod**. This message indicates an attempt to create a Before/After Metaclass where only one of the above methods is overridden.

Chapter 9. The Event Management Framework

The **Event Management Framework** is a central facility for registering all events of an application. Such a registration facilitates grouping of various application events and waiting on multiple events in a single event-processing loop. This facility is used by the DSOM Framework to wait on events of interest. The Event Management Framework must also be used by any interactive application that contains DSOM or replicated objects.

9.1 Event Management Basics

The Event Management Framework consists of an Event Manager (EMan) class, a Registration Data class and several Event classes. It provides a way to organize various application events into groups and to process all events in a single event-processing loop. The need for this kind of facility is seen very clearly in interactive applications that also need to process some background events (say, messages arriving from a remote process). Such applications must maintain contact with the user while responding to events coming from other sources.

One solution in a multi-threaded environment is to have a different thread service each different source of events. For a single-threaded environment it should be possible to recognize and process all events of interest in a single main loop. EMan offers precisely this capability. EMan can be useful even when multiple threads are available, because of its simple programming model. It avoids contention for common data objects between EMan event processing and other main-loop processing activity.

Model of EMan usage

The programming model of EMan is similar to that of many GUI toolkits. The main program initializes EMan and then registers interest in various types of events. The main program ends by calling a non-returning function of EMan that waits for events and dispatches them as and when they occur. In short, the model includes steps that:

1. Initialize the Event Manager,
2. Register with EMan for all events of interest, and
3. Hand over control to EMan to loop forever and to dispatch events.

The Event Manager is a SOM object and is an instance of the **SOMEEMan** class. Since any application requires only one instance of this object, the **SOMEEMan** class is an instance of the **SOMMSingleInstance** class. Creation and initialization of the Event Manager is accomplished by a function call to **SOMEEmanNew**.

Currently, EMan supports the four kinds of events described in the following topic. An application can register or unregister for events in a callback routine (explained below) even after control has been turned over to EMan.

Note: Under Windows, a single event processing loop must necessarily incorporate the program's message processing loop. See "Processing Events" below for a description of how this is accomplished by EMan.

Event types

Event types are categorized as follows:

- **Timer events**

  These can be either one-time timers or interval timers.

- **Sink events** (sockets, file descriptors, and message queues)

  On AIX, this includes file descriptors for input/output files, sockets, pipes, and message queues. On OS/2 and Windows, only TCP/IP sockets are supported.

  Note: On OS/2 and Windows, the Sockets classes for NetBIOS (**NBSockets**) and Novell IPX/SPX (**IPXSockets**) are primarily intended for use by DSOM and the Replication Framework, not for general application programming. (The Replication Framework is available as part of the full-capability SOMobjects Developer Toolkit.)

- **Client events** (any event that the application wants to queue with EMan)

  These events are defined, created, processed, and destroyed by the application. EMan simply acts as a place to queue these events for processing. EMan dispatches these client events whenever it sees them. Typically, this happens immediately after the event is queued.

- **Work procedure events** (procedures that can be called when there is no other event)

  These are typically background procedures that the application intends to execute when there are spare processor cycles. When there are no other events to process, EMan calls all registered work procedures.

The Event Management Framework is extendible (that is, other event types can be added to it) through subclassing. The event types currently supported by EMan are at a sufficiently low level so as to enable building other higher level application events on top of them. For example, you can build an X-event handler by simply registering the file descriptor for the X connection with EMan and getting notified when any X-event occurs.

Registration

This topic illustrates how to register for an event type.

Callbacks

The programmer decides what processing needs to be done when an event occurs and then places the appropriate code either in a procedure or in a method of an object. This procedure or method is called a *callback*. (The callback is provided to EMan at the time of registration and is called by EMan when a registered event occurs.) The signature of a callback is fixed by the framework and must have one of the following three signatures:

```
void SOMLINK EMRegProc(SOMEEvent, void *);
void SOMLINK EMMethodProc(SOMObject, SOMEEvent, void *);
void SOMLINK EMMethodProcEv(SOMObject, Environment *Ev,
                            SOMEEvent, void *);
/* On OS/2, they all use "system" linkage  */
/* On Windows, the SOMLINK keyword is NOT included if the
 * application is intended to support multiple instances. */
```

The three specified prototypes correspond to a simple callback procedure, a callback method using OIDL call style, and a callback method using IDL call style. The parameter type **SOMEEvent** refers to an event object passed by EMan to the callback. Event objects are described below.
Note: When the callbacks are methods, EMan calls these methods using **Name–lookup Resolution** (see Chapter 4, Section 4.3 on Method Resolution). One of the implications is that at the time of registration EMan queries the target object's class object to provide a method pointer for the method name supplied to it. Eman uses this pointer for making event callbacks.

Event classes

All event objects are instances of either the **SOMEEvent** class or a subclass of it. The hierarchy of event classes is as follows:

SOMObject ——— **SOMEEvent** ——— |——— **SOMETimerEvent**
                                    |——— **SOMEClientEvent**
                                    |——— **SOMESinkEvent**
                                    |——— **SOMEWorkProcEvent**
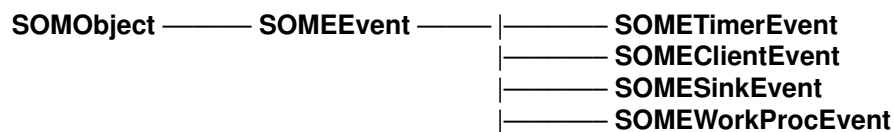
When called by EMan, a callback expects the appropriate event instance as a parameter. For example, a callback registered for a timer event expects a **SOMETimerEvent** instance from EMan.

EMan parameters

Several method calls in the Event Management Framework make use of bit masks and constants as parameters (for example, **EMSinkEvent** or **EMInputReadMask**). These methods are defined in the include file "eventmsk.h"*.* When a user plans to extend the Event Management Framework, care must be taken to avoid name and value collisions with the definitions in "eventmsk.h". For convenience, the contents of the "eventmsk.h" file are shown below.

```
#ifndef H_EVENTMASKDEF
#define H_EVENTMASKDEF

/* Event Types */
#define EMTimerEvent                    54
#define EMSignalEvent                   55
#define EMSinkEvent                     56

#define EMWorkProcEvent                 57

#define EMClientEvent                   58

#define EMMsgQEvent                     59

/* Sink input/output condition mask */

#define EMInputReadMask            (1L<<0)
#define EMInputWriteMask           (1L<<1)
#define EMInputExceptMask          (1L<<2)

/* Process Event mask */

#define EMProcessTimerEvent        (1L<<0)
#define EMProcessSinkEvent         (1L<<1)
#define EMProcessWorkProcEvent     (1L<<2)
#define EMProcessClientEvent       (1L<<3)
#define EMProcessAllEvents         (1L<<6)

#endif   /* H_EVENTMASKDEF */
```

Registering for events

In addition to the event classes, the Event Management Framework uses a registration data class (**SOMEEMRegisterData**) to capture all event-related registration information. The procedure for registering interest in an event is as follows:

1. Create an instance of the **SOMEEMRegisterData** class (this will be referred to as a "RegData" object).

2. Set the event type of "RegData."

3. Set the various fields of "RegData" to supply information about the particular event for which an interest is being registered.

4. Call the registration method of EMan, using "RegData" and the callback method information as parameters. The callback information varies, depending upon whether it is a simple procedure, a method called using OIDL call style, or a method called using IDL call style.

The following code segment illustrates how to register input interest in a socket "sock" and provide a callback procedure "ReadMsg".

```
data = SOMEEMRegisterDataNew( );       /* create a RegData object */
_someClearRegData(data, Ev);
_someSetRegDataEventMask(data,Ev,EMSinkEvent,NULL); /* Event type */
_someSetRegDataSink(data, Ev, sock);     /* provide the socket id */
_someSetRegDataSinkMask(data,Ev, EMInputReadMask );
                                          /*input interest */
regId = _someRegisterProc(some_gEMan, Ev, data,
         (EMRegProc *) ReadMsg, "UserData" );
/* some_gEMan points to EMan. The last parameter "userData" is any
 * data the user wants to be passed to the callback procedure as a
 * second parameter */
```

Unregistering for events

One can unregister interest in a given event type at any time. To unregister, you must provide the registration id returned by EMan at the time of registration. Unregistering a non-existent event (such as, an invalid registration id) is a no-op. The following example unregisters the socket registered above:

```
_someUnRegister(some_gEMan, Ev, regId);
```

An example callback procedure

The following code segment illustrates how to write a callback procedure:

```
void SOMLINK ReadMsg( SOMEEvent  event, void  *targetData )
{
int sock;
    printf( "Data = %s\n", targetData );
    switch( _somevGetEventType( event )) {
    case  EMSinkEvent:
        printf("callback: Perceived Sink Event\n");
        sock = _somevGetEventSink(event);
        /* code to read the message off the socket */
        break;
    default: printf("Unknown Event type in socket callback\n");
    }
}
    /* On OS/2, "system" linkage is also required.  */
    /* On Windows, callbacks do not use the SOMLINK keyword if
     * the application is intended to support multiple instances. */
```

Generating client events

While the other events are caused by the operating system (for example, Timer), by I/O devices, or by external processes, client events are caused by the application itself. The application creates these events and enqueues them with EMan. When client events are dispatched, they are processed in a callback routine just like any other event. The following code segment illustrates how to create and enqueue client events.

```
clientEvent1 = SOMEClientEventNew();  /* create a client event */
_somevSetEventClientType( clientEvent1, Ev, "MyClientType" );
_somevSetEventClientData( clientEvent1, Ev,
                                   "I can give any data here");
/* assuming that "MyClientType" is already registered with EMan */
/* enqueue the above event with EMan */
_someQueueEvent(some_gEMan, Ev, clientEvent1);
```

Examples of using other events

The sample program shipped with the Event Management Framework illustrates the tasks listed below. (Due to its large size, the source code is not included here.)

- Registering and unregistering for Timer events.
- Registering and unregistering for Workproc events.
- Registering an AIX Message Queue, sending messages on it, and unregistering the Message Queue.
- Registering a stream socket that listens to incoming connection requests. Also, sockets connecting, accepting a connection, and sending/receiving messages through EMan.
- Registering a file descriptor on AIX and reading one line of the file at a time in a callback.

Processing events

After all registrations are finished, an application typically turns over control to EMan and is completely event driven thereafter. Typically, an application main program ends with the following call to EMan:

```
_someProcessEvents(some_gEMan, Ev);
```

An equivalent way to process events is to write a main loop and call **someProcessEvent** from inside the main loop, as indicated:

```
while (1) { /* do forever */
       _someProcessEvent(some_gEMan, Ev, EMProcessTimerEvent  |
                                         EMProcessSinkEvent    |
                                         EMProcessClientEvent |
                                         EMProcessWorkProcEvent );
       /*** Do other main loop work, as needed. ***/
}
```

The second way allows more precise control over what type of events to process in each call. The example above enables all four types to be processed. The required subset is formed by logically OR'ing the appropriate bit constants (these are defined in "eventmsk.h)". Another difference is that the second way is a non-blocking call to EMan. That is, if there are no events to process, control returns to the main loop immediately, whereas **someProcessEvents** is a non-returning blocking call. For most applications, the first way of calling EMan is better, since it does not waste processor cycles when there are no events to process.

For Windows:
The _**someProcessEvents** method incorporates a standard window-message-processing loop into its event processing loop. In order to allow timer events to operate, messages are retrieved using **PeekMessage** rather than **GetMessage**. If you do not wish to use timer events, or if you want to perform other processing in your message loop (such as, translating accelerators), then you should use _**someProcessEvent**, as follows:

```
while (GetMessage(&msg, NULL, NULL, NULL))
{
   if (!TranslateAccelerator(hwndMain, haccel, &msg))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }

   _someProcessEvent(some_gEMan, Ev, EMProcessTimerEvent  |
                                     EMProcessSinkEvent    |
                                     EMProcessClientEvent |
                                     EMProcessWorkProcEvent );
    ...
}
```

Interactive applications

Note: This topic does *not* apply to SOMobjects For Windows, as user input is processed in the message loop that is incorporated into EMan.

Interactive applications need special attention when coupled with EMan. Once control is turned over to EMan by calling **someProcessEvents**, a single-threaded application (for example, on AIX) has no way of responding to keyboard input. The user must register interest in "stdin" with EMan and provide a callback function that handles keyboard input. In a multi-threaded environment (for example, OS/2), this problem can be solved by spawning a thread to execute **someProcessEvents** and another to handle keyboard input. (These two options are illustrated in the sample program shipped with the Event Management Framework.)

9.2 Event Manager Advanced Topics

Threads and thread safety

As indicated earlier, on OS/2, interactive programs call **someProcessEvents** in one thread and process keyboard input in a separate thread. (This recommended usage is illustrated in the sample program). The event manager object (EMan) is thread safe in the sense that concurrent method invocations on EMan are serialized. Even when **someProcessEvents** is invoked in a thread and other methods of EMan are invoked from other threads, EMan still preserves its data integrity. However, when Eman dispatches an event, a callback can call methods on the same data objects as the other interactive thread(s). The user must protect such data objects using appropriate concurrency control techniques (for example by using semaphores).

One must also be aware of some deadlock possibilities. Consider the following situation. EMan code holds some SOMobjects Toolkit semaphores while it is running (for example, while in **someProcessEvents**). A user-defined object protects its data by requiring its methods to acquire and release a sempahore on the object. If a separate thread running in this object were to call an operation that requires a SOMobjects Toolkit semaphore (which is currently held by EMan) and if concurrently EMan dispatches an event whose callback invokes a method of this object, a deadlock occurs. Two possibilities exist to cope with such a situation: One is to acquire all needed semaphores ahead of time, and the other is to abort the operation when you fail to obtain a semaphore. To achieve mutual exclusion with EMan, you can call the methods **someGetEManSem** and **someReleaseEmanSem**. These methods acquire and release the SOMobject Developer Toolkit semaphores that EMan uses.

Writing an X or MOTIF application

Although the Event Manager does not recognize X events, an X or MOTIF application can be integrated with EMan as follows. First, the necessary initialization of X or MOTIF should be performed. Next, using the Xlib macro "ConnectionNumber" or the "XConnectionNumber" function, you can obtain the file descriptor of the X connection. This file descriptor can be registered with EMan as a sink. It can be registered for both input events and exception events. When there is any activity on this X file descriptor, the developer-provided callback is invoked. The callback can receive the X-event, analyze it, and do further dispatching.

Extending EMan

The current event manager can be extended without having access to the source code. The use of EMan in an X or MOTIF application mentioned above is just one such example. Several other extensions are possible. For example, new event types can be defined by subclassing either directly from **SOMEEvent** class or from any of its subclasses in the framework. There are three main problems to solve in adding a new event type:

- How to register a new event type with EMan?

- How to make EMan recognize the occurrence of the new event?

- How to make EMan create and send the new event object (a subclass of **SOMEEvent**) to the callback when the event is dispatched?

Because the registration information is supplied with appropriate "set" methods of a RegData object, the RegData object should be extended to include additional methods. This can be achieved by subclassing from **SOMEEMRegisterData** and building a new registration data class that has methods to "set" and "get" additional fields of information that are needed to describe the new event types fully. To handle registrations with instances of new registration data subclass, we must also subclass from **SOMEEMan** and override the **someRegister** and the **someUnRegister** methods. These methods should handle the information in the new fields introduced by the new registration data class and call parent methods to handle the rest.

Making EMan recognize the occurrence of the new event is primarily limited by the primitive events EMan can wait on. Thus the new event would have to be wrapped in a primitive event that EMan can recognize. For example, to wait on a message queue on OS/2 concurrently with other EMan events, a separate thread can be made to wait on the message queue and to enqueue a client event with EMan when a message arrives on this message queue. We can thus bring multiple event sources into the single EMan main loop.

The third problem of creating new event objects unknown to EMan can be easily done by applying the previous technique of wrapping the new event in terms of a known event. In a callback routine of the known event, we can create and dispatch the new event unknown to EMan. Of course, this does introduce an intermediate callback routine which would not be needed if EMan directly understood the new event type.

A general way of extending EMan is to look for newly defined event types by overriding **someProcessEvent** and **someProcessEvents** in a subclass of EMan.

Using EMan from C++

The Event Management framework can be used from C++ just like any other framework in the SOMobjects Toolkit. You must ensure that the C++ usage bindings (that is, the .xh files) are available for the Event Management Framework classes. These .xh files are generated by the SOM Compiler in the SOMobjects Toolkit when the **-s** option includes an xh emitter.

Using EMan from other languages

The event manager and the other classes can be used from other languages, provided usage bindings are available for them. These usage bindings are produced from .idl files of the framework classes by the appropriate language emitter.

Tips on using EMan

The following are some do's and don'ts for EMan:

- Eman callback procedures or methods must return quickly. You cannot wait for long periods of time to return from the callbacks. If such long delays occur, then the application may not notice some subsequent events in time to process them meaningfully (for example, a timer event may not be noticed until long after it occurred).

- It follows from the previous tip that you should not do independent "select" system calls on file descriptors while inside a callback. (This applies to sockets and message queues, as well.) In general, a callback should not do any blocking of system calls. If an application must do this, then it must be done with a small timeout value.

- Since EMan callbacks must return quickly, no callback should wait on a semaphore indefinitely. If a callback has to obtain some semaphores during its processing, then the callback should try to acquire all of them at the very beginning, and should be prepared to abort and return to EMan if it fails to acquire the necessary semaphores.

- EMan callback methods are called using name-lookup resolution. Therefore, the parameters to an EMan registration call must be such that the class object of the object parameter must be able to provide a pointer to the method indicated by the method parameter. Although this requirement is satisfied in a majority of cases, there are exceptions. For example, if the object is a proxy (in the DSOM sense) to a remote object, then the "real" class object cannot provide a meaningful method pointer. Also note that, when **somDispatch** is overridden, the effect of such an override will *not* apply to the callback from EMan. Do not use a method callback in these situations; instead, use a procedure callback.

9.3 Limitations

The present implementation of the Event Management framework has the limitations described below. For a more up-to-date list of limitations, refer to the README file on EMan in the SOMobjects Developer Toolkit.

- EMan supports registering a maximum of 64 AIX message queues.

- EMan can only wait on file descriptors (including files, pipes, sockets, and message queues) on AIX, and socket identifiers on OS/2 and Windows.

- EMan supports registering a maximum of FILENO (the AIX limit on maximum number of open files) file descriptors on AIX. On OS/2 and Windows, the maximum number of socket identifiers depends on the underlying Sockets class.

Use of EMan DLL

The Event Manager Framework uses a **Sockets** "select" call to wait on multiple sockets. . At the time of EMan creation, the **SOMEEMan** class object loads one of the **Sockets** subclass DLLs, based on the value of the environment variable SOMSOCKETS. This environment variable should name the implementation class of sockets (see Appendix C describing the **Sockets** abstract class and the specific implementation DLLs available with the SOMobjects Toolkit.) The current choices for this environment variable are **TCPIPSockets** (and **TCPIPSockets32** for OS/2), **NBSockets**, and **IPXSockets**.

Appendix A. SOMobjects Error Codes

SOM Kernel Error Codes

Following are error codes with messages/explanations for the SOM kernel and the various frameworks of the SOMobjects Developer Toolkit.

<u>Value</u>	<u>Symbolic Name and Description</u>
20011	**SOMERROR_CCNullClass** The **somDescendedFrom** method was passed a null class argument.
20029	**SOMERROR_SompntOverflow** The internal buffer used in **somPrintf** overflowed.
20039	**SOMERROR_MethodNotFound** **somFindMethodOk** failed to find the indicated method.
20049	**SOMERROR_StaticMethodTableOverflow** A Method–table overflow occurred in **somAddStaticMethod**.
20059	**SOMERROR_DefaultMethod** The **somDefaultMethod** was called; a defined method probably was not added before it was invoked.
20069	**SOMERROR_MissingMethod** The specified method was not defined on the target object.
20079	**SOMERROR_BadVersion** An attempt to load, create, or use a version of a class-object implementation is incompatible with the using program.
20089	**SOMERROR_NullId** The **SOM_CheckId** was given a null ID to check.
20099	**SOMERROR_OutOfMemory** Memory is exhausted.
20109	**SOMERROR_TestObjectFailure** The **somObjectTest** found problems with the object it was testing.
20119	**SOMERROR_FailedTest** The **somTest** detected a failure; generated only by test code.
20121	**SOMERROR_ClassNotFound** The **somFindClass** could not find the requested class.
20131	**SOMERROR_OldMethod** An old-style method name was used; change to an appropriate name.
20149	**SOMERROR_CouldNotStartup** The **somEnvironmentNew** failed to complete.
20159	**SOMERROR_NotRegistered** The **somUnloadClassFile** argument was not a registered class.
20169	**SOMERROR_BadOverride** The **somOverrideSMethod** was invoked for a method that was not defined in a parent class.
20179	**SOMERROR_NotImplementedYet** The method raising the error message is not implemented yet.
20189	**SOMERROR_MustOverride** The method raising the error message should have been overridden.
20199	**SOMERROR_BadArgument** An argument to a core SOM method failed a validity test.
20219	**SOMERROR_NoParentClass** During the creation of a class object, the parent class could not be found.
20229	**SOMERROR_NoMetaClass** During the creation of a class object, the metaclass object could not be found.

DSOM Error Codes

The following table lists the error codes that may be encountered when using DSOM. (Obsolete messages have been removed, thus message numbers do not compose a full sequence.)

Value	Description
30001	**SOMDERROR_NoMemory** Memory is exhausted.
30002	**SOMDERROR_NotImplemented** Function or method has a null implementation.
30004	**SOMDERROR_IO** I/O error while accessing a file located in SOMDDIR.
30008	**SOMDERROR_HostAddress** Unable to retrieve local host address.
30019	**SOMDERROR_NoMessages** No messages available (and caller specified "no wait").
30020	**SOMDERROR_UnknownAddress** Invalid client or server address.
30023	**SOMDERROR_CommTimeOut** Communications timeout. Make sure the DSOM daemon is running.
30026	**SOMDERROR_NoHostName** Unable to get host name.
30029	**SOMDERROR_BadEnvironment** Invalid Environment value in request message.
30031	**SOMDERROR_BadNVList** Invalid Named Value List (NVList).
30032	**SOMDERROR_BadFlag** Bad flag in NVList item.
30033	**SOMDERROR_BadLength** Bad length in NVList item.
30034	**SOMDERROR_BadObjref** Invalid object reference.
30036	**SOMDERROR_UnknownReposId** Attempt to use invalid Interface Repository identifier.
30037	**SOMDERROR_NVListAccess** Invalid NVList object in request message.
30038	**SOMDERROR_NVIndexError** Attempt to use an out-of-range NVList index.
30039	**SOMDERROR_SysTime** Error retrieving system time.
30041	**SOMDERROR_CouldNotStartProcess** System error: Unable to start a new process.
30042	**SOMDERROR_NoServerClass** No SOMDServer (sub)class specified for server implementation.
30043	**SOMDERROR_NoSOMDInit** SOM or DSOM has not been initialized.
30045	**SOMDERROR_NoImplDatabase** Could not open Implementation Repository database.
30046	**SOMDERROR_ImplNotFound** Implementation not found in implementation repository.

30047	**SOMDERROR_ClassNotFound**
	Class not found in implementation repository.
30048	**SOMDERROR_ServerNotFound**
	Server not found in **somdd**'s active server table.
30049	**SOMDERROR_ServerAlreadyExists**
	Server already exists in **somdd**'s active server table.
30050	**SOMDERROR_ServerNotActive**
	Server is not active.
30052	**SOMDERROR_ObjectNotFound**
	Could not find desired object.
30053	**SOMDERROR_NoParentClass**
	Unable to find / load parent class during proxy class creation.
30055	**SOMDERROR_BadTypeCode**
	Invalid type code.
30056	**SOMDERROR_BadDescriptor**
	Invalid method descriptor.
30059	**SOMDERROR_KeyNotFound**
	Internal object key not found.
30060	**SOMDERROR_CtxInvalidPropName**
	Illegal context property name.
30061	**SOMDERROR_CtxNoPropFound**
	Could not find property name in context.
30062	**SOMDERROR_CtxStartScopeNotFound**
	Could not find specified context start scope.
30063	**SOMDERROR_CtxAccess**
	Error accessing context object.
30064	**SOMDERROR_CouldNotStartThread**
	System error: Unable to start a new thread.
30065	**SOMDERROR_AccessDenied**
	System error: Access to a system resource (file, queue, shared memory, etc.) is denied.
30066	**SOMDERROR_BadParm**
	Invalid parameter supplied to an operating system call.
30072	**SOMDERROR_NoSpace**
	System error: No space left on device.
30089	**SOMDERROR_WrongRefType**
	Operation attempted on an object reference is incompatible with the reference type.
30090	**SOMDERROR_MustOverride**
	This method has no default implementation and must be overridden.
30091	**SOMDERROR_NoSocketsClass**
	Could not find/load Sockets class.
30092	**SOMDERROR_EManRegData**
	Unable to register DSOM events with the Event Manager.
30093	**SOMDERROR_NoRemoteComm**
	Remote communications is disabled (for Workstation DSOM).
30096	**SOMDERROR_GlobalAtomError**
	On Windows only, an error occurred while adding a segment name to the Windows atom table.

30097	**SOMDERROR_NamedMemoryTableError**
	On Windows only, an error occurred while creating or deleting a (named) shared memory segment.
30098	**SOMDERROR_WMQUIT**
	On Windows only, indicates DSOM received a Windows WM_QUIT message. The developer of a server application should check for SOMDERROR_WMQUIT returned from method **execute_request_loop** and handle the error by cleaning up and exiting.
30105	**SOMDERROR_DuplicateImplEntry**
	Implementation repository identifier already exists. Add wait time between 'regimpl' calls.
30106	**SOMDERROR_InvalidSOMSOCKETS**
	SOMSOCKETS environment variable set inorrectly.
30107	**SOMDERR0R_IRNotFound**
	Interface Repository not found.
30108	**SOMDERR0R_ClassNotInIR**
	Attempt to create an object whose Class is not in the Interface Repository.
30110	**SOMDERROR_SocketError**
	A communications socket error has occurred. Make sure the DSOM daemon is running.
30111	**SOMDERROR_PacketError**
	A communications packet error has occurred.
30112	**SOMDERROR_Marshal**
	A marshalling error has occurred.
30113	**SOMDERROR_NotProcessOwner**
	On AIX only, the server cannot be killed because you are not the process owner.
30114	**SOMDERROR_ServerInactive**
	The requested server is not running.
30115	**SOMDERROR_ServerDisabled**
	The server has been disabled by the program servmgr.
XXXXX	**SOMDERROR_OperatingSystem**
	On AIX, this is the value of the C error variable "errno" defined in errno.h; on OS/2 and Windows, it is the DOS API return code.

Appendix B. SOM IDL Language Grammar

```
specification      : [comment] definition+

definition         : type_dcl ; [comment]
                   | const_dcl ; [comment]
                   | interface ; [comment]
                   | module    ; [comment]
                   | pragma_stm

module             : module identifier [comment]
                     { [comment] definition+ }

interface          : interface identifier
                   | interface_dcl

interface_dcl      : interface identifier [inheritance] [comment]
                     { [comment] export* } [comment]

inheritance        : : scoped_name {, scoped_name}*

export             : type_dcl ; [comment]
                   | const_dcl ; [comment]
                   | attr_dcl ; [comment]
                   | op_dcl ; [comment]
                   | implementation_body ; [comment]
                   | pragma_stm

scoped_name        : identifier
                   | :: identifer
                   | scoped_name :: identifer

const_dcl          : const const_type identifier = const_expr

const_type         : integer_type
                   | char_type
                   | boolean_type
                   | floating_pt_type
                   | string_type
                   | scoped_name

const_expr         : or_expr

or_expr            : xor_expr
                   | or_expr | xor_expr

xor_expr           : and_expr
                   | xor_expr ^ and_expr

and_expr           : shift_expr
                   | and_expr & shift_expr

shift_expr         : add_expr
                   | shift_expr >> add_expr
                   | shift_expr <<  add_expr

add_expr           : mult_expr
                   | add_expr + mult_expr
                   | add_expr − mult_expr

mult_expr          : unary_expr
                   | mult_expr * unary_expr
                   | mult_expr / unary_expr
                   | mult_expr % unary_expr
```

```
unary_expr              : unary_operator primary_expr
                        | primary_expr

unary_operator          : -
                        | +
                        | ~

primary_expr            : scoped_name
                        | literal
                        | ( const_expr )

literal                 : integer_literal
                        | string_literal
                        | character_literal
                        | floating_pt_literal
                        | boolean_literal

type_dcl                : typedef type_declarator
                        | constr_type_spec

type_declarator         : type_spec declarator {, declarator}*

type_spec               : simple_type_spec
                        | constr_type_spec

simple_type_spec        : base_type_spec
                        | template_type_spec
                        | scoped_name

base_type_spec          : floating_pt_type
                        | integer_type
                        | char_type
                        | boolean_type
                        | octet_type
                        | any_type
                        | voidptr_type

template_type_spec      : sequence_type
                        | string_type

constr_type_spec        : struct_type
                        | union_type
                        | enum_type

declarator              : [stars] std_declarator

std_declarator          : simple_declarator
                        | complex_declarator

simple_declarator       : identifier

complex_declarator      : array_declarator

array_declarator        : simple_declarator fixed_array_size+

fixed_array_size        : [ const_expr ]

floating_pt_type        : float
                        | double

integer_type            : signed_int
                        | unsigned_int

signed_int              : long
                        | short
```

```
unsigned_int       : unsigned signed_int

char_type          : char

boolean_type       : boolean

octet_type         : octet

any_type           : any

voidptr_type       : void stars

struct_type        : (struct|exception) identifier
                   | (struct|exception) [comment]
                    { [comment] member* }

member             : type_declarator ; [comment]

union_type         : union identifier
                   | union identifier switch
                     ( switch_type_spec ) [comment]
                     { [comment] case+ }

switch_type_spec   : integer_type
                   | char_type
                   | boolean_type
                   | enum_type
                   | scoped_name

case               : case_label+ element_spec ; [comment]

case_label         : case const_expr : [comment]
                   | default : [comment]

element_spec       : type_spec declarator

enum_type          : enum identifier { identifier
                     {, identifier}* [comment] }

sequence_type      : sequence < simple_type_spec , const_expr >
                   | sequence < simple_type_spec >

string_type        : string < const_expr >
                   | string

attr_dcl           : [readonly] attribute simple_type_spec
                     declarator {, declarator}*

op_dcl             : [oneway] op_type_spec [stars] identifier
                      parameter_dcls [raises_expr] [context_expr]

op_type_spec       : simple_type_spec
                   | void

parameter_dcls     : ( param_dcl {, param_dcl}* [comment] )
                   | ( )

param_dcl          : param_attribute simple_type_spec declarator

param_attribute    : in
                   | out
                   | inout

raises_expr        : raises ( scope_name+ )
```

```
context_expr          : context ( context_string {, context_string}* )

implementation_body : implementation [comment]
                        { [comment] implementation+ }

implementation        : modifier_stm
                        | pragma_stm
                        | passthru
                        | member

pragma_stm            : #pragma modifier modifier_stm
                        | #pragma somtemittypes on
                        | #pragma somtemittypes off

modifier_stm          : smidentifier : [modifier {, modifier}*] ;
[comment]
                        | modifier ; [comment]

modifier              : smidentifier
                        | smidentifier = modifier_value

modifier_value        : smidentifier
                        | string_literal
                        | integer_literal
                        | keyword

passthru              : passthru identifier = string_literal+ ;
[comment]

smidentifier          : identifer
                        | __identifier

stars                 | *+
```

Appendix C. Implementing Sockets Subclasses

Distributed SOM (DSOM) requires basic message services for inter-process communications. The Event Management Framework must be integrated with the same communication services in order to handle communications events.

To maximize their portability to a wide variety of local area network transport protocols, the DSOM and Event Management Frameworks have been written to use a *common communications interface*, which is implemented by one or more SOM class libraries using available local protocols.

The common communications interface is based on the "sockets" interface used with TCP/IP, since its interface and semantics are fairly widespread and well understood. The IDL interface is named **Sockets**. There is no implementation associated with the **Sockets** interface by default; specific protocol implementations are supplied by subclass implementations.

Note: The **Sockets** classes supplied with the SOMobjects Developer Toolkit and Enabler packages are *only* intended to support the DSOM and Event Management Frameworks. These class implementations are *not* intended for general application usage.

Available **Sockets** subclasses by SOMobjects product are as follows:

- For AIX:
  TCPIPSockets class for TCP/IP,
  IPXSockets class for NetWare IPX/SPX, and
  NBSockets class for NetBIOS.

- For OS/2 and Windows:
  TCPIPSockets class (a) for TCP/IP for Windows or (b) for TCP/IP 1.2.1 on OS/2,
  TCPIPSockets32 class for TCP/IP 2.0 on OS/2 only (see Note below),
  IPXSockets class for NetWare IPX/SPX, and
  NBSockets class for NetBIOS.

Note: The **TCPIPSockets32** class gives greater performance over the **TCPIPSockets** class on OS/2, but requires the 32–bit version of TCP/IP (version 2.0) rather than the 16–bit version of TCP/IP (version 1.2.1).

Application developers may need to develop their own **Sockets** subclass if the desired transport protocol or product version is not one of those supported by the SOMobjects run-time packages. This appendix explains how to approach the implementation of a **Sockets** subclass, if necessary. Warning: this may be a non-trivial exercise!

Sockets IDL interface

The base **Sockets** interface is expressed in IDL in the file **somssock.idl**, listed below. There is a one-to-one mapping between TCP/IP socket APIs and the methods defined in the **Sockets** interface.

Please note the following:

- The semantics of each of the **Sockets** methods must be that of the corresponding TCP/IP call. Currently, only Internet address family (AF_INET) addresses are used by the frameworks.

  (The TCP/IP sockets API is not documented as part of the SOMobjects Developer Toolkit. The implementor is referred to the programming references for IBM TCP/IP for AIX or OS/2, or to similar references that describe the sockets interface for TCP/IP.)

- Data types, constants, and macros which are part of the **Sockets** interface are defined in a C include file, **soms.h**. This file is supplied with the SOMobjects Toolkit, and is not shown in this manual.

- The **Sockets** interface is expressed in terms of a 32-bit implementation.

- Some of the method parameters and return values are expressed using pointer types, for example:

```
hostent *somsGethostent ();
```

This has been done to map TCP/IP socket interfaces as directly as possible to their IDL equivalent. (Use of strict CORBA IDL was not a primary goal for the **Sockets** interface, since it is only used internally by the frameworks.)

- The **Sockets** class and its subclasses are *single instance* classes.

Following is a listing of the file **somssock.idl**. Each socket call is briefly described with a comment.

```
// 96F8647, 96F8648 (C) Copyright IBM Corp. 1992, 1993
// All Rights Reserved
// Licensed Materials – Property of IBM

#ifndef somssock_idl
#define somssock_idl

#include <somobj.idl>
#include <snglicls.idl>

interface Sockets : SOMObject
{
    //# The following typedefs are fully defined in <soms.h>.
    typedef SOMFOREIGN sockaddr;
    #pragma modifier sockaddr : impctx="C", struct;
    typedef SOMFOREIGN iovec;
    #pragma modifier iovec : impctx="C", struct;
    typedef SOMFOREIGN msghdr;
    #pragma modifier msghdr : impctx="C", struct;
    typedef SOMFOREIGN fd_set;
    #pragma modifier fd_set : impctx="C", struct;
    typedef SOMFOREIGN timeval;
    #pragma modifier timeval : impctx="C", struct;
    typedef SOMFOREIGN hostent;
    #pragma modifier hostent : impctx="C", struct;
    typedef SOMFOREIGN servent;
    #pragma modifier servent : impctx="C", struct;
    typedef SOMFOREIGN in_addr;
    #pragma modifier in_addr : impctx="C", struct;

    long somsAccept (in long s, out sockaddr name, out long
namelen);
    // Accept a connection request from a client.

    long somsBind (in long s, inout sockaddr name, in long namelen);
    // Binds a unique local name to the socket with descriptor s.

    long somsConnect (in long s, inout sockaddr name,
                      in long namelen);
    // For streams sockets, attempts to establish a connection
    // between two sockets.  For datagram sockets, specifies the
    // socket's peer.

    hostent *somsGethostbyaddr (in char *addr, in long addrlen,
                                in long domain);
    // Returns a hostent structure for the host address specified on
    // the call.

    hostent *somsGethostbyname (in string name);
    // Returns a hostent structure for the host name specified on
    // the call.

    hostent *somsGethostent ();
    // Returns a pointer to the next entry in the hosts file.
```

```
unsigned long somsGethostid ();
// Returns the unique identifier for the current host.

long somsGethostname (in string name, in long namelength);
// Retrieves the standard host name of the local host.

long somsGetpeername (in long s, out sockaddr name,
                      out long namelen);
// Gets the name of the peer connected to socket s.

servent *somsGetservbyname (in string name, in string protocol);
// Retrieves an entry from the /etc/services file using the
// service name as a search key.

long somsGetsockname (in long s, out sockaddr name,
                      out long namelen);
// Stores the current name for the socket specified by the s
// parameter into the structure pointed to by the name
// parameter.

long somsGetsockopt (in long s, in long level, in long optname,
                     in char *optval, out long option);
// Returns the values of socket options at various protocol
// levels.

unsigned long somsHtonl (in unsigned long a);
// Translates an unsigned long integer from host-byte order to
// network-byte order.

unsigned short somsHtons (in unsigned short a);
// Translates an unsigned short integer from host-byte order to
// network-byte order.

long somsIoctl (in long s, in long cmd, in char *data,
                in long length);
// Controls the operating characteristics of sockets.

unsigned long somsInet_addr (in string cp);
// Interprets character strings representing numbers expressed
// in standard '.' notation and returns numbers suitable for use
// as internet addresses.

unsigned long somsInet_lnaof (in in_addr addr);
// Breaks apart the internet address and returns the local
// network address portion.

in_addr somsInet_makeaddr (in unsigned long net,
                           in unsigned long lna);
// Takes a network number and a local network address and
// constructs an internet address.

unsigned long somsInet_netof (in in_addr addr);
// Returns the network number portion of the given internet
// address.

unsigned long somsInet_network (in string cp);
// Interprets character strings representing numbers expressed
// in standard '.' notation and returns numbers suitable for use
// as network numbers.

string somsInet_ntoa (in in_addr addr);
// Returns a pointer to a string expressed in the dotted-decimal
// notation.

long somsListen (in long s, in long backlog);
// Creates a connection request queue of length backlog to queue
// incoming connection requests, and then waits for incoming
// connection requests.
```

```
unsigned long somsNtohl (in unsigned long a);
// Translates an unsigned long integer from network-byte order
// to host-byte order.

unsigned short somsNtohs (in unsigned short a);
// Translates an unsigned short integer from network-byte order
// to host-byte order.

long somsReadv (in long s, inout iovec iov, in long iovcnt);
// Reads data on socket s and stores it in a set of buffers
// described by iov.

long somsRecv (in long s, in char *buf, in long len,
               in long flags);
// Receives data on streams socket s and stores it in buf.

long somsRecvfrom (in long s, in char *buf, in long len,
    in long flags, out sockaddr name, out long namelen);
// Receives data on datagram socket s and stores it in buf.

long somsRecvmsg (in long s, inout msghdr msg, in long flags);
// Receives messages on a socket with descriptor s and stores
// them in an array of message headers.

long somsSelect (in long nfds, inout fd_set readfds,
                 inout fd_set writefds, inout fd_set exceptfds,
                 inout timeval timeout);
// Monitors activity on a set of different sockets until a
// timeout expires, to see if any sockets are ready for reading
// or writing, or if an exceptional condition is pending.

long somsSend (in long s, in char *msg, in long len,
               in long flags);
// Sends msg on streams socket s.

long somsSendmsg (in long s, inout msghdr msg, in long flags);
// Sends messages passed in an array of message headers on a
// socket with descriptor s.

long somsSendto (in long s, inout char msg, in long len,
               in long flags, inout sockaddr to, in long tolen);
// Sends msg on datagram socket s.

long somsSetsockopt (in long s, in long level, in long optname,
                     in char *optval, in long optlen);
// Sets options associated with a socket.

long somsShutdown (in long s, in long how);
// Shuts down all or part of a full-duplex connection.

long somsSocket (in long domain, in long type,
                 in long protocol);
// Creates an endpoint for communication and returns a socket
// descriptor representing the endpoint.

long somsSoclose (in long s);
// Shuts down socket s and frees resources allocated to the
// socket.

long somsWritev (in long s, inout iovec iov, in long iovcnt);
// Writes data on socket s.  The data is gathered from the
// buffers described by iov.

attribute long serrno;
// Used to pass error numbers.
```

```
#ifdef __SOMIDL__
    implementation
    {
    releaseorder:
        somsAccept, somsBind, somsConnect, somsGethostbyaddr,
        somsGethostbyname, somsGethostent, somsGethostid,
        somsGethostname, somsGetpeername, somsGetsockname,
        somsGetsockopt, somsHtonl, somsHtons, somsIoctl,
        somsInet_addr, somsInet_lnaof, somsInet_makeaddr,
        somsInet_netof, somsInet_network, somsInet_ntoa,
        somsListen, somsNtohl, somsNtohs, somsReadv,
        somsRecv, somsRecvfrom, somsRecvmsg, somsSelect,
        somsSend, somsSendmsg, somsSendto, somsSetsockopt,
        somsShutdown, somsSocket, somsSoclose, somsWritev,
        _set_serrno, _get_serrno, somsGetservbyname;

    //# Class modifiers
    callstyle=idl;
    metaclass = SOMMSingleInstance;
    majorversion=1; minorversion=1;
    dll="soms.dll";
    };
#endif /* __SOMIDL__ */
};
#endif  /* somssock_idl */
```

IDL for a Sockets subclass

Sockets subclasses inherit their entire interface from **Sockets.** All methods are overridden.

For example, here is a listing of the **TCPIPSockets** IDL description.

```
// 96F8647, 96F8648 (C) Copyright IBM Corp. 1992, 1993
// All Rights Reserved
// Licensed Materials – Property of IBM

#ifndef tcpsock_idl
#define tcpsock_idl

#include <somssock.idl>
#include <snglicls.idl>

interface TCPIPSockets : Sockets
{
#ifdef __SOMIDL__
    implementation
    {
    //# Class modifiers
    callstyle=idl;
    majorversion=1; minorversion=1;
    dllname="somst.dll";
    metaclass=SOMMSingleInstance;
```

```
        //# Method modifiers
        somsAccept: override;
        somsBind: override;
        somsConnect: override;
        somsGethostbyaddr: override;
        somsGethostbyname: override;
        somsGethostent: override;
        somsGethostid: override;
        somsGethostname: override;
        somsGetpeername: override;
        somsGetservbyname: override;
        somsGetsockname: override;
        somsGetsockopt: override;
        somsHtonl: override;
        somsHtons: override;
        somsIoctl: override;
        somsInet_addr: override;
        somsInet_lnaof: override;
        somsInet_makeaddr: override;
        somsInet_netof: override;
        somsInet_network: override;
        somsInet_ntoa: override;
        somsListen: override;
        somsNtohl: override;
        somsNtohs: override;
        somsReadv: override;
        somsRecv: override;
        somsRecvfrom: override;
        somsRecvmsg: override;
        somsSelect: override;
        somsSend: override;
        somsSendmsg: override;
        somsSendto: override;
        somsSetsockopt: override;
        somsShutdown: override;
        somsSocket: override;
        somsSoclose: override;
        somsWritev: override;
        _set_serrno: override;
        _get_serrno: override;
        };
#endif /* __SOMIDL__ */
};

#endif  /* tcpsock_idl */
```

Implementation considerations

- Only the AF_INET address family must be supported. That is, the DSOM and Event Manager frameworks both use Internet addresses and port numbers to refer to specific sockets.

- On OS/2, the SOMobjects run-time libraries were built using the C Set/2 32-bit compiler. If the underlying subclass implementation uses a 16-bit subroutine library, conversion of the method call arguments may be required. (This mapping of arguments is often referred to as "thunking.")

- **Sockets** subclasses to be used in multi-threaded environments should be made thread-safe. That is, it is possible that concurrent threads may make calls on the (single) **Sockets** object, so data structures must be protected within critical regions, as appropriate.

- Valid values for the **serrno** attribute are defined in the file **soms.h**. The subclass implementation should map local error numbers into the appropriate corresponding **Sockets** error numbers.

Example code

The following code fragment shows an example of the implementation of the **somsBind** method of the **TCPIPSockets** subclass, for both AIX and OS/2. The sample illustrates that, for TCP/IP, the implementation is basically a one-to-one mapping of **Sockets** methods onto TCP/IP calls. For other transport protocols, the mapping from the socket abstraction to the protocol's API may be more difficult.

For AIX, the mapping from **Sockets** method to TCP/IP call is trivial.

```
SOM_Scope long  SOMLINK somsBind(TCPIPSockets somSelf,
                                 Environment *ev,
                                 long s, Sockets_sockaddr* name,
                                 long namelen)
{
    long rc;

    TCPIPSocketsMethodDebug("TCPIPSockets","somsBind");

    rc = (long) bind((int)s, name, (int)namelen);

    if (rc == -1)
        __set_serrno(somSelf, ev, errno);

    return rc;
}
```

On OS/2, however, the TCP/IP Release 1.2.1 library is a 16-bit library. Consequently, many of the method calls require conversion ("thunking") of 32-bit parameters into 16-bit parameters, before the actual TCP/IP calls can be invoked. For example, the function prototype for the **somsBind** method is defined as:

```
SOM_Scope long  SOMLINK somsBind(TCPIPSockets somSelf,
                                 Environment *ev,
                                 long s, Sockets_sockaddr* name,
                                 long namelen);
```

whereas the file **socket.h** on OS/2 declares the **bind** function with the following prototype:

```
short _Far16 _Cdecl bind(short /*s*/, void * _Seg16 /*name*/,
                         short /*len*/);
```

In this case, the pointer to the "name" structure, passed as a 32-bit address, cannot be used directly in the **bind** call: a 16-bit address must be passed instead. This can be accomplished by dereferencing the 32-bit pointer provided by the "name" parameter in the **somsBind** call, copying the caller's Sockets_sockaddr structure into a local structure ("name16"), and then passing the address of the local structure ("&name16") as a 16-bit address in the **bind** call.

```
SOM_Scope long  SOMLINK somsBind(TCPIPSockets somSelf,
                                 Environment *ev,
                                 long s, Sockets_sockaddr* name,
                                 long namelen)
{
    long rc;
    Sockets_sockaddr name16;

    TCPIPSocketsMethodDebug("TCPIPSockets","somsBind");

    /* copy user's parameter into a local structure */
    memcpy ((char *)&name16, (char *)((sockaddr32 *)name), namelen);
    rc = (long) bind((short)s, (void *)&name16, (short)namelen);

    if (rc == -1)
        __set_serrno(somSelf, ev, tcperrno());

    return rc;
}
```

For Windows, a developer would follow the OS/2 example for implementing the **bind** function with 16–bit addresses (but using the **IPXSockets** class for NetWare IPX/SPX or the **NBSockets** class for NetBIOS, rather than the **TCPIPSockets** class).

Glossary

Note: In the following definitions, words shown in *italics* are terms for which separate glossary entries are also defined.

abstract class
A *class* that is not designed to be instantiated, but serves as a *base class* for the definition of subclasses. Regardless of whether an abstract class inherits *instance data* and *methods* from *parent classes*, it will always introduce methods that must be *overridden* in a *subclass,* in order to produce a class whose objects are semantically valid*.*

affinity group An array of *class objects* that were all registered with the *SOMClassMgr* object during the dynamic loading of a *class*. Any class is a member of at most one affinity group.

ancestor class
A *class* from which another class inherits *instance methods, attributes,* and *instance variables*, either directly or indirectly. A direct descendant of an ancestor class is called a *child class, derived class,* or *subclass.* A direct ancestor of a class is called a *parent class, base class*, or *superclass*.

aggregate type
A user-defined data type that combines basic types (such as, char, short, float, and so on) into a more complex type (such as structs, arrays, strings, sequences, unions, or enums).

apply stub A *procedure* corresponding to a particular *method* that accepts as arguments: the *object* on which the method is to be invoked, a pointer to a location in memory where the method's result should be stored, a pointer to the method's procedure, and the method's arguments in the form of a va_list. The apply stub extracts the arguments from the va_list, invokes the method with its arguments, and stores its result in the specified location. Apply stubs are registered with class objects when instance methods are defined, and are invoked using the somApply function. Typically, *implementations* that *override* somDispatch call somApply to invoke a method on a va_list of arguments.

attribute A specialized syntax for declaring "set" and "get" methods. Method names corresponding to attributes always begin with "_set_" or "_get_". An attribute name is declared in the body of the *interface statement* for a class. Method procedures for get/set methods are automatically defined by the *SOM Compiler* unless an attribute is declared as "noget/noset". Likewise, a corresponding *instance variable* is automatically defined unless an attribute is declared as "nodata". IDL also supports "readonly" attributes, which specify only a "get" method. (Contrast an attribute with an *instance variable.*)

auxiliary class data structure
A structure provided by the SOM API to support efficient static access to *class*-specific information used in dealing with SOM *objects*. The structure's name is *<className>*CClassData. Its first component (parentMtab) is a list of *parent-class method tables* (used to support efficient parent method calls). Its second component (instanceDataToken) is the *instance token* for the class (generally used to locate the *instance data* introduced by *method procedures* that implement *methods* defined by the class).

base class See *parent class.*

behavior (of an object)
The *methods* that an *object* responds to. These methods are those either introduced or inherited by the *class* of the object. See also *state.*

bindings   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. The *SOM Compiler* generates binding files for C and C++. These binding files include an *implementation template* for the class and two header files, one to be included in the class's implementation file and the other in client programs.

BOA (basic object adapter) class

A CORBA *interface* (represented as an *abstract class* in DSOM), which defines generic *object-adapter (OA)* methods that a *server* can use to register itself and its *objects* with an *ORB (object request broker)*. See also *SOMOA (SOM object adapter) class.*

callback   A user-provided procedure or method to the Event Management Framework that gets invoked when a registered event occurs. (See also *event*).

casted dispatching

A form of method dispatching that uses *casted method resolution*; that is, it uses a designated ancestor class of the actual *target object's* class to determine what procedure to call to execute a specified method.

casted method resolution

A *method resolution* technique that uses a *method procedure* from the *method table* of an ancestor of the *class* of an *object* (rather than using a procedure from the method table of the object's own class).

child class   A class that inherits *instance methods, attributes,* and *instance variables* directly from another class, called the *parent class, base class*, or *superclass,* or indirectly from an *ancestor class.* A child class may also be called a *derived class* or *subclass.*

class   A way of categorizing *objects* based on their behavior (the *methods* they support) and shape (memory layout). A class is a definition of a generic object. In SOM, a class is also a special kind of object that can manufacture other objects that all have a common shape and exhibit similar behavior. The specification of what comprises the shape and behavior of a set of objects is referred to as the "definition" of a class. New classes are defined in terms of existing classes through a technique known as *inheritance*. See also *class object.*

class variable *Instance data* of a *class object*. All instance data of an *object* is defined (through either introduction or *inheritance*) by the object's class. Thus, class variables are defined by *metaclasses*.

class data structure

A structure provided by the SOM API to support efficient static access to *class*-specific information used in dealing with SOM *objects*. The structure's name is *<className>*ClassData. Its first component (classObject) is a pointer to the corresponding *class object*. The remaining components (named after the *instance methods* and *instance variables*) are *method tokens* or *data tokens*, in order as specified by the class's implementation. Data tokens are only used to support data (public and private) introduced by classes declared using *OIDL*; IDL *attributes* are supported with method tokens.

class manager

An *object* that acts as a run-time registry for all SOM *class objects* that exist within the current process and which assists in the dynamic loading and unloading of class libraries. A class implementor can define a customized class manager by subclassing *SOMClassMgr* class to replace the SOM-supplied SOMClassMgrObject. This is done to augment the functionality of the default class-management registry (for example, to coordinate the automatic quiescing and unloading of classes).

class method (Also known as *factory method* or *constructor.*) A class method is a *method* that a *class object* responds to (as opposed to an *instance method*). A class method that class <X> responds to is provided by the *metaclass* of class <X>. Class methods are executed without requiring any *instances* of class <X> to exist, and are frequently used to create instances of the class.

class object The run-time *object* representing a SOM *class.* In SOM, a class object can perform the same behavior common to all *objects*, inherited from *SOMObject.*

client code (Or *client program* or *client.*) An application program, written in the programmer's preferred language, which invokes *methods* on *objects* that are *instances* of SOM *classes.* In DSOM, this could be a program that invokes a method on a remote object.

constructor See *class method*.

context expression

An optional expression in a method's IDL declaration, specifying identifiers whose value (if any) can be used during SOM's *method resolution* process and/or by the *target object* as it executes the *method procedure.* If a context expression is specified, then a related Context parameter is required when the method is invoked. (This Context parameter is an *implicit parameter* in the IDL specification of the method, but it is an explicit parameter of the method's procedure.) No SOM-supplied methods require context parameters.

CORBA The Common Object Request Broker Architecture established by the Object Management Group. *Interface Definition Language* used to describe the *interface* for SOM classes is fully compliant with CORBA standards.

data token A value that identifies a specific *instance variable* within an *object* whose *class inherits* the instance variable (as a result of being derived, directly or indirectly, from the class that introduces the instance variable). An object and a data token are passed to the SOM run-time procedure, somDataResolve, which returns is a pointer to the specific instance variable corresponding to the data token. (See also *instance token.*)

derived class See *subclass* and *subclassing.*

derived metaclass

(Or *SOM-derived metaclass.*) A *metaclass* that SOM creates automatically (often even when the *class* implementor specifies an explicit metaclass) as needed to ensure that, for any code that executes without *method-resolution* error on an *instance* of a given class, the code will similarly execute without method-resolution error on instances of any *subclass* of the given class. SOM's ability to derive such metaclasses is a fundamental necessity in order to ensure binary compatibility for client programs despite any subsequent changes in class *implementations.*

descriptor (Or *method descriptor.*) An ID representing the identifier of a *method* definition or an *attribute* definition in the Interface Repository. The IR definition contains information about the method's return type and the type of its arguments.

directive A message (a pre-defined character constant) received by a *replica* from the Replication Framework. Indicates a potential failure situation.

dirty object A persistent *object* that has been modified since it was last written to persistent storage.

dispatch-function resolution

Dispatch-function resolution is the slowest, but most flexible, of the three *method-resolution* techniques SOM offers. Dispatch functions permit method resolution to be based on arbitrary rules associated with an *object*'s *class.* Thus, a class implementor has complete freedom in determining how methods invoked on its *instances* are resolved. See also *dispatch method* and *dynamic dispatching.*

dispatch method

A *method* (such as somDispatch or somClassDispatch) that is invoked (and passed an argument list and the ID of another method) in order to determine the appropriate *method procedure* to execute. The use of dispatch methods facilitates *dispatch-function resolution* in SOM applications and enables method invocation on remote objects in DSOM applications. See also *dynamic dispatching*.

dynamic dispatching

Method dispatching using *dispatch-function resolution*; the use of dynamic *method resolution* at run time. See also *dispatch-function resolution* and *dynamic method.*

Dynamic Invocation Interface (DII)

The *CORBA*-specified *interface*, implemented in DSOM, that is used to dynamically build requests on remote *objects*. Note that DSOM applications can also use the somDispatch method for *dynamic method* calls when the object is remote. See also *dispatch method.*

dynamic method

A method that is not declared in the IDL *interface statement* for a *class* of *objects*, but is added to the *interface* at run time, after which *instances* of the class (or of its *subclasses*) will respond to the registered dynamic method. Because dynamic methods are not declared, *usage bindings* for SOM classes cannot support their use; thus, *offset method resolution* is not available. Instead, *name-lookup* or *dispatch-function method resolution* must be used to invoke dynamic methods. (There are currently no known uses of dynamic methods by any SOM applications.) See also *method* and *static method.*

encapsulation

An object-oriented programming feature whereby the implementation details of a class are hidden from client programs, which are only required to know the *interface* of a *class* (the signatures of its *methods* and the names of its *attributes*) in order to use the class's methods and attributes.

encoder/decoder

In the Persistence Framework, a *class* that knows how to read/write the persistent object format of a *persistent object*. Every persistent object is associated with an Encoder/Decoder, and an encoder/decoder object is created for each *attribute* and *instance variable*. An Encoder/Decoder is supplied by the Persistence Framework by default, or an application can define its own.

entry class   In the Emitter Framework, a *class* that represents some syntactic unit of an *interface* definition in the *IDL source file.*

Environment parameter

A *CORBA*-required parameter in all *method procedures*, it represents a memory location where exception information can be returned by the *object* of a method invocation. [Certain methods are exempt (when the class contains a modifier of callstyle=oidl), to maintain upward compatibility for client programs written using an earlier release.]

emitter   Generically, a program that takes the output from one system and converts the information into a different form. Using the Emitter Framework, selected output from the *SOM Compiler* (describing each syntactic unit in an *IDL source file*) is transformed and formatted according to a user-defined template. Example emitter output, besides the implementation template and language bindings, might include reference documentation, class browser descriptions, or "pretty" printouts.

event   The occurrence of a condition, or the beginning or ending of an activity that is of interest to an application. Examples are elapse of a time interval, sending or receiving of a message, and opening or closing a file. (See also *event manager* and *callback*.)

event manager (EMan)

The chief component of the Event Management Framework that registers interest in various *events* from calling modules and informs them through *callbacks* when those events occur.

factory method See *class method*.

ID See *somId.*

IDL source file

A user-written .idl file, expressed using the syntax of the *Interface Definition Language* (IDL), which describes the *interface* for a particular *class* (or classes, for a *module*). The IDL source file is processed by the *SOM Compiler* to generate the *binding files* specific to the programming languages of the class implementor and the client application. (This file may also be called the "IDL file," the "source file," or the "interface definition file.")

implementation

(Or *object implementation*.) The specification of what *instance variables* implement an *object's* *state* and what *procedures* implement its *methods* (or *behaviors*). In DSOM, a remote object's implementation is also characterized by its *server implementation* (a program).

Implementation Repository

A database used by DSOM to store the implementation definitions of DSOM *servers*.

implementation statement

An optional declaration within the body of the *interface* definition of a *class* in a SOM *IDL source file,* specifying information about how the class will be implemented (such as, version numbers for the class, overriding of inherited methods, or type of method resolution to be supported by particular methods). This statement is a SOM-unique statement; thus, it must be preceded by the term "#ifdef __SOMIDL__" and followed by "#endif". See also *interface declaration.*

implementation template

A template file containing *stub procedures* for *methods* that a *class* introduces or *overrides*. The implementation template is one of the *binding files* generated by the *SOM Compiler* when it processes the *IDL source file* containing class *interface declarations.* The class implementor then customizes the *implementation,* by adding language-specific code to the *method procedures*.

implicit method parameter

A *method* parameter that is not included in the IDL *interface* specification of a method, but which is a parameter of the *method's* *procedure* and which is required when the method is invoked from a *client program*. Implicit parameters include the required *Environment* parameter indicating where exception information can be returned, as well as a *Context* parameter, if needed.

incremental update

A revision to an *implementation template* file that results from reprocessing of the *IDL source file* by the *SOM Compiler*. The updated implementation file will contain new *stub procedures*, added comments, and revised *method prototypes* reflecting changes made to the *method* definitions in the IDL specification. Importantly, these updates do not disturb existing code that the class implementor has defined for the prior method procedures.

inheritance   The technique of defining one *class* (called a *subclass, derived class,* or *child class*) as incremental differences from another class (called the *parent class, base class, superclass,* or *ancestor class*). From its parents, the subclass inherits variables and *methods* for its *instances*. The subclass can also provide additional *instance variables* and methods. Furthermore, the subclass can provide new procedures for implementing inherited methods. The subclass is then said to *override* the parent class's methods. An overriding method procedure can elect to call the parent class's *method procedure*. (Such a call is known as a *parent method call.*)

inheritance hierarchy

The sequential relationship from a root class to a subclass, through which the subclass inherits *instance methods, attributes,* and *instance variables* from all of its ancestors, either directly or indirectly. The root class of all SOM classes is SOMObject.

instance   (Or *object instance* or just *object*.) A specific object, as distinguished from a *class* of objects. See also *object.*

instance method

A method valid for an object *instance* (as opposed to a *class method*, which is valid for a *class object*). An instance method that an object responds to is defined by its class or inherited from an ancestor class.

instance token

A *data token* that identifies the first *instance variable* among those introduced by a given *class*. The somGetInstanceToken *method* invoked on a *class object* returns that class's instance token.

instance variables

(Or, *instance data*.) Variables declared for use within the *method procedures* of a *class*. An instance variable is declared within the body of the *implementation statement* in a SOM *IDL source file*. An instance variable is "private" to the class and should not be accessed by a client program. (Contrast an instance variable with an *attribute.*)

interface   The information that a *client* must know to use a *class* — namely, the names of its *attributes* and the signatures of its *methods*. The interface is described in a formal language (the *Interface Definition Language*, IDL) that is independent of the programming language used to implement the class's methods.

interface declaration

(Or *interface statement.*) The statement in the *IDL source file* that specifies the name of a new class and the names of its *parent class*(es). The "body" of the interface declaration defines the *signature* of each new *method* and any *attribute*(s) associated with the class. In SOM IDL, the body may also include an *implementation statement* (where *instance variables* are declared or a *modifier* is specified, for example to *override* a method).

Interface Definition Language (IDL)

The formal language (independent of any programming language) by which the *interface* for a *class* of *objects* is defined in a .idl file, which the *SOM Compiler* then interprets to create an *implementation template* file and *binding* files. SOM's Interface Definition Language is fully compliant with standards established by the Object Management Group's Common Object Request Broker Architecture (*CORBA*).

Interface Repository (IR)

The database that SOM optionally creates to provide persistent storage of objects representing the major elements of *interface* definitions. Creation and maintenance of the IR is based on information supplied in the *IDL source file.* The SOM IR Framework supports all interfaces described in the *CORBA* standard.

Interface Repository Framework

A set of *classes* that provide *methods* whereby executing programs can access the persistent objects of the *Interface Repository* to discover everything known about the programming *interfaces* of SOM classes.

macro   An alias for executing a sequence of hidden instructions; in SOM, typically the means of executing a command known within a *binding file* created by the *SOM Compiler*.

metaclass   A *class* whose *instances* are classes. In SOM, any class descended from *SOMClass* is a metaclass. The *methods* a class inherits from its metaclass are sometimes called *class methods* (in Smalltalk) or *factory methods* (in Objective-C) or *constructors*. See also *class method.*

metaclass incompatibility

A situation where a *subclass* does not include all of the *class variables* or respond to all of the *class methods* of its *ancestor classes*. This situation can easily arise in *OOP* systems that allow programmers to explicitly specify *metaclasses*, but is not allowed to occur in SOM. Instead, SOM automatically prevents this by creating and using *derived metaclasses* whenever necessary.

method   A combination of a *procedure* and a name, such that many different procedures can be associated with the same name. In object-oriented programming, invoking a method on an *object* causes the object to execute a specific *method procedure*. The process of determining which method procedure to execute when a method is invoked on an object is called *method resolution*. (The *CORBA* standard uses the term "operation" for method invocation). SOM supports two different kinds of methods: static methods and dynamic methods. See also *static method* and *dynamic method.*

method descriptor   See *descriptor.*

method ID   A number representing a zero-terminated string by which SOM uniquely represents a *method* name. See also *somId.*

method procedure

A function or procedure, written in an arbitrary programming language, that implements a *method* of a *class*. A method procedure is defined by the class implementor within the *implementation template* file generated by the *SOM Compiler.*

method prototype

A *method* declaration that includes the types of the arguments. Based on method definitions in an *IDL source file*, the *SOM Compiler* generates method prototypes in the *implementation template*. A class implementor uses the method prototype as a basis for writing the corresponding *method procedure* code. The method prototype also shows all arguments and their types that are required to invoke the method from a *client program*.

method resolution

The process of selecting a particular *method procedure*, given a *method* name and an object *instance*. The process results in selecting the particular function/ procedure that implements the abstract method in a way appropriate for the designated object. SOM supports a variety of method-resolution mechanisms, including *offset method resolution, name-lookup resolution,* and *dispatch-function resolution*.

method table   A table of pointers to the *method procedures* that implement the *methods* that an *object* supports. See also *method token.*

method token A value that identifies a specific *method* introduced by a *class*. A method token is used during *method resolution* to locate the *method procedure* that implements the identified method. The two basic method-resolution procedures are somResolve (which takes as arguments an *object* and a method token, and returns a pointer to a procedure that implements the identified method on the given object) and somClassResolve (which takes as arguments a *class* and a method token, and returns a pointer to a procedure that implements the identified method on an instance of the given class).

modifier Any of a set of statements that control how a *class*, an *attribute*, or a *method* will be implemented. Modifiers can be defined in the *implementation statement* of a SOM *IDL source file*. The implementation statement is a SOM-unique extension of the *CORBA* specification. [User-defined modifiers can also be specified for use by user-written emitters or to store information in the *Interface Repository*, which can then be accessed via methods provided by the *Interface Repository Framework*.]

module The organizational structure required within an *IDL source file* that contains *interface declarations* for two (or more) classes that are not a class–metaclass pair. Such *interfaces* must be grouped within a module declaration.

multiple inheritance
The situation in which a *class* is derived from (and inherits *interface* and *implementation* from) multiple parent classes.

name-lookup method resolution
Similar to the *method resolution* techniques employed by Objective-C and Smalltalk. It is significantly slower than *offset resolution*. Name-lookup resolution, unlike offset resolution, can be used when the name of the method to be invoked is not known until run time, or the method is added to the class interface at run time, or the name of the class introducing the method is not known until run time.

naming scope See *scope*.

object (Or *object instance* or just *instance*.) An entity that has *state* (its data values) and *behavior* (its *methods*). An object is one of the elements of data and function that programs create, manipulate, pass as arguments, and so forth. An object is a way to encapsulate state and behavior. *Encapsulation* permits many aspects of the *implementation* of an object to change without affecting client programs that depend on the object's behavior. In SOM, objects are created by other objects called *classes*.

object adapter (OA)
A *CORBA* term denoting the primary interface a *server implementation* uses to access *ORB* functions; in particular, it defines the mechanisms that a server uses to interact with DSOM, and vice versa. This includes server activation/deactivation, dispatching of *methods*, and authentication of the *principal* making a call. The basic object adapter described by CORBA is defined by the *BOA (basic object adapter) abstract class*; DSOM's primary object adapter implementation is provided by the *SOMOA* (*SOM Object Adapter*) *class*.

object definition See *class*.

object implementation See *implementation.*

object instance See *instance* and *object*.

object reference
A *CORBA* term denoting the information needed to reliably identify a particular *object*. This concept is implemented in DSOM with a *proxy object* in a *client* process, or a *SOMDObject* in a *server* process. See also *proxy object* and *SOMDObject.*

object request broker (ORB)   See *ORB*.

offset method resolution

The default mechanism for performing *method resolution* in SOM, because it is the fastest (nearly as fast as an ordinary procedure call). It is roughly equivalent to the C++ "virtual function" concept. Using offset method resolution requires that the name of the *method* to be invoked must be known at compile time, the name of the *class* that introduces the method must be known at compile time (although not necessarily by the programmer), and the method to be invoked must be a *static method*.

OIDL   The original language used for declaring SOM *classes*. The acronym stands for Object Interface Definition Language. OIDL is still supported by SOM release 2, but it does not include the ability to specify *multiple inheritance* classes.

one-copy serializable

The consistency property of the Replication Framework which states that the concurrent execution of *methods* on a *replicated object* is equivalent to the serial execution of those same methods on a nonreplicated object.

OOP   An acronym for "object-oriented programming."

operation   See *method.*

operation logging

In the Replication Framework, a technique for maintaining consistency among *replicas* of a replicated object, whereby the execution of a *method* that updates the *object* is repeated at the site of each replica.

ORB (object request broker)

A *CORBA* term designating the means by which *objects* transparently make requests (that is, invoke *methods*) and receive responses from objects, whether they are local or remote. With SOMobjects Developer Toolkit and Runtimes, this functionality is implemented in the DSOM Framework. Thus, the DSOM (Distributed SOM) system is an ORB. See also *BOA (basic object adapter) class* and *SOMOA (SOM object adapter) class.*

override   (Or *overriding method.*) The technique by which a *class* replaces (redefines) the *implementation* of a *method* that it inherits from one of its *parent classes*. An overriding method can elect to call the parent class's *method procedure* as part of its own implementation. (Such a call is known as a *parent method call.*)

parent class   A *class* from which another class inherits *instance methods*, *attributes*, and *instance variables*. A parent class is sometimes called a *base class* or *superclass*.

parent method call

A technique where an *overriding method* calls the *method procedure* of its *parent class* as part of its own *implementation*.

persistent object

An *object* whose *state* can be preserved beyond the termination of the *process* that created it. Typically, such objects are stored in files.

polymorphism

An object-oriented programming feature that may take on different meanings in different systems. Under various definitions of polymorphism, (a) a *method* or *procedure* call can be executed using arguments of a variety of types, or (b) the same variable can assume values of different types at different times, or (c) a method name can denote more than one *method procedure.* The SOM system reflects the third definition (for example, when a SOM class *overrides* a *parent class* definition of a method to change its behavior). The term literally means "having many forms."

principal   The user on whose behalf a particular (remote) *method* call is being performed.

procedure   A small section of code that executes a limited, well-understood task when called from another program. In SOM, a *method procedure* is often referred to as a procedure. See also *method procedure.*

process   A series of instructions (a program or part of a program) that a computer executes in a multitasking environment.

proxy object   In DSOM, a SOM *object* in the *client*'s address space that represents a remote *object.* The proxy object has the same *interface* as the remote object, but each *method* invoked on the proxy is *overridden* by a *dispatch method* that forwards the invocation request to the remote object. Under DSOM, a proxy object is created dynamically and automatically in the client whenever a remote method returns a pointer to an object that happens to be remote.

readers and writers

In the Replication Framework, different processes can access the same replicated object in different modes. A "reader" is a process that does not intend to update the object, but wants to continually watch the object as other processes update it. A "writer" is a process that wants to update the object, as well as continually watch the updates performed by others.

receiver   See *target object.*

redispatch stub

A *procedure*, corresponding to a particular *method*, which has the same *signature* as the method's procedure but which invokes somDispatch to dispatch the method. The somOverrideMtab method can be used to replace the procedure pointers in a *class*'s *method table* with the corresponding redispatch stubs. This is done when *overriding* somDispatch to customize *method resolution* so that all *static method* invocations will be routed through somDispatch for selection of an appropriate *method procedure*. (*Dynamic methods* have no entries in the method table, so they cannot be supported with redispatch functionality.)

reference data

Application-specific data that a *server* uses to identify or describe an *object* in DSOM. The data, represented by a sequence of up to 1024 bytes, is registered with DSOM when a server creates an *object reference.* A server can later ask DSOM to return the reference data associated with an object reference. See also *object reference.*

replica   When an object is replicated among a set of processes (using the Replication Framework), each process is said to have a replica of the object. From the viewpoint of any application model, the replicas together represent a single object.

replicated object

An *object* for which *replicas* (copies) exist. See *replica.*

run-time environment

The data structures, objects, and global variables that are created, maintained, and used by the functions, procedures, and methods in the SOM run-time library.

scope   (Or *naming scope.*) That portion of a program within which an identifier name has "visibility" and denotes a unique variable. In SOM, an *IDL source file* forms a scope. An identifier can only be defined once within a scope; identifiers can be redefined within a nested scope. In a .idl file, modules, interface statements, structures, unions, methods, and exceptions form nested scopes.

serializable   See *one-copy serializable.*

server   (Or *server implementation.*) In DSOM, a *process*, running in a distributed environment, that executes the *implementation* of an *object*. DSOM provides a default server implementation that can dynamically load SOM *class* libraries, create SOM objects, and make those objects accessible to *clients*. Developers can also write application-specific servers for use with DSOM.

server object   In DSOM, every *server* has an *object* that defines *methods* for managing objects in that server. These methods include object creation, object destruction, and maintaining mappings between *object references* and the objects they reference. A server object must be an *instance* of the *class* SOMDServer (or one of its *subclasses*). See also *object reference* and *SOMDObject.*

shadowing   In the Emitter Framework, a technique that is required when any of the *entry classes* are subclassed. Shadowing causes instances of the new subclass(es) (rather than instances of the original entry classes) to be used as input for building the object graph, without requiring a recompile of emitter framework code. Shadowing is accomplished by using the macro SOM_SubstituteClass.

signature   The collection of types associated with a *method* (the type of its return value, if any, as well as the number, order, and type of each of its arguments).

sister class object

A duplicate of a *class object* that is created in order to save a copy of the class's original *method table* before replacing the method table to customize *method resolution.* The sister class object is created so that some original *method procedures* can be called by the replacement method procedures.

Sockets class   A class that provides a common communications interface to Distributed SOM, the Replication Framework, and the Event Management Framework. The Sockets class provides the base interfaces (patterned after TCP/IP sockets); the *subclasses* TCPIPSockets, NBSockets, and IPXSockets provide actual implementations for TCP/IP, Netbios, and Netware IPX/SPX, respectively.

SOM Compiler

A tool provided by the SOM Toolkit that takes as input the interface definition file for a class (the .idl file) and produces a set of *binding files* that make it more convenient to implement and use SOM classes.

SOMClass   One of the three primitive *class objects* of the SOM run-time environment. SOMClass is the root (meta)class from which all subsequent *metaclasses* are derived. SOMClass defines the essential *behavior* common to all SOM *class objects.*

SOMClassMgr

One of the three primitive *class objects* of the SOM run-time environment. During SOM initialization, a single *instance* (*object*) of SOMClassMgr is created, called SOMClassMgrObject. This object maintains a directory of all SOM classes that exist within the current process, and it assists with dynamic loading and unloading of class libraries.

SOM-derived metaclass   See *derived metaclass.*

SOMDObject   The *class* that implements the notion of a *CORBA "object reference"* in DSOM. An *instance* of SOMDObject contains information about an object's *server implementation* and *interface,* as well as a user-supplied identifier.

somId   A pointer to a number that uniquely represents a zero-terminated string. Such pointers are declared as type somId. In SOM, somId's are used to represent *method* names, *class* names, and so forth.

SOMObject   One of the three primitive *class objects* of the SOM run-time environment. SOMObject is the root class for all SOM (sub)classes. SOMObject defines the essential *behavior* common to all SOM *objects.*

SOMOA (SOM object adapter) class

In DSOM, a *class* that dispatches *methods* on a *server*'s *objects*, using the *SOM Compiler* and run-time support. The SOMOA class implements methods defined in the *abstract* BOA class (its *base class*). See also *BOA class.*

somSelf   Within *method procedures* in the *implementation* file for a class, a parameter pointing to the *target object* that is an *instance* of the *class* being implemented. It is local to the *method procedure.*

somThis   Within *method procedures,* a local variable that points to a data structure containing the *instance variables* introduced by the *class*. If no instance variables are specified in the SOM *IDL source file*, then the somThis assignment statement is commented out by the *SOM Compiler*.

state (of an object)
The data (*attributes, instance variables* and their values) associated with an *object.* See also *behavior.*

static method   Any *method* that can be accessed through *offset method resolution*. Any method declared in the IDL specification of a class is a static method. See also *method* and *dynamic method.*

stub procedures
Method procedures in the *implementation template* generated by the *SOM Compiler*. They are procedures whose bodies are largely vacuous, to be filled in by the implementor.

subclass   A *class* that inherits *instance methods*, *attributes*, and *instance variables* directly from another class, called the *parent class, base class*, *superclass,* or indirectly from an *ancestor class*. A subclass may also be called a *child class* or *derived class.*

subclassing   The process whereby a new *class*, as it is created (or *derived*), inherits *instance methods, attributes,* and *instance variables* from one or more previously defined *ancestor classes.* The immediate *parent class(es)* of a new class must be specified in the class's *interface declaration.* See also *inheritance.*

superclass   See *parent class.*

symbol   In the Emitter Framework, any of a (standard or user-defined) set of names (such as, className) that are used as placeholders when building a text template to pattern the desired *emitter* output. When a template is emitted, the symbols are replaced with their corresponding values from the emitter's symbol table. Other symbols (such as, classSN) have values that are used by section-emitting methods to identify major sections of the template (which are correspondingly labeled as "classS" or by a user-defined name).

target object   (Or *receiver.*) The object responding to a *method* call. The target object is always the first formal parameter of a *method procedure*. For SOM's C-language bindings, the target object is the first argument provided to the method invocation macro, _*methodName*.

usage bindings
The language-specific *binding* files for a *class* that are generated by the *SOM Compiler* for inclusion in client programs using the class.

value logging   In the Replication Framework, a technique for maintaining consistency among *replicas* of a replicated object, whereby the new value of the object is distributed after the execution of a method that updates the object.

view–data paradigm
A Replication Framework construct similar to the Model-View-Controller paradigm in SmallTalk. The "view" object contains only presentation-specific information, while the "data" object contains the *state* of the application. The "view" and "data" are connected by means of an "observation" protocol that lets the "view" be notified whenever the "data" changes.

writers   See *readers and writers.*

Vos remarques sur ce document / Technical publication remark form

Titre / **Title :**   Bull DPX/20 SOMobject Base Toolkit User's Guide

Nº Reférence / **Reference Nº :**   86 A2 27AQ 01

Daté / **Dated :**   June 1995

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : _____   Date : _____

SOCIETE / COMPANY : _____

ADRESSE / ADDRESS : _____

_____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

BULL S.A. CEDOC
Atelier de Reproduction
FRAN–231
331 Avenue Patton BP 428
49 005 ANGERS CEDEX
FRANCE

BULL S.A. CEDOC
Atelier de Reproduction
FRAN–231
331 Avenue Patton BP 428
49 005 ANGERS CEDEX
FRANCE

ORDER REFERENCE
86 A2 27AQ 01

Bull

Use the cut marks to get the labels.

DPX/20

AIX

SOMobject Base
Toolkit
User's Guide

86 A2 27AQ 01

DPX/20

AIX

SOMobject Base
Toolkit
User's Guide

86 A2 27AQ 01

DPX/20

AIX

SOMobject Base
Toolkit
User's Guide

86 A2 27AQ 01