SOMobjects Developer's Toolkit

Programmer's Guide, Volume I: SOM and DSOM

SOMobjects Version 3.0

Note: Before using this information and the product it supports, be sure to read the general information under **Notices** on page iii.

Second Edition (December 1996)

This edition of *Programmer's Guide, Volume I: SOM and DSOM* applies to SOMobjects Developer's Toolkit for SOM Version 3.0 and to all subsequent releases of the product until otherwise indicated in new releases or technical newsletters.

To initiate changes to this publication, submit a problem report from the technical support web page at URL: http://www.austin.ibm.com/somservice/supform.html. Otherwise, address comments to IBM Corporation, Internal Zip 1002, 11400 Burnet Road, Austin, Texas 78758-3493. IBM Corporation may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing representative.

Notices

IBM Corporation may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate AIX, OS/2, or Windows programming techniques. You may copy and distribute these sample programs in any form without payment to IBM Corporation, for the purposes of developing, using, marketing, or distributing application programs conforming to the AIX, OS/2, or Windows application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "© (your company name) (current year), All Rights Reserved." However, the following copyright notice protects this documentation under the Copyright Laws of the United States and other countries which prohibit such actions as, but not limited to, copying, distributing, modifying, and making derivative works.

References in this publication to IBM products, program, or services do not imply that IBM Corporation intends to make these available in all countries in which it operates.

Any reference to IBM licensed programs, products, or services is not intended to state or imply that only IBM licensed programs, products, or services can be used. Any functionally-equivalent product, program or service that does not infringe upon any of the IBM Corporation intellectual property rights may be used instead of the IBM Corporation product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM Corporation, are the user's responsibility.

IBM Corporation may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries in writing to the:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594, USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department 931S
11400 Burnet Road
Austin, Texas 78758 USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Asia-Pacific users can inquire, in writing, to the:

IBM Director of Intellectual Property and Licensing
IBM World Trade Asia Corporation,
2-31 Roppongi 3-chome,
Minato-ku, Tokyo 106, Japan

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks and Acknowledgements

AIX is a trademark of International Business Machines Corporation.
FrameViewer is a trademark of Frame Technology.
IBM is a registered trademark of International Business Machines Corporation.
OS/2 is a trademark of International Business Machines Corporation.
SOM is a trademark of International Business Machines Corporation.
SOMobject is a trademark of International Business Machines Corporation.
Windows and Windows NT are trademarks of Microsoft **Corporation.**

Java and Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. IBM is independent of Sun Microsystems, Inc.

Contents

Figures

Chapter 1. Introduction to the SOMobjects Toolkit

This chapter provides a brief introduction to the SOMobjects Developer's Toolkit.

Introducing SOM and the SOMobjects Toolkit

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

- With SOM, you 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 (IDL).
- You then implement methods in your preferred programming language; an object-oriented programming language or a procedural language such as C.

Programmers can begin using SOM quickly. SOM extends the advantages of object-oriented programming to programmers who use non-object-oriented programming languages.

SOM accommodates changes in implementation details without breaking the binary interface to a class library or requiring recompiling client programs. If changes to a SOM class do not require source code changes in client programs, then those client programs do not need to be recompiled. SOM classes can undergo the following structural changes, 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
- Relocating methods upward in the class hierarchy

You can make the typical kinds of changes to an implementation and its interfaces that evolving software systems experience over time without starting over.

SOM is language-neutral. It preserves the key object-oriented programming characteristics of encapsulation, inheritance and polymorphism, without requiring that the user and the creator of a SOM class use the same programming language. SOM is said to be language-neutral for the following reasons:

- All SOM interactions consist of standard procedure calls. On systems with 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.
- The form of the SOM Application Programming Interface (API) can vary widely from language to language, due to SOM bindings. 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.
- 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 with differing object models. A SOM object may be accessed with the following 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 lets you 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: 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 a high degree of encapsulation for the implementation of an object, with some cost in performance.

- SOM conforms with the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standards:
  - Interfaces to SOM classes are described in CORBA's Interface Definition Language, IDL. 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 does not replace existing object-oriented languages; it complements 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 Developer Toolkit contains the SOM Compiler to 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 programmers
- 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 example, C programs can invoke methods in the same way they make ordinary procedure calls. The C++ bindings *wrap* SOM objects as C++ objects, so that C++ programs 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. The SOM Compiler can generate both C and C++ language bindings for a class. The C and C++ bindings work with a variety of commercial products available from IBM and others. Vendors of other programming languages may also offer SOM bindings.

The SOM Run-Time Library

The SOM run-time library provides, among other things, a set of classes, methods and procedures used to create objects and invoke methods. The library allows any programming language to use classes developed using SOM if that language can:

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

The user and the creator 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 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 classes. Moreover, SOM's run-time environment allows applications to access information about classes dynamically.

Frameworks provided in the SOMobjects Toolkit

In addition to 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 Emitter Framework and the Metaclass Framework.

Distributed SOM

Distributed SOM (DSOM) lets application programs access SOM objects across address spaces. Application programs can access objects in other processes, even on different machines. DSOM provides this transparent access to remote objects through its Object Request Broker (ORB): the location and implementation of the object are hidden from the client, and the client accesses the object as if were local. DSOM supports distribution of objects among processes within a workstation, and across a local area network consisting of AIX, OS/2, or Windows NT systems, or some mix of these systems.

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 **TypeCodes**, a CORBA-defined way of encoding the complete description of any data type that can be constructed in IDL.

Emitter Framework

The Emitter Framework is a collection of SOM classes that you write as your own emitters. Emitter is a general term used to describe a back-end output component of the SOM Compiler. Each emitter takes 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). You can write your own special-purpose emitters. For example, you can write an emitter to produce documentation files or binding files for programming languages other than C or C++.

Metaclass Framework

The Metaclass Framework is a collection of SOM metaclasses that provide functions used by SOM class designers to modify the default semantics of method invocation and object creation. These metaclasses are described in **Chapter 10, The Metaclass Framework** on page 357.

What's New And Changed in SOMobjects Version 3.0

SOMobjects Version 3.0 offers many enhancements, changes, and additions to Version 2.1. In addition to the changes listed, many editorial changes have been made to the documentation.

General Enhancements

This section describes new and changed functions that affect all users.

SOMobjects on the World Wide Web

You can find the latest on IBM's Enterprise Object Technology and Application Development Solutions, including SOMobjects, on the World Wide Web. Look at:

http://www.software.ibm.com/clubopendoc

for the latest information.

Configuration Information

All the configuration information for SOM and DSOM has been reorganized into a separate chapter.

The **SOMENV** environment variable points to a configuration file containing many settings needed by the SOMobjects frameworks, components and DSOM communications protocols. See “Configuration File Stanzas” on page 16 for additional information relating to information contained in the configuration file.

SOMobjects Enhancements

- The **imod** emitter produces a C source program that implements a class library’s initialization and termination function. Compile and link this source file with all the class implementation source files. See “Specifying the Initialization and Termination Function” on page 215 for additional information.
- The **mods** emitter creates a file with a list of modifiers specified in an IDL file. Utilities can use this file to determine the values of the modifiers.
- New modifiers have been defined for use in the implementation sections of SOM IDL files, and some existing modifiers have been enhanced. A majority of the new modifiers support the DSOM enhancements of Version 3.0. See “Modifier Statements” on page 133 for a description of each modifier:
  - Unqualified modifiers:

    classinit (enhanced)
    factory
    nomplans
  - Qualified modifiers:

    dual_owned_parameters
    dual_owned_result
    impctx (enhanced)
    impldef_prompts
    length
    maybe_by_value
    maybe_by_value_result
    mplan
    object_owns_parameters (enhanced)
    object_owns_result (enhanced)
    pass_by_copy
    pass_by_copy_result
    pointer
    staticdata (enhanced)
    struct
    suppress_inout_free
- The SOM Compiler’s **-m** flag that sets global modifiers has new options to control the actions of the **ir**,**exp**, **def**, and **imod** emitters.
- For situations where a method needs to modify an **in** parameter, the method can receive a copy of the parameter. The IDL modifier **pass_by_copy** is used to identify parameters that should be copied when passed from the caller of a method to the method’s implementation. See “Passing Parameters by Copying” on page 146. The new modifiers listed include the related **pass_by_copy** and **pass_by_copy_result**.
- SOMobjects 3.0 provides the ability to cast objects. See the SOM kernel methods **somCastObj** and **somResetObj**.
- The Service Provider Interface (SPI) interface for thread or semaphore-related functions is no longer available.

- The Error Log facility records exceptions and error conditions that occur within many of the SOMobjects services. DSOM and Object Services use this facility. You can use the Error Log to help debug your new applications.

DSOM Enhancements

- The Implementation Repository no longer needs to be replicated or shared between client and server processes. Instead, the Implementation Repository is used only on the machine where the server runs. DSOM clients obtain information about DSOM servers through the distributed Naming Service, rather than from the Implementation Repository. See “Migration Relationship to the 2.x Implementation Repository” on page 42 for how the Implementation Repository in this release differs from DSOM 2.x.
- DSOM includes a migration tool, **migimpl3**, for converting DSOM 2.1 Implementation Repositories into the DSOM 3.x format. See “Differences between 2.x and 3.x” on page 42.
- An Implementation Repository can now be extended through user-defined subclasses of the **ImplementationDef** class. A single Implementation Repository can support a heterogeneous mix of different types of **ImplementationDef** objects. See “Customizing ImplementationDef Objects” on page 41.
- New interfaces are provided to the **regimpl** utility for registering servers, to support the new features of the Implementation Repository. See “The regimpl Registration Utility” on page 32.
- Improved local and remote transparency is supported. Many methods that previously could be invoked only on or with a proxy object can now be invoked on or with a local **SOMObject**. This includes DII support for local objects, **object_to_string** and **string_to_object** methods, and most methods of **SOMDObject** class. See “Dynamic Invocation Interface” on page 311 as well as the specific methods and classes.
- There is a facility for finding “factory” objects (objects that create other objects) using the Naming Service. This facility supersedes previous **SOMDObjectMgr** methods and **SOMDServer** methods for creating remote objects. The same interfaces can be used to create both local and remote objects. See “Creating Remote Objects” on page 245 for more information.
- DSOM includes two new initialization functions, **list_initial_services** and **resolve_initial_references**, that initialize certain object services (for example, the Interface Repository, Naming Service, and Factory Service). For additional information, see “Finding Initial Object References” on page 244 as well as reference for the methods themselves.
- Support for communication through multiple protocols. On OS/2, TCP/IP and NetBIOS are supported through AnyNet. TCP/IP can also be used alone, without AnyNet. On AIX and Windows NT, the only supported protocol is TCP/IP. See the **SOMDPROTOCOLS** environment setting.
- Support for compiling information about method signatures (marshal plans) into the class library (DLL), so that Interface Repository accesses can be reduced or eliminated. This is done automatically by the current **ih** and **xih** emitters. For information on when the Interface Repository is required by DSOM, see “Registering Class Interfaces” on page 30.
- Support for marshalling pointer types within larger data structures. See “Making Remote Method Calls” on page 250.

- Support for marshalling non-CORBA IDL (foreign) data types. See “Passing Foreign Data Types” on page 262.
- Support for passing a copy of an object parameter from a client to a server, to provide improved performance. See “Passing Objects by Copying” on page 261.
- A server’s object reference table files are now stored in the directory indicated by the **SOMDDIR** setting, rather than as specified in the Implementation Repository. The same files can be used by multiple servers.
- Support for Java applications and applets as DSOM clients. This allows DSOM server objects to be accessed from any Java-enabled client or web browser, without requiring the installation of any additional software on the client machine. The Java client ORB is written entirely in Java, which means that it runs unmodified on all Java-enabled platforms.
- Support for Internet Inter-ORB Protocol 1.0, the CORBA standard for message protocol. This allows DSOM to interoperate with other CORBA-compliant ORBs.

Metaclass Framework Enhancements

The enhancements facilitate the making of a proxy for in-memory objects. The facility consists of the metaclass **SOMMProxyFor** and helper class, **SOMMProxyForObject**. The helper class may be subclassed to produce special kinds of proxies (as DSOM does). Proxy classes can be composed with Before/After Metaclasses.

Object Services

In general, the object services are provided as a combination of concrete and mix-in classes. Mix-in classes provide the needed behavior to manage your objects. The concrete classes instantiate distinguished objects used in support of managing your objects.

Externalization Service

The externalization service provides the basis by which an object can externalize or internalize its state. By inheriting this mix-in class, an object class programmer can enable their object’s state to be managed without breaking the encapsulation boundary of their object. Externalization is used by the DSOM pass-by-value facility.

Naming Service

The naming service provides a Naming Context. Instances of naming contexts can be organized into a name hierarchy. You can assign a name to your objects within a particular context. During the configuration process, SOMobjects will build a default global name tree and bind certain distinguished objects within that name tree.

Object Identity Service

The object identity service lets you determine whether two objects are in fact the same object or different. Due to the nature of encapsulation, it is not possible to tell whether two objects are the same or different merely by examining their object pointers or object references. By mixing the object identity service, your object is assigned an identity that can be used by the service to unambiguously determine whether two objects are identical.

Security Service

The security service can be used to authenticate clients to secure servers. This is useful for ensuring that client principals are indeed who they say they are, allowing the principal's identity (which can be defined from the **Principal** object) to be used in access decisions.

Migration Considerations

When migrating to this version from earlier versions, the following changes may require changes to your applications.

The SOM 3.0 kernel is much more sensitive to memory-management bugs than the SOM 2.1 kernel. For example, if you try to perform a **SOMFree** on the same pointer twice, or try to perform a **SOMFree** on stack-based storage (rather than storage allocated by **SOMMalloc**), a trap will occur. In the SOM 2.1 Kernel, such an act was often harmless or caused corruption that may not have been apparent. If new traps occur from using your code with the SOM 3.0 kernel, a memory-management bug is the likely cause of the traps.

The enumeration types exception_type and completion_status, defined in **somcorba.h**, are now four bytes long instead of one byte long in order to comply with the CORBA specification. This also aids in using these enumerations across DSOM. If your application uses variables of these enumeration types, recompile it using the SOM header files in this release.

The semantics of the **somFree** method when invoked on a proxy object has been changed so that both the remote object and the local proxy object are destroyed. See “Destroying Remote Objects” on page 265.

When a method invoked remotely raises a USER_EXCEPTION that is not declared in the IDL for the invoked method, DSOM returns a SYSTEM_EXCEPTION to the client with minor code **SOMDERROR_UndeclaredException**.

DSOM's default parameter-memory-management policy now fully adheres to the CORBA 1.1 specification. Parameter memory is now treated as uniformly caller-owned by the DSOM runtime in both the client and server address space. Different behavior can be requested through IDL modifiers. See “Memory Allocation and Ownership” on page 252.

Overview of the Programmer's Guide

Chapter 2, Configuration and Startup on page 11 contains information relating to SOM and DSOM configuration.

Chapter 3, Tutorial for Implementing SOM Classes on page 49 is a Tutorial with examples that illustrate techniques for implementing classes in SOM. Study this chapter if you are new to SOM.

Chapter 4, Using SOM Classes in Client Programs on page 69 describes how an application program creates instances of a SOM class, how it invokes methods, and so on. Read this chapter if you plan to create or use SOM classes.

Chapter 5, SOM Interface Definition Language on page 115 describes SOM IDL syntax. Read this chapter if you plan to create SOM classes.

Chapter 6, The SOM Compiler on page 155 describes the SOM compiler. Read this chapter if you plan to create SOM classes.

Chapter 7, Implementing Classes in SOM on page 171 provides more comprehensive information about the SOM system itself, including operation of the SOM run-time environment, inheritance, and method resolution. This chapter also describes how to create language-neutral class libraries using SOM. It includes advanced topics for customizing SOM to better suit the needs of a particular application. Read this chapter if you plan to create SOM classes.

Chapter 8, Distributed SOM on page 229 describes DSOM and how to use it and customize it to access objects across address spaces, even on different machines.

Chapter 9, The Interface Repository Framework on page 337 describes the Interface Repository Framework of classes supplied with the SOMobjects Toolkit.

Chapter 10, The Metaclass Framework on page 357 describes the Metaclass Framework and some utility metaclasses that SOM provides to help you derive new classes with special abilities to execute before and after operations when a method call occurs. It also tells how to modify the default semantics of method invocation and object creation.

Appendix A, Error Codes on page 397 contains lists of the error codes that can be issued by the SOM kernel or by the various frameworks.

Appendix B, Converting OIDL Files to IDL on page 417 tells how to convert class definition files from OIDL syntax (the interface definition language used by a previous release of SOM) to IDL syntax (the language prescribed by the CORBA standard).

Appendix C, SOM IDL Language Grammar on page 421 contains the SOM IDL language grammar.

Chapter 2. Configuration and Startup

This chapter describes the steps between installing the SOMobjects Developer's Toolkit software and running your applications. These steps include:

- Configuration
- Verifying the configuration
- Registering applications and servers
- Running sample programs

A Quick Guide to Configuration

The configuration information in this chapter covers a broad spectrum of needs, starting from casual one-time SOMobjects use to the most advanced use. Identify your case in the following list to get the information you need to configure your system efficiently.

Case 1
You want to use just SOM and do not want to use DSOM or Object Services.

- Do **Step 1. Installation and Operating System Environment Variables**
- Do **Step 2. Generate Header Files**.
- Skip the rest of this chapter. No further configuration is needed to use SOM.

Case 2
You want to run, on a single machine, just the DSOM sample programs supplied in the Toolkit.

- Do the installation and configuration, **Step 1. Installation and Operating System Environment Variables** on page 13 through **Step 2. Generate Header Files**.
- If you are reconfiguring your system, first do the steps described in **Reconfiguring DSOM** on page 28.
- Do the minimal configuration for running the sample applications, Step 5. In **Step 3. Customize Settings in the Configuration File** the minimum required is:
  - Decide if you want to modify the default **somenv.ini** file supplied with the Toolkit or instead modify a copy of it. If you modify a copy, make sure that you set the **SOMENV** environment variable to point to your copy.
- Do **Step 4. Issue somdchk**.
- In **Step 5. Issuing the Configuration Command** on page 24, issue the **som_cfg -i** command and make sure it completes without error. This is all you need do of this step.

Case 3
You want to build and run your own simple DSOM application on a single machine.

- Review DSOM concepts and the **DSOM Tutorial** on page 233.
- Read the information about client and server programming and then build your application.
- Read about other dependencies of the Object Services your application may use.
- Do the minimal configuration described in **Case 2**.
- **Register User Applications**.

Case 4
You want to build and run your own DSOM application on two or more machines.

- Read the configuration steps in this chapter paying special attention to the following:
  - Learn about configuring one of the machines as an Install host and the others as DSOM hosts.
  - Learn about the prerequisites for re-running **som_cfg**.
  - Read about **SOMDPROTOCOLS** and **HOSTNAME**.
  - Do the steps in **Case 3**, but do all the configuration described in **Step 5. Issuing the Configuration Command**.
- **Register User Applications**.

Case 5
You are an experienced DSOM user and want to customize the DSOM environment.

- Read all of this chapter and exploit the various options provided.

Configuring and Customizing a New Installation

This section first gives a quick list of the steps between installing and using SOM, then describes each of the steps in detail. Configuration takes many steps, but each of them is simple and you do most of them only once. The following sections give the steps for configuring a new installation and for updating your configuration.

Configuration Steps

1. Complete the installation steps as described in the product **README** file.
2. Generate header files.
3. Customize settings in the configuration file. (This step is optional for single workstation environments.)
4. Invoke **somdchk** to verify the environment settings. (This step is optional for all environments.)
5. Issue the configuration command. All machines in a multi-machine system must be configured.
6. Register your user applications.

Register User Applications

Configuration is typically a one-time process. However, you register each application before you use it. Thus, you may repeat these steps as you continue to develop applications.

1. Customize settings in the configuration file. Each application might have different settings, so this step can be redone as needed.
2. Run **somdchk**. This step is optional.
3. Register servers and classes in the Implementation Repository using **regimpl**.
4. Migrate any 2.1 Implementation Repositories to the current level. See **Migrating 2.x Implementation Repositories to Current DSOM Format** on page 44.

Also, see **Running DSOM Applications** on page 308.

This sequence can be done as part of your configuration sequence if you have existing applications. Or, you can repeat some or all of this step as often as you need once you have created applications.

Running the Sample Programs as an Installation Test

The SOMobjects Developer Toolkit includes a set of sample programs that you can use both to learn more about using SOM and to verify your installation. The following steps tell how to use the sample programs.

1. Change directory to a sample directory.
2. Read the **README** files associated with samples in general, and with the specific sample you plan to run.
3. Set SOMIR to include `.\som.ir` as the rightmost file, which is required by the samples.
4. Start the DSOM daemon, **somdd**, from the sample directory, required for building and running the sample. If **somdd** is already running, terminate it and restart it from the sample directory so that **somdd** can find and start the sample server. (This is not generally required for real application servers, which can be found using PATH.)
5. Build the sample as described in its **README** file.
6. Run the sample application as described in its **README** file.

Step 1. Installation and Operating System Environment Variables

This process is described in detail in the product **README** file.

Step 2. Generate Header Files

Ensure that the `$SOMBASE/include` directory (for AIX) or the `%SOMBASE%\include` directory (for OS/2 and Windows NT) exists and is writable. If you already have **.h**, **.xh**, or **.bld** files there, make sure they are writable.

Select the type of header files appropriate for your development environment.

C++
Issue the **somxh** command to generate the **.xh** files for the classes supplied with SOMobjects Developer Toolkit.

If you plan to program with SOM using the C bindings, you need to select either the strict CORBA-compliant form in which asterisks (*) are not exposed in object references or the C++-friendly form in which asterisks are visible in object references. The latter, C++-friendly form is more appropriate if you plan to later move your class implementations from C to C++. This choice determines how references appear in your C programs. For example, to declare a reference to an instance of class `Foo` you could code:

```
Foo afoo     /*Strict CORBA-compliant form */
```

or

```
Foo *afoo  /* C++ migration or C++-friendly form */
```

CORBA-compliant C
Issue the **somcorba** command to generate the header files if asterisks (*) are not exposed in object references.

C++-like C
Issue the **somstars** command to generate the headers for if asterisks are exposed in object references.

All the sample C programs provided with SOMobjects Developer Toolkit assume the CORBA-compliant coding style and the **somcorba** command.

If you use **somstars**, you should also set the SMADDSTAR variable in your local environment to 1. All subsequent use of the SOM compiler depends on the proper setting of this environment variable.

If you later switch from one coding style to another, you must convert any C code to that other style. If you switch from **somstars** style to **somcorba** style, you must remove the SMADDSTAR variable from your environment.

If you install only some of the SOMobjects Developer Toolkit components, and later add components, you must repeat this step after each component installation.

Step 3. Customize Settings in the Configuration File

SOMobjects looks for run-time environment settings in a configuration file designated by the **SOMENV** environment setting. Compile-time settings are taken from your operating system environment. You can edit the configuration file to specify default settings for your installation, or you can change **SOMENV** to specify another file with alternate settings. The **SOMENV** environment variable can contain one or more path names speared by colons on AIX or by semicolons on OS/2 and Windows NT. The default setting of **SOMENV** is `SOMENV/etc/somenv.ini on AIX and %SOMBASE%\etc\somenv.ini` on OS/2 and Windows NT.

The default configuration file defines many settings required by the different frameworks, including settings for supported communications protocols. You can define your own configuration settings. This section describes each of the default settings, tells which can be changed, and, typically, gives suggestions for changing values.

SOMobjects provides three functions relating to the configuration file, **somutgetenv**, **somutgetshellenv** and **somutresetenv,** that you can use within your applications. See **Checking Configuration Values** on page 45 for additional information on these three functions.

Is Customization Required?

To configure a single-machine installation, no configuration of the default configuration file is required.

The **SOMIR** and **SOMDDIR** settings are, perhaps, the most common customizations, but are not required. You can set **SOMIR** as an operating system environment variable. If set, the environment variable takes precedence over the configuration file setting. For this reason, a **SOMIR** setting in the configuration file is optional. See **SOMIR** and **SOMDDIR**.

If you are configuring a single-machine installation and do not plan to customize any environment settings in the configuration file, skip to **Step 5. Issuing the Configuration Command**.

If you are configuring a multiple-machine installation, you must set the following in your configuration file:

> **SOMDPROTOCOLS**
> **HOSTNAME** (one setting for each protocol named in **SOMDPROTOCOLS**)

Decide whether you want to modify the default configuration file or modify a copy of it. If you modify a copy, set the **SOMENV** environment variable to designate the new copy. The configuration file can be edited with any ASCII text editor. After you modify the required settings, if you do not plan to customize other environment settings in the configuration file, skip to the next step.

The Configuration File

Configuration File Stanzas on page 16 shows each stanza of the configuration file, then describes the settings you can accept or change. The descriptions tell you which settings should be left unchanged. In most cases, comments in the file itself give you condensed instructions.

Syntax of the Configuration File

Follow these syntax rules when you edit the configuration file:

- Each stanza contains the settings for a component of SOM. The name of the stanza is enclosed in square brackets ([ ]) at the beginning of a line. Do not edit the stanza names.
- Comment lines begin with a semicolon (;) in the first position of the line.
- Blanks and spaces can appear anywhere in the file.
- Some settings are commented out in the sample file; delete the semicolon to set the value.
- Some values should not be edited. Those values are noted in the file.
- Most settings are of the form `item=value.`
- The actual file may differ slightly in format from this explanation; the items and values are the same.

Processing the Configuration Files

If a stanza in a configuration file contains multiple entries for the same identifier name, the value that takes effect is the *first* setting for that identifier. For example, in the following code, `namex` is set to 10:

```
[foo]
    namex = 10
    ...
    namex = 20
```

If **SOMENV** defines two or more path names such that multiple configuration files contain the same stanza and identifier names, but set different values, then the value from the *leftmost* file in the path takes effect. For example, on OS/2, given the definition `SOMENV=C:\SOM\ETC\SOMENV.INI;C:\MYAPP\somenv.ini`, where `namey` in stanza `foo` is defined as shown, then `namey` is set to 30, as defined in the leftmost file:

```
[foo] in C:\SOM\ETC\SOMENV.INI
    namey = 30
    ...
[foo] in C:\MYAPP\SOMENV.INI
    namey = 15
```

The SOMobjects configuration file name/value pairs are read into memory on the first call to the **somutgetenv** function by a SOMobjects process. Typically, the call to **somutgetenv** is done by one of the SOMobjects frameworks. Unless a SOMobjects application wants to get the value of a specific setting, there is no need for an application program to call this function.

The configuration file is not read again unless, prior to calling **somutgetenv**, the process calls **somutresetenv**. Therefore, if you change any settings in the configuration file, you may need to either refresh or restart your SOM application before the new setting is used.

There is a separate instance of the configuration-file settings in memory for each user process running a SOM framework. This means that if the configuration file settings are changed, only new user processes of a SOM framework pick up the changes. SOM framework processes running before the configuration change continue to run with the older settings until they are restarted or refreshed by calling **somutresetenv** followed by a call to **somutgetenv**. See **Checking Configuration Values** on page 45 for additional information on **somutgetenv** and **somutresetenv**.

Configuration File Stanzas

This section describes each stanza of the configuration file. You can read it as you edit your configuration file.

If your system is a single machine, you might not have to edit the configuration file. A multiple-system installation might need only to specify communications protocols.

Error Log Facility

The [somras] stanza contains the settings for the Error Log Facility. You can control the size of the error log, what errors get stored, the name of the file, and whether error information is displayed on the screen. See **The Error Log Facility** on page 107 for more information.

```
[somras]
; RAS configuration stanza. This controls the Error Log Facility.
;
   SOMErrorLogFile=SOMERROR.LOG
; The name of the file where error log entries will be
; stored. If unset, the default is SOMErrorLogFile=SOMERROR.LOG.
; The error log is always
; placed in the directory pointed to by the SOMDDIR
; configuration variable of the [somd] stanza. We recommend that
; all processes on a system share one error log file.
;
   SOMErrorLogSize=128
; The size, in kilobytes, of the error log file. If unset, the
; default is SOMErrorLogSize=128. The default allows space for
; several hundred average sized log entries.
;
   SOMErrorLogControl=WARNING ERROR MAPPED_EXCEPTION
; A filter to control what types of log entries will be included
; in the error log file. Multiple values may be specified,
; delimited by spaces. Valid values include INFO, WARNING,
; ERROR, and MAPPED_EXCEPTION. If unset, the default is
; SOMErrorLogControl=WARNING ERROR MAPPED_EXCEPTION.
;
   SOMErrorLogDisplayMsgs=YES
; In addition to making a log entry also display each formatted
; error log message, without the extended log data, to the standard
; output device. Valid values are NO or YES. If unset, the
; default is SOMErrorLogDisplayMsgs=YES.
```

Security Service

The [somsec] stanza contains the settings for security. You need to consider only the setting of the `LOGIN_INFO_SOURCE` value which affects DSOM remote method calls.

If using the SOMobjects Security Service, users must log in to run as an authenticated client. Users not logged in run as unauthenticated clients; their requests are rejected by any server registered as a secure server.

How you login will depend somewhat on the platform you are logging in from and your preferences.On OS/2, you can login with the OS/2 User Profile Manager (UPM). On AIX, you can be prompted for your user name and password. On Windows NT, or as an alternative to OS/2 or AIX, you can supply your user name and password in environment variables.

If you want to login using UPM on OS/2, set the following:

```
LOGIN_INFO_SOURCE=UPM
```

If you want to be prompted for your user name and password on AIX, set the following:

```
LOGIN_INFO_SOURCE=PROMPT
```

If you want to supply your user name and password in environment variables, set the following:

```
LOGIN_INFO_SOURCE=ENV
```

Setting `LOGIN_SOURCE_INFO=DEFAULT` is equivalent to setting `LOGIN_SOURCE_INFO` to `UPM` on OS/2 and `ENV` on AIX or Windows NT. If you want to run as an unauthenticated client, leave the value for `LOGIN_SOURCE_INFO` blank, as follows:

```
LOGIN_INFO_SOURCE=
```

The default for `REGISTRY DB DIR` is **SOMDDIR**.

```
[somsec]
;
; SECURITY_SERVER ALIAS is the name used by som_cfg to register the
; security server implementation. If unset, the default is
; securityServer
;    SECURITY_SERVER_ALIAS
;
; LOGIN_INFO_SOURCE is a list of sources for obtaining login
; information of a user for authentication. The possibilities are
; LOGIN_INFO_SOURCE=DEFAULT ENV UPM PROMPT
; The options may be specified in any order and any combination.
; The first option to yield the required login information is used
; and subsequent ones ignored. Unrecognized options are ignored.
; DEFAULT is equivalent to UPM on OS2 and ENV on AIX and other
; platforms.
; LOGIN_INFO_SOURCE=
; turns off authentication. Beware that a secure server may reject
; requests from unauthenticated clients.
;
    LOGIN_INFO_SOURCE=DEFAULT
;
; LOGIN_TIMEOUT specifies the duration (in seconds) after which
; a quest for login information will timeout.
    LOGIN_TIMEOUT=30
;
```

Naming Service

The [somnm] stanza contains Naming Service settings and specifies whether the machine is an Install host or DSOM host for the Naming and Security Services. The **HOSTKIND** setting is inserted into the configuration file by **som_cfg**. Do not set or change **HOSTKIND**. If the system needs to be reconfigured, however, by re-running **som_cfg**, remove **HOSTKIND** from the configuration file before rerunning **som_cfg**.

```
[somnm]
;
   HOSTKIND=DSOM
; Name Service environment configuration stanza.
; Uses SOMDDIR setting specified in the [somd] stanza to
; store Naming related files.
; If unset, the default is %SOMBASE%\etc\dsom.
; For the install host, the setting is HOSTKIND=INSTALL
```

SOMNMOBJREF in the [somnm] stanza, is set by the **som_cfg** utility. The setting is used by DSOM for connecting a client application to the Naming Service. It should not be changed by the user. If, however, the system needs to be reconfigured, by re-running **som_cfg**, this setting must be removed from the configuration file before rerunning **som_cfg**.

```
; NAMING_SERVER_ALIAS=
; NAMING_SERVER_ALIAS is the name used by som_cfg to register the
; naming server implementation. If unset, the default is
; namingServer.
;
; GLOBAL_OBJREF_FILE=
; GLOBAL_OBJREF_FILE is the file used by som_cfg to hold
; the global object reference. In the case of an install host, the
; object reference is written to this file. In the case of a
; DSOM host, the object reference is read from this file.
; If unset, the default places the file SOMNM.REF in the
; directory pointed to by the SOMDDIR configuration variable of
; the [somd] stanza.
; For the DSOM host, this value must be set to a directory other
; than the default.
```

Interface Repository

The **SOMIR** specifies a list of files, separated by colons on AIX and semicolons on OS/2 and Windows NT. The **SOMIR** setting can be specified either using the **SOMIR** environment variable or by setting **SOMIR** in the configuration file. The default is **.\som.ir**. See **Registering Class Interfaces** on page 30 and **Chapter 9, The Interface Repository Framework** on page 337 for more information on how to set **SOMIR** and create an IR. For DSOM, it is preferable to use full path names in the list of IR files, because the IR will be shared by several programs that may not all be started in the same directory.

```
  SOMIR=
;      The location of the Interface Repository.
```

SOM Utilities and Metaclass Framework

The [somu] stanza specifies settings for the SOM utilities and the metaclass framework.

```
;[somu]
;      SOM utilities and metaclass framework
;  SOMM_TRACED=
```

DSOM Configuration

The [somd] stanza and the following communications stanzas contain settings for DSOM and for communication between systems. When you edit the **SOMDPROTOCOLS** setting of this stanza to specify a communications protocol, make sure to edit any subsequent stanzas required for that protocol.

```
[somd]
;       DSOM environment configuration stanza.
```

SOMDDIR

specifies the directory where various DSOM-related files are stored, including the Implementation Repository files. See **Registering Servers and Classes** on page 31 for more information.

Note: If this value is not set, DSOM attempts to use a default directory:either $SOMBASE/etc/dsom on AIX or `%SOMBASE%\ETC\DSOM` on OS/2 and Windows NT.

Because the configuration file uses the backslash character as a line-continuation character, do not end the **SOMDDIR** setting with a backslash.

```
;  SOMDDIR
;       The location of the Implementation Repository and other
;       DSOM-related files.
;       If unset, the default is %SOMBASE%\etc\dsom.
```

The directory named by the **SOMDDIR** setting must exist, be empty, and be writable.

HOSTNAME

specifies the name by which the machine is known (used to set the **hostName** attribute of the **Principal** object, which identifies the caller of a method in a server). **HOSTNAME** is also used by the Factory Service to identify the requester of a local (same-process) factory, and by **regimpl** when registering classes for _LOCAL creation. If the **HOSTNAME** environment variable is set, that setting supersedes the **HOSTNAME** setting in the [somd] stanza of the configuration file.

There are separate **HOSTNAME** settings within the stanzas corresponding to the individual protocols named by **SOMDPROTOCOLS**; these are unaffected by the **HOSTNAME** environment variable setting.

```
   HOSTNAME=thehostname
;  USER=anyuser
;       Used to set the Principal object that represents the client.
```

SOMDPROTOCOLS

should be set to specify the names of the DSOM communications protocols for which the machine has been configured, separated by spaces. Valid values include: SOMD_IPC and SOMD_TCPIP on AIX and NT; and SOM_IPC, SOMD_TCPIP, and SOMD_NetBIOS on OS/2. (Additional protocols may be provided by other vendors.) The default value is SOMD_IPC only. When the SOMD_IPC (single-machine) protocol is used, it should be the first name in the list of protocols specified for **SOMDPROTOCOLS.**

The setting of **SOMDPROTOCOLS** for a client and server process must have at least one entry in common; otherwise, communication between the two processes is not possible.

On OS/2, when AnyNet is running, DSOM processes on different machines can communicate over any combination of TCP/IP or NetBIOS (provided they are also running), as designated by the setting of **SOMDPROTOCOLS**. When AnyNet is not running, the **SOMDPROTOCOLS** setting SOMD_NetBIOS is invalid. (**SOMDPROTOCOLS** should include SOMD_TCPIP in this case.)

SOMDRECVWAIT

specifies the number of seconds a receiver should wait for a message to complete transmission before generating a communications time-out error. The default value is 30 seconds. This setting is meaningful only for workgroup applications.

```
. SOMDRECVWAIT=30
; The number of seconds to wait for a socket to become readable
; before generating a communications timeout error.
```

Note: This setting replaces the 2.1 environment variable **SOMDTIMEOUT**. For backward compatibility, if **SOMDRECVWAIT** is not set, the value of **SOMDTIMEOUT** will be used. Unlike **SOMDTIMEOUT**, however, **SOMDRECVWAIT** does not include the time required for the remote method to execute.

SOMDSENDWAIT
SOMDSENDWAIT specifies the number of seconds a sender should try to send a message before generating a communications time-out error. The default value is 30 seconds. This setting is meaningful only for workgroup applications.

```
  SOMDSENDWAIT=30
; The number of seconds to wait for a socket to become writable
; before generating a communications timeout error.
```

SOMDNUMTHREADS
may optionally be set to the maximum number of request threads created per server. If **SOMDNUMTHREADS** is not set, then a separate thread is created for each request, if the server is registered as multithreaded. **SOMDNUMTHREADS** is relevant only for DSOM server processes.

```
;  SOMDNUMTHREADS=
;     The maximum number of threads a multithreaded server will use.
;        The default is unlimited.
```

SOMDTHREADSTACKSIZE
may optionally be set to increase the stack size used when creating new threads within a multithreaded DSOM server process. The default is 65536 (bytes). This setting is only relevant for DSOM server processes.

```
;  SOMDTHREADSTACKSIZE=
;    The stack size used when creating new threads in a server, in
;       bytes.
;       The default is 65536.
```

CHECK_CONNECTION_INTERVAL
may optionally be set to determine how often, in minutes, DSOM should check for broken IPC connection.

```
;CHECK_CONNECTION_INTERVAL=5
```

DSOM IPC

The [SOMD_IPC] stanza, in conjunction with the [somd] stanza contains settings for the IPC protocol.

```
[SOMD_IPC]
; Stanza for the DSOM IPC protocol for workstation communication
; (UNO IIOP).
```

HOSTNAME
specifies the name by which the machine is known, for that protocol. For the IPC transport, this setting must be the same for both the client and the server.

```
  HOSTNAME=thehostname
;       Set this value to the name by which this machine is known.
```

SOMDPORT

specifies the well-known port number (a 16-bit integer) over which the DSOM daemon, **somdd**, listens for requests for the current protocol. Each protocol should have its own port number. Select port numbers that are not likely to be used by other applications. (Check the `%ETC%\SERVICES` (on OS/2 and Windows NT) or /etc/services (on AIX) file for ports reserved for other applications on your machine.) Typically, values below 1024 are reserved and should not be used.

```
     SOMDPORT=3002
;       The port on which the DSOM daemon will listen for requests.
```

CSFactoryClass

is the name of a class capable of creating client-to-server connections for that protocol. The default **SOMobjects** configuration file contains valid settings of **CSFactoryClass** for the protocols that DSOM provides (SOMD_IPC, SOMD_TCPIP, and SOMD_NetBIOS). This setting should not be changed or removed.

```
     CSFactoryClass=SOMDCallStrmIIOP::CallStreamFactoryIIOP
;       The CallStreamFactory class name for this protocol.
;       This setting should not be changed or removed.
```

CSRegistrarClass

specifies the name of a class that **regimpl** can use to register a server supporting that protocol. The configuration file contains valid settings of **CSRegistrarClass** for the protocols that DSOM provides (SOMD_IPC, SOMD_TCPIP, and SOMD_NetBIOS). This setting should not be changed or removed.

```
    CSRegistrarClass=SOMDCSRegRI::CallStreamRegistrarRI
;       The CallStreamRegistrar class name for this protocol.
;       This setting should not be changed or removed.
```

CSTransportClass

specifies the name of a transport class supporting that protocol. The configuration file contains valid settings of **CSTransportClass** for the protocols that DSOM provides (SOMD_IPC, SOMD_TCPIP, and SOMD_NetBIOS). This setting should not be changed or removed.

```
    CSTransportClass=SOMDtipc::IPCTransportFactory
;       The Listening transport class name for this protocol.
;       This setting should not be changed or removed.
```

CSLocationName

CSLocationName specifies the protocol used by the server to contact the DSOM daemon, **somdd**, when registering itself. The default is to use the same protocol to contact the daemon as to communicate with client processes. This setting is meaningful only within the SOMD_IPC, SOMD_TCPIP, and SOMD_NetBIOS stanzas. This setting should not be changed or removed.

```
CSLocationName=SOMD_IPC
; This is the name of the protocol the server uses to communicate
; with the location service (SOMDD). This setting should not be
; changed or removed.
```

CSProfileTag

CSProfileTag is a unique tag representing the protocol within object references. These are defined by the OMG and DSOM. This setting should not be changed or removed.

```
    CSProfileTag=1229081857;
;       CallStreamFactory tag for this protocol; its decimal "IBM1".
;       This is a protocol-unique tag used to represent the protocol
;       within object references.
```

CALL_POOL_SIZE

CALL_POOL_SIZE specifies the average number of outstanding remote method calls a DSOM process expects to have at any one time, over that protocol. The default, if not specified, is 16. DSOM maintains a pool of internal objects, of this maximum size, for remote requests. If the number of such objects needed exceeds this limit, they are created and destroyed on demand.

```
  CALL_POOL_SIZE=16
;      The average number of requests that a client expects to send
;      simultaneously over this protocol, or if its a server, then
;      its the average number of simultaneous objects the server
;      expects to export (or return) over this protocol.
```

ENCAP_POOL_SIZE

ENCAP_POOL_SIZE specifies the average number of requests that a client expects to send simultaneously over the protocol, or, if a server, the average number of simultaneous objects the server expects to export (or return) over the protocol. The default is 4.

```
; ENCAP_POOL_SIZE=4
;      The average number of requests that a client expects to send
;      simultaneously over this protocol, or if a server,
;      the average number of simultaneous objects the server
;      expects to export (or return) over this protocol.
```

DSOM TCP/IP

The [SOMD_TCPIP] stanza, in conjunction with the [somd] stanza contains settings for the TCP/IP protocol. These settings are the same as those documented for the [SOMD_IPC] stanza, although the default values may differ.

```
[SOMD_TCPIP]
; Stanza for the DSOM TPC/IP protocol (UNO IIOP).
```

DSOM NetBIOS

The [SOMD_NetBIOS] stanza, in conjunction with the [somd] stanza contains settings for the NetBIOS Protocol. These settings are the same as those documented for the [SOMD_IPC] stanza, although the default values may differ.

```
[SOMD_NetBIOS]
; Stanza for the DSOM NetBIOS protocol (UNO IIOP).
```

SOMobjects Java Client

The [SOMD_JCLIENT]] stanza contains settings used for configuring the SOMobjects Java client.

SERVICES_FILE_TARGET

specifies the directory in which the Java client "well-known services" object reference file will be placed. This file is read by the Java client ORB during its initialization (that is, when the **resolve_initial_references** function is called).

```
SERVICES_FILE_TARGET=
; The web server directory location of the "well-known objects" file.
; This file can be placed anywhere in the HTML directory tree of the
; Java applet web server, as long as it it not protected by
; any security access mechanisms.  A recommended location for this
; file is in the top-level HTML directory, for example
```

```
; SERVICES_FILE_TARGET=d:\WWW\HTML     (/WWW/HTML on AIX)
; where "d:" is the drive letter.
; The services file will be named "services" and created in this
; directory during som_cfg execution.  If you do not intend to use
; Java clients from this SOM server machine, you can comment this
; line out.
```

Step 4. Issue somdchk

This step is optional. After you issue the **som_cfg** command you can re-run **somdchk** and save the output in a file. The output may be useful in later problem determination.

Issue the **somdchk** command to verify the configuration file settings.

somdchk evaluates the environment to verify whether DSOM can operate correctly. As described in **Implementing Classes** on page 304 and **DSOM Configuration** on page 18, to operate correctly DSOM must be able to find the appropriate libraries (DLLs), the Interface Repository, and, for servers, the Implementation Repository. The settings of various environment variables and/or configuration file settings help DSOM find the path to the libraries and repositories.

The **somdchk** program generates messages that evaluate the DSOM environment to determine whether the necessary SOM DLLs can be located, whether Interface and Implementation Repositories can be located, and it displays important environment settings. In verbose mode, **somdchk** gives the default settings for DSOM configuration-file settings and explains how DSOM uses them.

Invoke the program from the command line using the following. The **-v** option turns on verbose mode:

somdchk [-v]

The following example shows sample output from the **somdchk -v** command. Your output may differ.

Sample somdchk Output for AIX

```
     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 be set to the base directory of the SOMObjects
Developer Toolkit.

SOMENV = /usr/lpp/som/etc/somenv.ini.
SOMENV specifies a list of filenames (similar to SOMIR) that make
up the single, logical configuration file.
  Default is /usr/lpp/som/etc/somenv.ini

  /usr/lpp/som/etc/somenv.ini found.

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/somdcomm.dll found.

SOMDDIR    = /u/somuser/impl_rep/
Valid Implementation Repository found in /u/somuser/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/somuser/somuser.ir
SOMIR may be set to a list of file names which together form the
Interface Repository.

SOMDDEBUG    = 1
SOMDDEBUG may be set to 1 to enable run time 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.

SOMDPROTOCOLS = SOMD_IPC SOMD_TCPIP.
SOMDPROTOCOLS provides a list of communications protocols that a
DSOM client/server will attempt to use.
  Default is SOMD_IPC.

etc.
```

Step 5. Issuing the Configuration Command

som_cfg, the configuration command, configures DSOM and Object Services for a network of machines. See **Chapter 4, Naming Service** on page 17 of *Programmer's Guide for Object Services*.

On a single machine you need only to configure an Install host. For a multi-machine system, you first configure one Install host, then copy configuration information to the other machines, and then configure those other machines as DSOM hosts.

After Naming Service configuration is complete, **somdd**, the DSOM daemon remains running. The DSOM daemon can be stopped and restarted repeatedly without requiring reconfiguration. Servers can also be started, stopped, and restarted without reconfiguration. Servers need not be started manually; when a server is required by an application including the naming or security server), it is started automatically by **somdd**.

Be aware that if certain environment settings are changed after configuration is done, you must reconfigure and re-register all application servers and classes. See **Reconfiguring DSOM** on page 28.

Selecting the Install Host

One machine in a multiple-system installation is the Install host; the others are the DSOM hosts. The Security Service and the Global Root Naming Tree are stored at the Install host. See **Roots and Namespaces** on page 23 in *Programmer's Guide for Object Services* for a discussion of Naming Concepts. During DSOM processing the DSOM hosts communicate with the Install host for security authentication and for naming services. Therefore, your Install host should be physically secure to protect your installation's security information and should have the capacity to handle the communications and storage required by your applications.

Configuring the Install Host

Issue **som_cfg -i** for the Install host. **som_cfg** gives you progress messages.

The **som_cfg** tool performs the following tasks for **som_cfg -i**:

1. Verifies the environment; makes sure important variables are set.

2. Starts the DSOM daemon (**somdd**), if it is not already running.
3. Registers a naming server (a server to host the root Naming Context) into the (initially empty) Implementation Repository, in the directory designated by the current **SOMDDIR** setting. The default alias of the naming server is **namingServer**. After configuration is complete, this server's entry in the Implementation Repository can be viewed using **regimpl**.
4. Creates and initializes the global root Naming Context and local root Naming Context and the Factory Naming Context within the naming server. These Naming Context objects (as well as others created later) are stored persistently in the directory designated by the current **SOMDDIR** setting.
5. Creates the GLOBAL_OBJREF_FILE.
6. Binds the global root Naming Context to the local root Naming Context.
7. Updates the leftmost configuration file (specified by **SOMENV**) with information needed by clients to connect to the Naming Service. Two settings are added to the [somnm] stanza: **HOSTKIND** and **SOMNMOBJREF**. **SOMNMOBJREF** is the string form of an object reference (proxy) to the local root Naming Context. This information is used by the DSOM runtime within client applications to establish the initial connection between the client and the Naming Service. All other proxies can then be obtained from the Naming Service.
8. Builds the global naming tree. See **Naming Service Concepts** on page 27 in this section, and **Roots and Namespaces** on page 23 of *Programmer's Guide for Object Services*.
9. Configures the Security Service, which includes registering a security server in the Implementation Repository, with the default alias **securityServer**, and initializing a security database.
10. Configures the well-known services file for Java clients, if the **SERVICES_FILE_TARGET** variable in the [SOM_JOE] stanza is non-null. This is accomplished by copying and reformatting the information in the GLOBAL_OBJREF_FILE that contains a reference to the global root naming context object.

After Naming Service configuration is complete, **somdd**, the DSOM daemon remains running.

If you plan to reconfigure the Naming Service see **Reconfiguring DSOM** on page 28.

Copying the GLOBAL_OBJREF_FILE

The GLOBAL_OBJREF_FILE setting in the [somnm] stanza of the configuration file specifies the name of the file used by **som_cfg** to hold a reference to the global root Naming Context object. In the case of an Install host, the object reference is written to this file. In the case of a DSOM host, the object reference is read from this file. If unset, the default is to place the file **somnm.ref** in the directory pointed to by the **SOMDDIR** configuration variable of the [somd] stanza.

For a DSOM host this value must be set to a directory other than the default because **som_cfg** does not run if there is anything in the default directory.

Do the following steps:

1. Locate the file specified in the GLOBAL_OBJREF_FILE setting of the [somnm] stanza in the configuration file on the Install host.

If GLOBAL_OBJREF_FILE is unset, as it is in the default configuration file, locate the **$SOMBASE/etc/dsom/somnm.ref** file on AIX or the **%SOMBASE%\etc\dsom\somnm.ref** file on OS/2 or Windows NT. SOMDDIR should be set to the appropriate path in the [somd] stanza. If SOMDDIR is also unset, as it is in the default configuration file, then locate the file **%SOMBASE%\etc\dsom\somnm.ref** created when som_cfg -i was run on the Install host.

2. Make this file accessible to all DSOM hosts either by copying to all DSOM hosts or by using a shared file system. This file must appear in a directory other than the one specified by the SOMDDIR setting in the [somd] stanza of the configuration file at the DSOM host.
3. Update the GLOBAL_OBJREF_FILE setting of the [somnm] stanza in each DSOM host's configuration file to refer to this file using its fully qualified file name.

Configuring DSOM Hosts

If you are configuring systems for multi-system DSOM applications, after you configure one machine as Install host you then configure the others as DSOM hosts. Ensure that **somdd**, the DSOM daemon, is running at the Install host.

Issue **som_cfg -d** for each DSOM host. **som_cfg** gives you progress messages.

The **som_cfg** tool performs the following tasks for **som_cfg -d**:

1. Verifies the environment; makes sure important variables are set.
2. Starts the DSOM daemon (**somdd**), if it is not already running.
3. Registers a naming server (a server to host the local root Naming Context) into the (initially empty) Implementation Repository, in the directory designated by the current **SOMDDIR** setting. The default alias of the naming server is **namingServer**. After configuration is complete, this server's entry in the Implementation Repository can be viewed using **regimpl**.
4. Creates and initializes the local root Naming Context within the naming server. This Naming Context object and others created later are stored persistently in the directory designated by the current **SOMDDIR** setting.
5. Reads from the GLOBAL_OBJREF_FILE.
6. Binds the global root naming context on the Install host to the local root naming context.
7. Updates the leftmost configuration file (specified by **SOMENV**) with information needed by clients to connect to the Naming Service. Two settings are added to the [somnm] stanza: **HOSTKIND** and **SOMNMOBJREF**. **SOMNMOBJREF** is the string form of an object reference (proxy) to the local root Naming Context. This information is used by the DSOM runtime within client applications to establish the initial connection between the client and the Naming Service. All other proxies can then be obtained from the Naming Service.

After Naming Service configuration is complete, **somdd**, the DSOM daemon remains running.The DSOM daemon can be stopped and restarted repeatedly without requiring reconfiguration. Servers can also be started, stopped, and restarted without reconfiguration. Servers need not be started manually; when a server is required by an application (including the naming or security server), it is started automatically by **somdd**.

Be aware that if certain environment settings are changed after configuration is done, you must reconfigure and re-register all application servers and classes. See **Reconfiguring DSOM** on page 28.

Naming Service Concepts

This is a quick overview of the Naming Service and its relation to DSOM.

The Naming Service is a general directory service that allows an object reference to be associated with (bound to) a user-defined name, yielding a name binding. User-defined properties can also be associated with the name binding, a mapping from a name to an object reference. The Naming Service supports searching for the object reference, given its name (called *resolving* the name) or given a constraint on its associated properties.

Properties associated with name bindings are simply name-value pairs, where the names are unbounded strings and the values can be any CORBA type. The Naming Service provides methods for searching for name bindings whose properties match a given constraint expression. The constraint is a simple string, formed according to a constraint grammar. The grammar supports simple and compound boolean, logical, mathematical, and set expressions. Although property values can be any CORBA type, searches can only be performed on simple types.

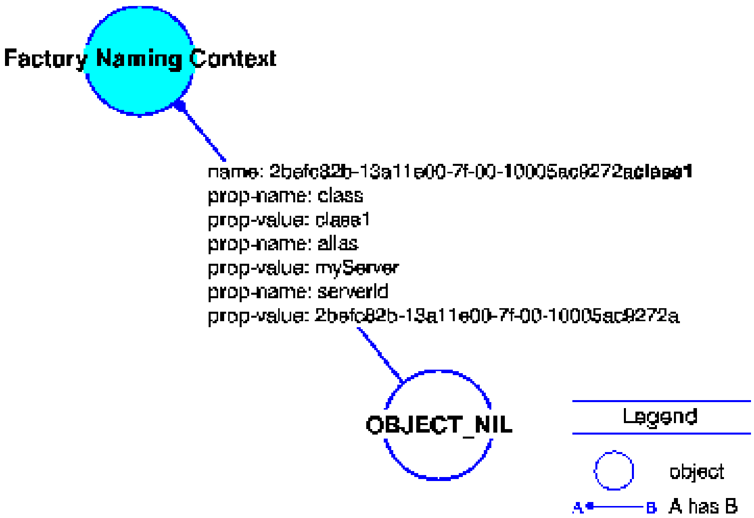

Figure 1. Naming Service entries made by DSOM when a server is associated with a class

For additional information, see **The Naming Service and Registering Servers** on page 40.

Structure

The Naming Service is composed of objects called naming contexts, organized in a tree structure, analogous to a file directory structure. There is a root *Naming Context*, which can refer to other Naming Contexts. The name-context objects at the leaves of the tree contain name bindings to application objects (rather than other Naming Contexts). The name-context objects that make up the Naming Service reside in one or more DSOM servers, so that client applications can access the Naming Service from anywhere in the network.

Naming-service names are data structures, not simple strings. A simple name, called a *name component*, is a structure consisting of two fields: an id and a kind. A name component specifies a particular object reference relative to a particular Naming Context. A compound name is a sequence of name components. It allows traversal of successive sub-contexts within a name tree, analogous to how a compound pathname can specify a file in a subdirectory of the current directory. Naming Service names are always specified relative to some Naming Context. Unlike file directories, there are no absolute names in the Naming Service.

Factory Service

DSOM provides an extension to the Naming Service, called the *Factory Service.* The Factory Service is implemented as a specialized and well-known Naming Context, known as the *Factory Naming Context*. When DSOM servers are registered, and application classes are associated with them, this information is stored in the Factory Naming Context of the Naming Service. At runtime, when client applications need to create remote objects, they can make remote invocations on the Factory Naming Context to request an appropriate factory object (an object capable of creating another object). This is done via the usual Naming Service interfaces for finding objects based on a well-known name or based on a given constraint expression. The Factory Naming Context finds an appropriate server, starts it dynamically, creates the requested factory object in the server, and returns to the waiting client a proxy to it. The client application then invokes methods on the factory proxy to create other objects as needed.

Reconfiguring DSOM

After the Naming and Security Service have been configured using **som_cfg**, the settings of **SOMDDIR** and **SOMNMOBJREF** and the contents of the directory specified by **SOMDDIR** collectively form a single DSOM environment. These settings should not be subsequently changed independently without reconfiguring the Naming Service.

In addition, the SOMNMOBJREF setting created by **som_cfg** embeds the SOMDPROTOCOLS, HOSTNAME, and SOMDPORT settings of the servers it registers and their associated daemon, **somdd**. Therefore you must reconfigure if you change these settings.

If it is necessary to change environment settings and reconfigure the Naming or Security Service, do the following:

- Change the **SOMDDIR** setting, if necessary, then remove all files in the directory indicated by **SOMDDIR**.
- Remove the **HOSTKIND** and **SOMNMOBJREF** settings from the configuration file.
- Rerun **som_cfg**.
- Re-register all application servers using **regimpl**.

Alternative Configurations

This section describes an alternative DSOM network configuration, which involves less configuration and runtime overhead than the default described previously. This alternative is suitable *only* for environments in which the only use of the Naming Service is through the DSOM Factory Service.

The default Naming Service network configuration, which results from configuring (using som_cfg) one machine as install host and the remaining machines as DSOM hosts, yields a network in which the DSOM daemon (somdd) and a Naming server run on all machines in the network. In other words, there are no 'pure client' machines in the default network configuration.

When a client application running on a DSOM host in such a network makes a request to the DSOM Factory Service, first a remote method invocation will be made on the local root Naming Context in that machine's Naming server, then a remote invocation is made on the global root Naming Context on the install host because DSOM's Factory Naming Context always resides in the global root Naming Context on the install host. The alternative configuration described below eliminates the first remote invocation and the client application contacts the install host directly.

In the alternative configuration, rather than configuring an install host and several DSOM hosts, one configures only the install host. One then transfers information from the install host to the DSOM hosts to allow them to contact the install host's Naming server directly, rather than via a local Naming server on the DSOM host. Such a configuration has the advantage that only the install host incurs the overhead of a Naming server process. If no other servers are to be run on a DSOM host, then the DSOM daemon (somdd) and the Implementation Repository can also be omitted from that host. (The DSOM host then acts as a client-only machine.) This configuration also has the advantage that accessing the DSOM Factory Service is more efficient.

This configuration is suitable only for networks in which the Naming Service is only used through the DSOM Factory Service. If any application to be run on the DSOM host uses the Naming Service to name or find application objects, then the host must be configured using **som_cfg** as described earlier.

Specifying an Alternative Configuration

To define an the alternative configuration, first configure the install host, as already described. Then copy the SOMNMOBJREF setting from the [somnm] stanza of the configuration file on the install host (stored in that file as a side-effect of running **som_cfg**) to the [somnm] stanza of the configuration files to be used on all the DSOM hosts. This setting requires several lines.

Every process running on a DSOM host must have SOMENV set to include some configuration file whose SOMNMOBJREF setting was copied from the install host, and it must appear in the [somnm] stanza. This is the information DSOM uses to contact the Naming Service directly on the install host at run time.

If it later becomes necessary to change a DSOM network from the alternative configuration to the default configuration, the following steps are required.

Note: The install host does not need to be reconfigured to perform this conversion.

1. Change the SOMDDIR setting on the DSOM host to refer to an empty directory. (Reconfiguring the DSOM host requires that any server registrations on that machine be redone.)
2. Copy the GLOBAL_OBJREF_FILE from the install host to the DSOM host, and set GLOBAL_OBJREF_FILE on the DSOM host to point to the copy, as described earlier.
3. Run **som_cfg -d** as described earlier, to configure the DSOM host.
4. Re-register any servers to be run on the DSOM host, using **regimpl**.

This change can be done gradually, converting a few machines at a time to be full-fledged DSOM hosts, as necessary.

Step 6. Configuring User Applications

The following sections tell how to configure DSOM applications. Configuring application consists of the steps:

1. Customize settings in **The Configuration File** on page 15. Each application might have different settings, so this step can be redone as needed.
2. Run **somdchk**. This step is optional.
3. **Registering Class Interfaces**.

4. Register servers and classes in the Implementation Repository using **regimpl**. See **The regimpl Registration Utility** on page 32
5. Migrate any 2.1 Implementation Repositories to the current level. See **Migrating 2.x Implementation Repositories to Current DSOM Format** on page 44

Also, see **Running DSOM Applications** on page 308.

This step can be done as part of your configuration sequence if you have existing applications. Or, you can repeat some or all of this step as often as you need once you have created applications.

Registering Class Interfaces

The Interface Repository (IR) is a collection of files that make up a database of information about the classes that a DSOM application uses. DSOM relies on the IR for the following:

- Dynamically loading classes (by obtaining the **dllname** modifier from the IR), including application classes, user-defined subclasses of **SOMDServer** (for servers only), user-defined subclasses of **ImplementationDef**, user-defined proxy-base classes and classes to support vendor-supplied DSOM protocols.
- Retrieving the value of the **factory** modifier for a class with an application-specific factory (used by servers only).
- Retrieving the value of the **baseproxyclass** modifier for classes with user-defined proxy-base classes.
- Registering (with **regimpl**) a server using a user-defined subclass of **ImplementationDef**. (The IR is used to determine the new attributes for which a value should be solicited from the **regimpl** user.)
- Making requests using the Dynamic Invocation Interface.
- Responding to the **get_interface** method.

If a DSOM application relies on any of these services, the appropriate classes should be registered in the IR prior to running the application. A class's **dllname** modifier need not be registered in the Interface Repository if any of the following holds:

- The class library (DLL) name is the same as the class name.
- The class is guaranteed to be used only after the DLL has been dynamically loaded for some other class, and the DLL provides a **SOMInitModule** that causes the class object to be created automatically when the DLL is loaded by the SOM Class Manager.
- The application client and server are each statically linked to the class library, and either create the class object explicitly (by calling *className***NewClass** or *className***New**) or the DLL provides a **_DLL_InitTerm** routine that causes the class object to be created automatically when the DLL is loaded (by calling a library-provided **SOMInitModule** routine). This requirement is not satisfied for applications using the **somdsvr** or **somossvr** server programs, because these server programs load all application classes dynamically.

Note: DSOM 2.x used the IR to discover information about method signatures for remote method invocations. The IR is no longer used in this way. For libraries built using SOMobjects 2.x, method signature information must be compiled into the IR as before.

Information that DSOM uses when making a remote method invocation is stored directly in the class library (DLL), provided the library was built using the current **ih** or **xih** emitter. If this information is not found, DSOM consults the IR. Class implementors can prevent method signature information, called a *marshal plan*, from being compiled into the DLL by using the **mplan** SOM IDL method modifier or the **nomplans** class modifier.

To register a class in the Interface Repository, compile the IDL description of the class by running the SOM Compiler and the **ir** emitter using the following command syntax:

```
sc -sir -u stack.idl
```

Before updating the IR, you should set the **SOMIR** environment variable or the corresponding entry in the configuration-file [somir] stanza. See **Interface Repository** on page 18.

When the **ir** emitter is run, only the last file specified by **SOMIR** is updated. At run-time, however, the sequence of files is examined from left to right. The **irdump** tool can be used to examine the contents of the IR. See **Using the SOM Compiler to Build an Interface Repository** on page 337 and **Managing Interface Repository Files** on page 338 for more information on the **ir** emitter and **SOMIR**.

Registering Servers and Classes

Before a server is used, it must be registered in the Implementation Repository. The Implementation Repository is a persistent store of **ImplementationDef** objects residing on a server machine in the directory indicated by the current **SOMDDIR** setting. The **ImplementationDef** class defines attributes necessary for the DSOM daemon to start a server and for the server to initialize itself.

Each **ImplementationDef** object in the Implementation Repository represents a single *logical server*. At run-time, each logical server is implemented by some executable program running as a server process, but the logical server exists regardless of whether that server program is running.

Note: If you have been using the Implementation Repository provided by DSOM 2.x, see **Migration Relationship to the 2.x Implementation Repository** on page 42 for important information about how the Implementation Repository has changed for DSOM 3.x.

Implementation Definitions

Details of the **ImplementationDef** object are not currently defined in the CORBA specification. The attributes defined are required by DSOM and are listed below. You can subclass **ImplementationDef** to add your own attributes, as described in **Customizing ImplementationDef Objects** on page 41.

impl_id (string)
Contains the DSOM-generated identifier for a server implementation. This identifier is unique throughout the network and can be used as a key into the Implementation Repository.

impl_alias (string)
Contains the alias for a server implementation, specified by the system administrator when registering the server. This alias must be unique within a particular Implementation Repository database and can be used as a key. Unlike the **impl_id** attribute, the alias does not need to uniquely identify the server throughout the network.

impl_program (string)
Contains the name of the program or command file that will be executed when a process for this server is started by **somdd**. If the full pathname is not specified, the directories specified in **PATH** will be searched for the named program or command file. The **impl_program** attribute need not be unique for different **ImplementationDef** objects.

Many servers are registered to use the DSOM default server program, **somdsvr**, or the object-services server, **somossvr**. The default server program name is **somdsvr**. If the **somossvr** server program is used, the **somOS::Server** class must be used. See **impl_server_class (string)** for additional information.)

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, on OS/2

```
icsdebug myserver
```

will start myserver under the control of the icsdebug debugger. Servers that are started automatically by **somdd** always pass their **impl_id** as the first parameter.

impl_flags (Flags)
Contains a bit-vector of flags used to identify server options (for example, the IMPLDEF_MULTI_THREAD flag indicates a multi-threaded server). Review the **impldef.idl** file for the complete set of valid **ImplementationDef** flags. Unused flag bits are reserved for future use by IBM.

impl_server_class (string)
Contains the name of the **SOMDServer** class or subclass to be instantiated by the server process during initialization, to yield the server's server object. The default is **SOMDServer**. Note that when the **somossvr** server program is used, the **somOS::Server** server class must also be used.

impldef_class
Contains the class name of the implementation definition. Class must inherit from **ImplementationDef**, which is the default.

config_file (string)
Contains the name of the configuration file to be used by the server, if different from the **SOMENV** setting of the user/process that initiates the server process such as the DSOM daemon. When the server invokes **SOMOA::impl_is_ready**, if the server's **ImplementationDef::config_file** attribute differs from the current **SOMENV** setting and is non-NULL, the contents of the configuration file named by **ImplementationDef::config_file** will be read, any DSOM run-time initialization performed during **SOMD_Init** will be refreshed, and for the duration of the server process the setting of **ImplementationDef::config_file** will be prepended to the current **SOMENV** setting.

The regimpl Registration Utility

The **regimpl** utility is used to register servers and classes.

Registering Servers

To register a server in the Implementation Repository, the system administrator can use a DSOM registration utility: **regimpl**. The regimpl has a command line interface. You can also use the **ImplRepository** interface to do the same function in an application.

Servers may be registered only on the machine on which they will run and should not be registered on another machine because the **HOSTNAME** and **SOMDPORT** entries in the configuration file of the **regimpl** user are assumed to be those of the server. This information is stored in the Naming Service when the server is registered, to be used by the DSOM Factory Service to assist clients in locating servers.

The settings in the configuration file at the time a server is registered must be consistent with those in effect when the DSOM daemon is started on the server machine. The **SOMDPROTOCOLS** setting used by the daemon must have at least one entry in common with the setting used when the server was registered, and for the common **SOMDPROTOCOLS** entries, the **HOSTNAME** and **SOMDPORT** entries must match.

Note: The examples of registering servers in the following sections assume the use of the generic DSOM server program, **somdsvr.exe**. A discussion on how to write a specific server program is found in **Basic Server Programming** on page 286.

When using the SOM object services, the **somossvr** server program and the **somOS::Server** server class must be used. This server program and server class provide persistent object references and other services. After registering a server to use **somossvr** and **somOS::Server**, run the server from the command line with the **-i** option the first time it is executed, to allow the server to initialize its persistent storage. Subsequently, the server can be started by **somdd** on demand, just as with **somdsvr**. See **Chapter 5, Object Services Server** on page 35 of *Programmer's Guide for Object Services* for more information on the **somossvr** server program and the **somOS::Server** server class.

Registering Classes

During the execution of **regimpl**, or when using the programmatic interface, DSOM updates the Implementation Repository. In addition to updating the Implementation Repository, **regimpl** and the programmatic interface allow you to designate which SOM classes a server supports. Although the **somdsvr** and **somossvr** server programs can load any accessible SOM class library using **somFindClass**, the DSOM Factory Service only creates a factory for a requested class in a server that is registered to support that class.

When you use **regimpl** to associate a class with a server, DSOM stores this association in the Naming Service. For more information on how DSOM updates the Naming Service as servers are registered, see **The Naming Service and Registering Servers** on page 40.

Servers can be registered to support specific classes or the special class keyword "**_ANY**." DSOM Factory Service users can then request a server that supports a specific class name or class **_ANY**. In addition to registering classes with specific servers, classes can be registered using the server alias keyword "**_LOCAL**." This indicates that clients should be allowed to create local instances of that class using the DSOM Factory Service, provided they have the same **HOSTNAME** setting as the one in effect at the time the class was registered for **_LOCAL** creation. (The **HOSTNAME** setting used in this case is the one found in the [somd] stanza of the SOMobjects configuration file.) For more information on the DSOM Factory Service, see **Creating Remote Objects** on page 245.

Registration Steps Using regimpl

To register a server implementation and its classes using the **regimpl** utility, at the system prompt enter:

```
regimpl
```

This brings up the DSOM Implementation Registration Utility menu, shown below.

Adding Implementations

1. To begin registering the new implementation, select `1. Add` from the `IMPLEMENTATION OPERATIONS` section: that is, at the `Enter operation`: prompt, enter `1`

```
DSOM IMPLEMENTATION REGISTRATION UTILITY
(C) Copyright IBM Corp. 1992, 1996. All rights reserved.

[ 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
 11. Add to all

[ SAVE & EXIT OPERATIONS ]
 12.SAVE and 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.

2. Enter a shorthand name for conveniently referencing the registered server implementation while using **regimpl**. Avoid using names that being with an underscore.

```
Enter an alias for new implementation:  myServer
```

3. Enter the name of the **ImplementationDef** class for **regimpl** to use for this entry (the default is **ImplementationDef**). For details on providing user-written subclasses of **ImplementationDef**, see **Customizing ImplementationDef Objects** on page 41.

```
Enter ImplDef Class name (default ImplementationDef): <return>
```

   If a user-defined subclass of **ImplementationDef** was chosen for this server, **regimpl** will then prompt for additional information as defined by the **ImplementationDef** subclass.

4. Enter the name of the program that will execute as the server. This may be the name of one of the SOMobjects server programs, **somdsvr** or **somossvr** (discussed in **Running DSOM Servers** on page 309) or a user-written server program. If the **somossvr** server program is used, the **somOS::Server** server class must also be used (see Step 8 in this list). 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>
```

5. Specify whether the server expects the SOM Object Adapter (SOMOA) to run each method in a separate thread. Note: You must ensure that methods executed by the server are *thread safe*.

```
Allow multiple threads in the server? [y/n] (default: no) : n
```

6. Specify whether the server should accept requests from authenticated clients only.

```
Make server secure? [y/n] (default: no) : <return>
```

7. Specify whether the server should be managed.

```
Make server a managed one? [y/n] (default: no) : n
```

8. Enter the name of the **SOMDServer** class or subclass that will manage the objects in the server. If the **somossvr** server program was specified for this server, then the **somOS::Server** server class must also be used.

```
Enter server class (default: SOMDServer) : <return>
```

9. Specify the configuration file to be used by the server, if different from the SOMENV setting of the user/process that initiates the server process (for example, the DSOM daemon).

```
Enter Config file name (default: none) : <return>
```

10. Each protocol listed in the **SOMDPROTOCOLS** setting will be presented; select those protocols that this server will support. If **SOMDPROTOCOLS** is not set, or if no protocols are selected, then SOMD_IPC is assumed.

```
Select protocol 'SOMD_IPC' [y/n] (default: yes) : y
```

The **regimpl** system next displays a summary of the information defined so far, and asks for confirmation. Enter `y` to save the implementation information in the Implementation Repository.

```
================================================================
Implementation id.........: 2befc82b-13a11e00-7f-00-10005ac9272a
Implementation alias......: myServer
ImplDef Class name........: ImplementationDef
Program name..............: somdsvr
Multithreaded.............: No
Server secure.............: No
Server class..............: SOMDServer
Configuration file........:
Protocol information......:
   Protocol: SOMD_IPC; Hostname: sherman; Port: 3002;

The above implementation is about to be added. Add? [y/n]  y

Implementation 'myServer' successfully added
```

At this point, **regimpl** records information about the server in the Naming Service, for use by the DSOM Factory Service. If the Naming Service has not been configured, or is not available, an error is reported, but the server is still registered in the Implementation Repository, and the next time the server's registration is updated, DSOM attempts to update the Naming Service. For information on how **regimpl** updates the Naming Service, see **The Naming Service and Registering Servers** on page 40.

Adding classes

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.

1. To begin, registering new classes, select `7. Add` from the `CLASS OPERATIONS` section; that is, at the `Enter operation`: prompt, enter `7`

```
[ 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
 11. Add to all

[ SAVE & EXIT OPERATIONS ]
 12. SAVE and EXIT

Enter operation: 7
```

2. Enter the name of a class associated with the implementation alias. This can be a fully-scoped class name (for example, ::M::**I** or M::**I**) or an unscoped class name (**I**), provided that clients use the same form when requesting a factory for the class using the DSOM Factory Service. The fully scoped form is recommended when the unscoped form is ambiguous. To indicate that a server can create instances of any class whose DLL can be loaded, use the keyword **_ANY** in place of an actual class name.

```
Enter name of class:  class1
```

3. Enter the alias for the server that implements the new class (this should be the same alias as given above). To indicate that DSOM clients running on this machine can create instances of the class locally (within the same process) using the DSOM Factory Service, use the keyword "_LOCAL" rather than an actual server alias.

```
Enter alias of implementation that implements class:  myServer
```

4. **regimpl** also lets you associate your own user-defined properties with the name bindings it creates when a class/server pair is registered. These additional properties can then be used in queries to the DSOM Factory Service by client programs. Additional properties are specified to **regimpl** as the property's name, followed by a space, followed by the property value. Signify no more additional properties by pressing `<return>.`

```
Enter additional property <name value> pairs:  Version 3.0
Enter additional property <name value> pairs:  Owner gardner
Enter additional property <name value> pairs:  <return>

Implementation id.........: 307e1c84-03bde08c-7f-oo-08005a883a0b
Implementation alias......: myServer
Class names...............: class1
   Property / Value.......: Version = 3.0
   Property / Value.......: Owner = gardner
```

5. All class/server associations are registered in the Naming Service, for use by the DSOM Factory Service. When a **regimpl** user associates a class with a server, as above, **regimpl** creates a name binding in the Naming Service to reflect that association. The server alias, server id, and class name are stored as properties of the name binding. For more information on how **regimpl** updates the Naming Service, see **The Naming Service and Registering Servers** on page 40.

```
The above class is about to be added.  Add? [y/n] y

Class 'class1' now associated with implementation 'myServer'
```

The top-level menu will then reappear. Repeat the previous five steps until all classes have been associated with the server. Finally, select `12. SAVE and EXIT` to exit the **regimpl** utility.

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.

- To enter interactive mode:

```
regimpl
```

- To add an implementation:

```
regimpl -A -i str [-p str] [-v str]
                  [-e str {param...}]
                  [-k {on|off}] [-m {on|off}] [-z str]
                  [-s {on|off}] [-g str]
                  [-t str {param...} [-t ...]]
```

- To update an implementation:

```
regimpl -U -i str [-p str] [-v str]
                  [-e str {param...}]
                  [-k {on|off}] [-m {on|off}] [-z str]
                  [-s {on|off}] [-g str]
                  [-t str {param...} [-t ...]]
```

- 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 previous commands:

-c *str* Class name (maximum of 16 -c names)

-e *str* ImplementationDef class name (optional; default ImplementationDef)

param Values for additional attributes needed by ImplementationDef subclass. Can be zero or more strings, delimited by spaces.

-g *str* Server configuration file name (optional)

-i *str* Implementation alias name (maximum of 16 -i names)

-m {on|off} Enable multi-threaded server (optional; default off)

-p *str* Server program name (optional; default: **somdsvr**)

-s {on|off} Allow server to be secure (optional; default off)

-t *str* Transport protocol

param Additional data needed by transport protocol. Can be zero or more strings, delimited by spaces.

The **-t** option designates one (of possibly many) transport protocols that the server supports. (**regimpl** intersects this list with those protocols actually available on the machine.) The string following the **-t** option is the name of a protocol as it appears in the configuration file (in the **SOMDPROTOCOLS** setting and in its own stanza).

Following a particular **-t** *protocol* option can be zero or more strings, delimited by spaces. (If there are spaces needed within a particular string, then the string must be enclosed in quotes.) These strings are the information required by the specified protocol. Each protocol provider specifies what strings are required or allowed for that protocol. (The protocols shipped with DSOM do not support any.) If any of the protocols specified generates an error due to the strings specified, then **regimpl** rejects the entire command.

-v *str* Server-class name (optional; default: **SOMDServer**)

-z *str* Implementation ID (optional; normally generated by DSOM)

As with the interactive **regimpl** commands, use the class name keyword **_ANY** to indicate that a server can create instances of any class whose DLL can be loaded. Similarly, use the server alias keyword **_LOCAL** to indicate that DSOM clients running on this machine can create instances of the class locally (within the same process) using the DSOM Factory Service.

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, this text-based interface is particularly useful in batch files.

For information on how **regimpl** updates the Naming Service when servers or classes are registered, see **The Naming Service and Registering Servers** on page 40.

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**). With the **ImplRepository** programmatic interface, it is possible for an application to define additional server implementations at run time. The global variable **SOMD_ImplRepObject** (in a server process) is initialized by **SOMD_Init** to point to the **ImplRepository** object that represents the Implementation Repository stored in the directory indicated by the current **SOMDDIR** setting. The following methods are defined on it:

- Add an implementation definition to the Implementation Repository. The value of the **impl_id** attribute in the supplied **ImplementationDef** object is optional; a unique **ImplId** will be generated for the newly added **ImplementationDef** unless the **impl_flags** attribute of the **ImplementationDef** specifies the IMPLDEF_IMPLID_SET flag. All other attributes, except **impl_alias**, are optional. As with registering a server using **regimpl**, the **add_impldef** method will attempt to record information about the server in the Naming Service, in addition to updating the Implementation Repository. If the Naming Service has not been configured or is not available (for example, when **somdd** is not running), an exception will be returned but the server will still have been registered in the Implementation Repository.

```
void  add_impldef (in ImplementationDef impldef);
```

- Delete an implementation definition from the Implementation Repository, given its **impl_id** attribute.

```
void  delete_impldef (in ImplId implid);
```

- Update the implementation definition (defined by the **impl_id** attribute of the supplied **ImplementationDef**) in the Implementation Repository. If the Naming Service has not been configured or is not available (for example, when **somdd** is not running), an exception will be returned but the server will still have been updated in the Implementation Repository.

```
void  update_impldef (in ImplementationDef impldef);
```

- Return a server implementation definition given its **impl_id** attribute value.

```
ImplementationDef  find_impldef (in ImplId implid);
```

- Return a server implementation definition, given its user-friendly alias (the **impl_alias** attribute of the **ImplementationDef**).

```
ImplementationDef  find_impldef_by_alias (in string alias_name);
```

- Return 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.

```
sequence<ImplementationDef>  find_impldef_by_class (
                     in string classname);
```

- Retrieve all **ImplementationDef** objects in the Implementation Repository.

```
ORBStatus  find_all_impldefs (
                     out sequence<ImplementationDef> outimpldefs);
```

- Search the Implementation Repository and return a sequence of the **impl_alias** attributes associated with each **ImplementationDef** object therein.

```
ORBStatus  find_all_aliases (out sequence<string> impl_aliases);
```

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* in the Naming Service. For more information, see **The Naming Service and Registering Servers** on page 40.

- Associate a class, identified by name, with a server, identified by the **impl_id** attribute of its **ImplementationDef**. This type of association is used to look up server implementations via the **find_impldef_by_class** method. The classname can be a fully scoped class name (such as, ::M::**I** or M::**I**) or an unscoped class name (like **I**), provided that clients use the same form when requesting a factory for the class using the DSOM Factory Service. The fully scoped form is recommended when the unscoped form is ambiguous. The class name keyword **_ANY** indicates that the server can create instances of any class whose DLL can be loaded, and the **ImplId** keyword **_LOCAL** indicates that DSOM clients running on this machine can create instances of the class locally (within the same process), using the DSOM Factory Service.

```
void  add_class_to_impldef (
                     in ImplId implid,
                     in string classname);
```

- Remove the association of a particular class with a server.

```
void  remove_class_from_impldef (
                     in ImplId implid,
                     in string classname);
```

- Remove the association of a particular class from all server implementations in the Implementation Repository.

```
void  remove_class_from_all (in string classname);
```

- Return a sequence of class names associated with a server.

```
sequence<string>  find_classes_by_impldef (in ImplId implid);
```

- Associate the specified class with all servers currently registered in the Implementation Repository.

```
ORBstatus  add_class_to_all (in string classname);
```

- Associate the specified class with the specified server. The optional **PVList** sequence associates additional, user-defined property values with the name binding created in the Naming Service.

```
ORBstatus  add_class_with_properties (
                    in ImplId implid,
                    in string classname,
                    in PVList pvl);
```

The Naming Service and Registering Servers

Because the DSOM Factory Service relies on the Naming Service, the Naming Service must be configured before DSOM servers can be registered. See **Naming Service Concepts** on page 27.

When a **regimpl** user associates a particular server with a particular class, DSOM stores this association in the Naming Service, as a name binding. This information is recorded in a specialized *Naming Context* (specialized for the DSOM factory-finding service), called the *Factory Naming Context*.

The first time a process (such as, **regimpl** or an application using the **ImplRepository** interface) attempts to add or update a class/server association, DSOM establishes a connection to the root Naming Context of the Naming Service. DSOM then invokes a method on the root Naming Context to obtain a proxy to the Factory Naming Context. The Factory Naming Context is the Naming Context in which associations between servers and classes are stored.

If the Naming Service is not available, then all operations that require an update to the factory context will return a system exception.

For each server/class pair registered, DSOM generates a name of the form *<serverUUID><className>* (the server's **ImplementationDef::impl_id** attribute concatenated with the class name). DSOM associates this name, in the Naming Service, with properties indicating the class name (*class*), server ID (*serverId*), server alias (*alias*) and information that allows DSOM clients to locate the server. Additional properties may be associated with the name (for example, by using the **add_class_with_properties** method of **ImplRepository** or via **regimpl**).

Although names and properties in the Naming Service are usually associated with, or bound to, non-NULL object references, the names and properties that **regimpl** stores in the Factory Naming Context are associated with NULL object references, indicating that the factory object does not yet exist. (In fact, the server in which the factory object will reside is probably not even running yet.) The DSOM Factory Service will generate these factory objects on demand.

When a server is registered to create any class, DSOM uses the keyword **_ANY** as the value of the *class* property.

When a class is registered for server **_LOCAL** (indicating that DSOM clients can create local instances of a class using the DSOM Factory Service), the *alias* property is set to **_LOCAL**. (**_LOCAL** should not be used as a real server alias.) DSOM then concatenates the string **_LOCAL** with the current setting of **HOSTNAME** (from the [somd] stanza of the configuration file) to construct the value of the *serverId* property. The Name bound in the Naming Service is likewise constructed by concatenating **_LOCAL**, the current **HOSTNAME** setting, and the class name.

When a server is registered, even if no classes are associated with it, DSOM attempts to update the Naming Service to contain the information DSOM clients need to locate the server. This is done primarily to provide continued support for the deprecated **SOMDObjectMgr** methods and may be discontinued in a future release. If no classes are registered for the server, **_NULL** is used as the value of the *class* property in the Naming Service, as a placeholder. If this update of the Naming Service cannot be performed (for example, if the Naming Service has not been configured or **somdd** is not running), the server is still registered in the Implementation Repository, and the Naming Service will be updated (if available) the next time the server's **ImplementationDef** is updated.

Each time an **ImplementationDef** is updated, the set of classes associated with the server is updated in the Naming Service. A new name binding is created, if necessary. Properties associated with the server in the Naming Service are also updated each time the server registration is updated.

Because server aliases are not guaranteed to be unique throughout the network (but only within a single Implementation Repository), multiple name bindings in the Naming Service may have the same *alias* property value. If this occurs, then the result of the (now deprecated) method **somdFindServerByName** will be nondeterministic. The first server entry that is found will be returned.

Customizing ImplementationDef Objects

DSOM allows users to store, in the Implementation Repository, instances of user-defined subclasses of **ImplementationDef**. This is useful when additional, user-defined attributes need to be saved as part of a server definition. Each **ImplementationDef** object in an Implementation Repository can potentially be an instance of a different subclass. The **regimpl** tools let you indicate which class is to be used for each entry. For each new attribute introduced by that class, **regimpl** prompts for a value as indicated in the IDL for the class.

The following conventions must be followed when writing a subclass of **ImplementationDef**:

1. Methods **externalize_to_stream** and **internalize_from_stream**, which are inherited from **CosStream::Streamable**, must be overridden.

   This is necessary so that instances of the class can be externalized and stored in the Implementation Repository database.

   Within the overriding implementation, each method must first make a parent method call, then store/retrieve the newly introduced attributes using the methods **write_string** and **read_string**. (New attributes that do not need to be stored persistently need not be stored/retrieved. Also, the new attributes and their values cannot exceed 255 characters.) The same order of attribute reading/writing should be used in both methods. For example,

   In **externalize_to_stream**:

   ```
   _write_string(...,attribute1);
   _write_string(...,attribute2);
   ```

   In **internalize_from_stream**:

   ```
   attribute1 = _read_string(...);
   attribute2 = _read_string(...);
   ```

2. Identify attributes that require user input.

   Newly introduced attributes for which **regimpl** should prompt must be identified by the special SOM IDL modifier **impldef_prompts** in the implementation section of the class's IDL definition. The syntax is as follows:

   impldef_prompts: *attribute1*, *attribute2*, ...;

   More than one occurrence of this modifier in the IDL is acceptable; in such a case the following equivalence rule applies:

impldef_prompts: ***attribute1*****;**
impldef_prompts: ***attribute2*****;**

is equivalent to

impldef_prompts: ***attribute1*****,** ***attribute2*****;**

The new attributes for which **regimpl** can prompt must be of type **string**. This is not as strict a restriction as it appears, since servers that use this attribute can convert it to an appropriate format. For example, a string representation of a numeric value could be stored, and converted to an **integer** or **float** by a customized server.

The get and set methods for the new attributes must adhere to the *caller-owned* memory-management policy (indicated by the IDL modifier **memory_management=corba** or by using the **caller_owns_result** and **caller_owns_parameters** IDL modifiers). Because these attributes are of type string, this means that the attributes must be annotated with the **noget** and **noset** IDL modifiers. In your implementation of the **set** methods, you must make a copy of the input string before storing it; in your implementation of the **get** methods, you must make a copy of the string to be returned.

3. Package the class as a DLL so DSOM can dynamically load the class, using **somFindClass**.
4. Compile the new subclass's interface into the Interface Repository.

   Ensure that the interface repository specified by **SOMIR** contains this interface. This is required so that SOM can load the class (by getting the **dllname** modifier from the IR) and so that **regimpl** can query the value of the **impldef_prompts** modifier.

Migration Relationship to the 2.x Implementation Repository

The Implementation Repository has been improved significantly in DSOM over earlier releases. The first topic below discusses the main differences between 2.x Implementation Repositories and the Implementation Repositories of the current release. The second topic describes a tool, **migimpl3**, for converting 2.x Implementation Repositories to 3.x Implementation Repositories. (This conversion must be done on any existing 2.x Implementation Repositories before they can be used with the current release of DSOM.)

Differences between 2.x and 3.x

This section describes the major differences between DSOM 2.x Implementation Repositories and DSOM 3.x Implementation Repositories.

- For increased scalability and manageability of the Implementation Repository, DSOM no longer requires access to the Implementation Repository by client programs. Instead, the information needed by clients that was previously stored in the Implementation Repository is now stored in the Naming Service. Information no longer stored in the Implementation Repository includes the associations between servers and classes, and information needed for clients to locate servers.
- In the current DSOM, the Implementation Repository files stored in the **SOMDDIR** directory contain only the information needed on the server machine. (This includes information that the DSOM daemon needs to automatically activate the server, and information that the server needs to initialize itself). Although the underlying implementation is different, the interfaces for registering servers have been preserved as much as possible.

  Because server/class associations in DSOM are now stored in the Naming Service, the Naming Service must be configured and the DSOM daemon running on the machines on which naming servers will run before updating or accessing this information (for example, with **regimpl** or the **ImplRepository** programmatic interface). In addition, servers must be registered by running **regimpl** on the machine on which the server will run.

Because information that DSOM clients need to locate servers is now stored in the Naming Service, the **impl_hostname** attribute of **ImplementationDef** objects is no longer used by DSOM.

Because all server registrations throughout the network result in updates to the Naming Service, it is possible for multiple server entries in the Naming Service to have the same server alias. (Server aliases are required to be unique within a single Implementation Repository, but not within the Naming Service.) This means that **somdFindServerByName** (which is now deprecated) is no longer deterministic; it returns the first entry in the Naming Service that matches the given server alias.

- Because DSOM now supports communication over multiple protocols simultaneously, and each protocol can have its own **HOSTNAME** and **SOMDPORT** setting, this information is taken from the configuration file at the time the server is registered, rather than being specified explicitly. This is another reason that the server must be registered on the machine on which it will run.
- For greater extensibility of the Implementation Repository, DSOM now supports user-defined subclasses of the **ImplementationDef** class. A single Implementation Repository can contain a heterogeneous mix of **ImplementationDef** objects. For more information, see **Customizing ImplementationDef Objects** on page 41.
- The current DSOM offers an improved storage mechanism for the Implementation Repository, with improved performance and scalability. This includes the addition of file-level locking of Implementation Repository files. As a result of these changes, the **regimpl** tool no longer offers an *abort* operation after an operation has been confirmed.
- The port number of the DSOM daemon that is used to contact a server (**SOMDPORT**) is recorded when the server is registered, from the configuration file, rather than requiring the server's clients to have a matching setting for **SOMDPORT**. This means that clients can simultaneously communicate with different DSOM daemons using different port numbers.
- The object reference file and backup file for servers that create object references using the **BOA::create** method are now stored in the **SOMDDIR** directory and can be shared among multiple servers. The object reference files are no longer specific to a single server, and the **impl_refdata_file** and **impl_refdata_bkup** attributes of **ImplementationDef** are no longer used. In addition, the storage mechanism of the object reference file has been improved for performance and scalability. These changes require that object reference files created by DSOM 2.x be invalidated when a server system upgrades to DSOM 3.x.
- New attributes have been added to **ImplementationDef** to support new features of DSOM. For example, a **config_file** attribute allows a server's run-time environment to differ from the user or process that starts the server (for example, the DSOM daemon). The interfaces to **regimpl** have been revised to support these new features.

Migrating 2.x Implementation Repositories to Current DSOM Format

Because the format of 2.x Implementation Repositories differs from the format of current Implementation Repositories, a migration tool has been provided to assist in converting from one format to the other. This tool is called **migimpl3**.

Use **migimpl3** to migrate entries from a 2.x repository into a newer style repository. The only 2.x **ImplementationDef** entries that can migrate into a current repository are those that have their **impl_hostName** attribute set either to `localhost` or to the value of the current **HOSTNAME** environment setting. This is necessary because DSOM now requires that a server be registered only on the machine on which that server will run. Therefore, server entries should only be converted from 2.x to current-level Implementation Repositories on the machine on which the server will run.

In addition to converting server registrations to the current DSOM format, use **migimpl3** to register the server/class associations with the Naming Service. (These were previously stored in the 2.x Implementation Repository.) This information is used by DSOM clients and the DSOM Factory Service to locate the server.

For **migimpl3** to update the Naming Service, the **SOMDDIR** setting in the [somd] stanza of the SOMobjects configuration file should be set to the name of the Implementation Repository directory for both DSOM 2.1 and the current version of DSOM. This means that the DSOM 2.1 format files to be migrated should be copied into the **SOMDDIR** directory that was in effect when **som_cfg** was run.

The Implementation Repository database for DSOM 2.x consists of the following files:

somdimpl.dat
somdimpl.toc
somdcls.toc
somdcls.dat

The current form of the DSOM Implementation Repository database consists of the following files:

alias.db
aliasdat.db
impl.db

The **migimpl3** utility also requires that Naming Service configuration be complete and that the DSOM daemon, **somdd**, be running on the machines on which Naming Server server processes are located. The Naming Servers need not be running, **somdd** activates them as necessary.

The syntax of the **migimpl3** command is as follows:

migimpl3 [-l] [-U] [-i *str1* [-i *str*] ...]

The **-l** option is a lower case “el”.

To convert all entries from the Implementation Repository Database of DSOM 2.x to that of the current-level DSOM (provided that the **impl_hostname** attribute matches **HOSTNAME**), simply enter the command:

```
migimpl3
```

Default execution for the **migimpl3** operation is as follows:

- It does not replace any existing entry in the 3.x Implementation Repository. An error of `duplicate alias/implid entry` is returned if a matching **impl_alias** or **impl_id** is found in the Implementation Repository of the current-level DSOM.

- It does not convert entries that have the **HOSTNAME** setting `localhost` unless the **-l** option has been specified.
- The **config_file** attribute in the current-level **ImplementationDef** is not set.
- The **SOMDPROTOCOLS** setting determines which protocols are used when converted entries are registered. See **SOMDPROTOCOLS** on page 19 for more details on the **SOMDPROTOCOLS** setting.
- Although the **ImplementationDef** attributes **impl_hostname**, **impl_refdata_file**, and **impl_refdata_bkup** are no longer used in this release, these attributes have been retained for backward compatibility. The migration tool will retain any settings for these attributes found in the 2.x Implementation Repository. These settings will not be displayed, however, when viewing the implementation using **regimpl**. (They can still be accessed programmatically using the **ImplRepository** and **ImplementationDef** interfaces.)

The optional parameters to the **migimpl3** command specify the following operations:

-l
: Converts to an Implementation Repository of current-level DSOM if the DSOM **impl_hostname** attribute matches the **HOSTNAME** setting or is set to `localhost`. If the **-l** option is not specified, the **migimpl3** tool does not convert entries that have `HOSTNAME=localhost`.

-U
: Updates any existing entry in the current-level Implementation Repository. No update is performed if the corresponding **impl_id** of the **impl_alias** name does not exist in the current Implementation Repository database; instead, an error of update failure is returned. The **-U** option updates the current Implementation Repository for entries whose hostname field matches **HOSTNAME** or is set to `localhost`. (In other words, the **-U** option implies the **-l** option.)

-i *str*
: Specifies the Implementation alias names to be converted or updated (with a maximum of 16 **-i** names). Only the specified entries are converted/updated.

Moving Servers

In DSOM, it is possible to move a server from one machine to another, for purposes of system maintenance, system load balancing, and so on. To move a server:

- The server implementation (program and DLLs) must be moved to the new machine.
- Any data files associated with the server must be moved to the new machine.
- The server implementation definition must be removed from the Implementation Repository on the original machine and added to the Implementation Repository on the new machine. If the server program name has changed, the new name should be entered in the new Implementation Repository entry.

Checking Configuration Values

At times, you need to know the setting of the values defined in the configuration file. SOMobjects provides three function to perform this action: **somutgetenv**, **somutgetshellenv** and **somutresetenv**.

Using somutgetenv

The **somutgetenv** function lets a program determine the current setting of a value defined in the configuration file. The configuration file settings are read only on the first call to this function. Thereafter, calls to this function consult an in-memory version of the configuration settings. To refresh the in-memory settings, first call **somutresetenv** and then **somutgetenv**. The **somutgetenv** function has the syntax:

char * SOMLINK somutgetenv (char * *name*, char * *stanza*);

where *name* is the identifier whose value is requested and *stanza* is a stanza name. The given *name* must represent an identifier within the specified *stanza.* The **somutgetenv** function returns NULL if no value is found. The caller should not free the returned string value.

As an example, the following function call could be made, based on the **somenv.ini** excerpt shown previously:

```
value = somutgetenv ("CSFactoryClass", "[SOMD_TCPIP]");
```

The string value `SOMDCallStrmIIOP::CallStreamFactoryIIOP` would be returned from this call.

Using somutgetshellenv

The **somutgetshellenv** function returns the value of a SOMobjects setting and is identical to the **somutgetenv** function, except that it first checks the system environment to determine whether a value for the specified identifier exists. This is equivalent to calling the C library function **getenv** prior to calling **somutgetenv**. If the symbol is not defined in the system environment, this function uses **somutgetenv** to locate the requested identifier. The **somutgetshellenv** function has the syntax:

char * SOMLINK somutgetshellenv (char * *name*, char * *stanza*);

The **somutgetshellenv** function returns NULL if no value is found. Observe that the caller should not free the returned value.

As an example, the following function call could be made, based on the **somenv.ini** excerpt shown previously:

```
value = somutgetshellenv ("CSFactoryClass", "[SOMD_TCPIP]");
```

This call would return the string value `SOMDCallStrmIIOP::CallStreamFactoryIIOP` provided that **CSFactoryClass** was not defined in the system environment.

Using somutresetenv

If any values have been changed in the configuration file while the current process is running, the **somutresetenv** function can be called to refresh the in-memory representation of the configuration file. The **somutresetenv** function has the syntax:

void SOMLINK somutresetenv (char * ***newenv***);

where *newenv* is a new setting for the **SOMENV** environment variable. If *newenv* is non-NULL, **somutresetenv** resets the **SOMENV** environment variable, using the C library function **putenv** with the specified value.

The **putenv** call made to update the system environment affects only the C run-time environment used by functions **somutresetenv**, **somutgetenv** and **somutgetshellenv**.

The **somutresetenv** function should not be called if another thread is currently accessing the configuration file settings.

Chapter 3. Tutorial for Implementing SOM Classes

This tutorial contains five examples showing how to implement SOM classes.

If you plan to implement classes, follow the steps in the tutorial to understand the steps, processes, files, and the relationships of SOM classes

Even if you expect only to use SOM classes implemented by others, this tutorial can help you understand the process of using SOM classes.

Basic Concepts of SOM

The 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 object-oriented programming 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 or hidden from the caller. A program can use the object's methods without knowing 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 can be manipulated is represented by an object type. By declaring a type for an object variable, a programmer specifies an interface that can be used to access that object. The SOM Interface Definition Language (IDL) defines object interfaces. The interface names used in these IDL definitions are also the type names used by programmers when typing SOM object variables.
- A class defines the implementation of objects. The implementation of any SOM object is defined by a specific SOM class. A class definition begins with an IDL specification of the interface to its objects. The name of this interface is also used as the class name. Each object of a given class may also or instantiation of the class.
- SOM classes provide external data structures and functions that aid in the efficient use of objects whose interfaces are declared using SOM IDL. These low-level externals are determined by a class's IDL and are called the class's Abstract Binary Interface (ABI). Different ABI styles are with various efficiency implications. Specific details concerning different ABI styles are hidden by language bindings.
- Inheritance, or class derivation, is a technique for developing new classes from existing classes. The original class is called the base, parent, or 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. Through inheritance, a new class can override methods of its parent to provide new or changed function. 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 expanded capabilities. When 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. 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. The terms class methods and class variables are used to distinguish between the methods and variables of a class object versus those of its instances. 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 is creating an instance of itself. Just as classes are derived from parent classes, so metaclasses can be derived from parent metaclasses to define new functions for class objects.
- The SOM system contains three primitive classes that are the basis for all subsequent classes:
  - **SOMObject Class**
  - Root ancestor class for all SOM classes
  - **SOMClass Class**
  - Root ancestor class for all SOM metaclasses
  - **SOMClassMgr Class**
  - 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 and unloading

  SOMClass is defined as a subclass 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. **Figure 2** illustrates typical relationships of classes, metaclasses, and objects in the SOM run time. (This figure does not include the **SOMClassMgrObject**.)

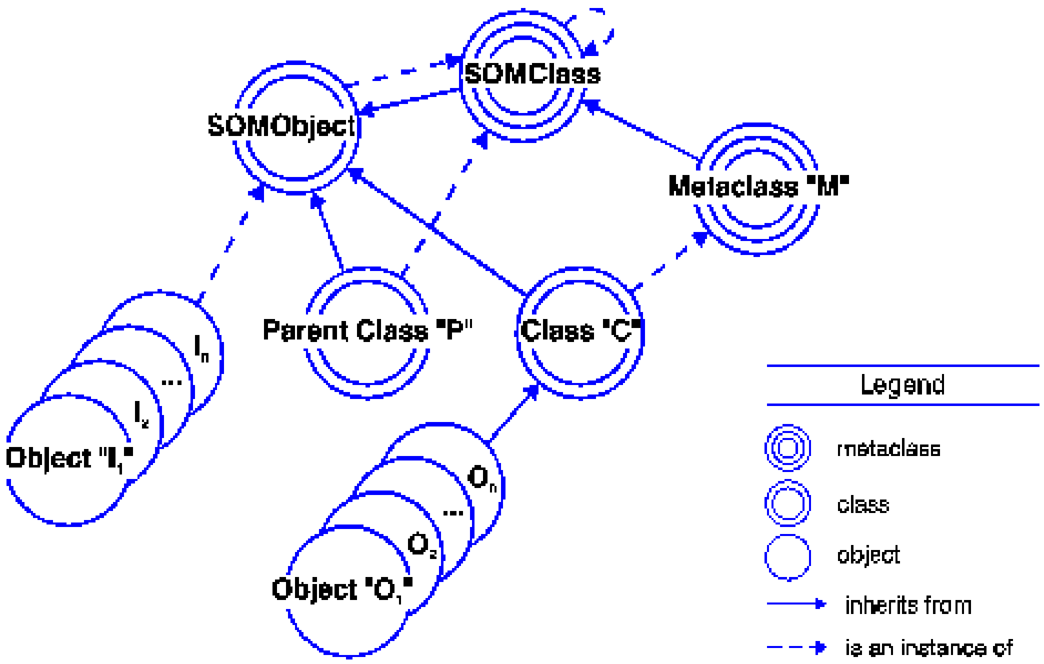

Figure 2. Typical class, metaclass and object relationships

SOM classes are designed to be language neutral. 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 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, the **.idl** file.

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

  - The interface name or class name and the names of the class's parents,
  - The names of the class's attributes and the signatures of its new 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 are automatically defined for use by the application program. Instance variables that are not attributes are hidden from the user.

- After 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 is defined. Before invoking the SOM Compiler, the class implementor can set an environment variable that determines which emitters (output-generating programs) the SOM Compiler calls and, consequently, to which programming language and operating system the resulting binding files relate.

In addition to the implementation template file itself, the binding files include two language-specific header files that are **#include**d in the implementation template file and in application program files. The header files define 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: 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 that use the objects and methods of the newly implemented class. (Observe 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 executes the program.

Attributes versus 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 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 **Modifier Statements** on page 133.)

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 **Implementation Statements** on page 132.

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.

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 to produce the following binding files:
   - Template implementation file:
   - **.c** file for C programs
   - **.C** file (on AIX) or a **.cpp** file (on OS/2 or Windows NT) for C++ programs;
   - Header file to be included in the implementation file:
   - **.ih** file for C programs
   - **.xih** file for C++ programmers
   - Header file to be included in client programs that use the class:
   - **.h** file for C clients
   - **.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** command as described in **Chapter 6, The SOM Compiler** on page 155. 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.

Using the Tutorial

The following examples show the 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 5, SOM Interface Definition Language** on page 115. Customization of each implementation file (step 3) is illustrated in both C and C++.

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.

When the SOMobjects Toolkit is configured, a choice is made between somcorba and somstars for the style of C bindings the SOM Compiler generates. 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. In the examples, a "*" does not explicitly appear for types that are pointers to objects. If your system is configured 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** on page 71 and **Object Types** on page 124**.**

Sequence of the Tutorial Examples

- **Example 1. Implementing a Simple Class with One Method** — 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** — 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** — 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** — 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** — 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 (as defined as in Example 4).

Example 1. Implementing a Simple Class with One Method

Example 1 defines a class Hello that introduces one new method, `sayHello`. When invoked from a client program, the `sayHello` method prints the fixed string Hello, World! The example follows the steps described in **Basic Steps for Implementing SOM Classes** on page 53.

1. Define the interface to class Hello that 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 name of a new class and any parents (base classes) it might have (here, root class **SOMObject**). The body of the interface declaration introduces the method `sayHello`. 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!".
               * This method returns the string "Hello, World!". */
   };
   ```

The method `sayHello` has no (explicit) arguments and returns no value. The characters "// " start a line comment that 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. The SOM Compiler ignores throw-away comments that start with the characters "//#" and finish at the end of the line.

Note: For simplicity, this IDL fragment does not include a releaseorder modifier; the SOM Compiler issues a warning for the method `sayHello`. For directions on using the releaseorder modifier to remove this warning, see **Modifier Statements** on page 133. (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, as follows:

```
> sc –s"c;h;ih" hello.idl      (for C bindings)
> sc –s"xc;xh;xih" hello.idl   (for C++ bindings)
```

When set to generate C binding files, the SOM Compiler generates the following template implementation file, named `hello.c`. The template implementation file contains stub procedures for each new method; these are procedures to be filled in by the implementor.

```
#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");
}
```

The terms **SOM_Scope** and **SOMLINK** in the prototype for all stub procedures are defined by SOM. In the method procedure for the `sayHello` method, **somSelf** is a pointer to the target object (here, an instance of the class Hello) that responds to the method. A somSelf parameter appears in the procedure prototype for every method, because 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**. The method can use this parameter to return exception information if the method encounters an error. (Contrast the prototype for the `sayHello` method in steps 1 and 2.)

The remaining lines of the preceding template are described in **Chapter 7, Implementing Classes in SOM** on page 171. 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`, similar to the one above.

In addition to generating a template implementation file, the SOM Compiler generates implementation bindings and usage bindings. These files are named `hello.ih` and `hello.h` for C bindings, and they are named `hello.xih` and `hello.xh` for C++ bindings. The `hello.c` file shown 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 or `hello.cpp`) 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");
}
```

4. Create a client program that uses the class.

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

   A C programmer would write the following program in `main.c`, that 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);
}
```

   In the statement `obj = HelloNew()` the `hello.h` header file automatically contains the SOM-defined macro *className***New()**, that is used to create an instance of the *className* class (here, the `Hello` class). In C a method is invoked on an object by using the form:

   ***_methodName*(*objectName*, *environment_arg*, *other_method_args*)**

   as used in the statement:

```
    _sayHello (obj, somGetGlobalEnvironment());
```

   As shown in this example, you can use the **somGetGlobalEnvironment Function** supply the (**Environment ***) argument of the method.

   The code uses **somFree Method** to free the object created by `HelloNew()`. **somFree** does not require an (**Environment ***) argument.

   A C++ programmer may write the following program in **main.C** or **main.cpp**, 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);
    }
```

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.)

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

Note: The environment variable **SOMBASE** represents the directory in which SOM has been installed.

Under AIX, for C programs

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

Under AIX, for C++ programs

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

Note: When building a multithreaded application, use the **xlc_r** (for C) or **xlC_r** (for C++) compiler instead of **xlc** or **xlC**.

Under OS/2 or Windows NT, for C programs

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

Under OS/2 or Windows NT, for C++ programs

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

6. Execute the client program.

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

Example 2. Adding an Attribute to the Hello Class

Example 1 introduced a class `Hello` that has a method `sayHello` which 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 called `msg` in this example. 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);
   ```

An attribute can be used instead of an instance variable to define **get** and **set** methods without having to write their method procedures. The new interface specification for Hello, that results from adding attribute `msg` to the Hello class, follows:

```
#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. Implementing a Simple Class with One Method** to produce 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 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. Because the **_get_msg** method name begins with an underscore, the method is invoked with two leading underscores in C.

```
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 to invoke the **_set_msg** method to initialize the `msg` attribute before the `sayHello` method is invoked. Because the **_set_msg** method name begins with an underscore, the C client program invokes the method with two leading underscores.

For C Programs

```
#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++ Programs

```
#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,.
6. Execute the client program:

```
> hello
Hello World Again
```

Example 3. Overriding an Inherited Method

In object-oriented programming a subclass can replace an inherited method implementation with a new implementation especially appropriate to its instances. This is called overriding a method. Sometimes a class introduces methods that every descendant class is expected to override. For example, **SOMObject** introduces the **somPrintSelf Method**; a SOM programmer generally overrides this method when implementing a new class.

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

This example illustrates how to override **somPrintSelf** for the `Hello` class. An important identifying characteristic of `Hello` objects is the message they hold. The following steps show how to override **somPrintSelf** in `Hello` to provide this information.

1. Modify the interface declaration in `hello.idl` to override the **somPrintSelf** method in `Hello`, in **hello.idl** in the form of an implementation statement, that gives information about the class, its methods and attributes, and any instance variables. In this example, the implementation statement introduces the modifiers for the `Hello` class:

```
#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
};
```

somPrintSelf introduces a list of modifiers in the class Hello. Modifiers are like C or C++ **#pragma** commands and give specific implementation details to the compiler. This example uses only the override modifier. 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) is 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. Adding an Attribute to the Hello Class** to include new stub procedures as needed (in this case, for **somPrintSelf**). Here 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);
}
```

The SOM Compiler adds code letting the Hello class redefine **somPrintSelf** and 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` uses to implement the **somPrintSelf** method). This parent method call is accomplished by the macro **Hello_parent_SOMObject_somPrintSelf**, defined in `hello.ih`.

The stub procedure for overriding the **somPrintSelf** method does not include an Environment parameter 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. The unnecessary parent method call 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 in 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. Notice the new message text. For C programs:

```
#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++ programs:

```
#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.
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 **somPrintSelf Method**, introduced by **SOMObject**. As 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** which provides 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. Initializers (including how to introduce new initializers that take arbitrary arguments, and how to explicitly invoke initializers), are described in **Initializing and Uninitializing Objects** on page 195.

The overall process of overriding **somDefaultInit** is similar to the previous example. 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 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. The SOM Compiler extends the existing implementation file. The following example shows the initializer stub procedure 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.

   The `msg` instance variable is set in the implementation template rather than in the client program. Therefore, 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, ev, "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.

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. Clients also can write messages to printers and disk files. This example references two additional classes: Printer and Disk. The Printer class manages messages to a printer, and the Disk class manages messages sent to files. Define these classes 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 below.

1. Modify the interface declaration in `hello.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`. **SOMObject** is a parent of Disk and Printer.

The IDL specification declares an enumeration outputTypes for the different forms of output and an attribute output whose value depends on where the client wants the output of the `sayHello` method to go.

OM IDL allows the use of structures, unions, enumerations, constants and typedefs, both inside and outside the body of an interface statement. Declarations that appear inside an interface body are emitted in the header file `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 **Running the SOM Compiler** on page 161).

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.

   The implementation for the **somDefaultInit Method** does not reflect the addition of two new parents to Hello 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 run the implementation emitter. Once this is done, the code in the renamed methods can be merged into the new templates, which include all the appropriate parent method calls. When this is done here, the new implementation for **somDefaultInit** appears:

```
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, ev, "Initial Message");
}
```

3. Continue to customize the implementation file, `hello.c`. The `sayHello` method discussed in Example 2 now allows 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. The **case** statements use 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.

```
#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
    __set_output(a, ev, Hello_screen) ;
    _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. Include the implementation files for the Printer and Disk classes.
6. Execute the client program. 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
```

Continuation of SOM

This chapter described features of SOM IDL that are be useful to C and C++ programmers.

SOM IDL 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 5, SOM Interface Definition Language** on page 115 gives a complete description of the SOM IDL syntax. **Chapter 6, The SOM Compiler** on page 155 describes how to use the SOM Compiler. **Chapter 7, Implementing Classes in SOM** on page 171 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 4. Using SOM Classes in Client Programs

This chapter discusses the use of fully implemented SOM classes. Read this chapter if you are using SOM classes created by someone else. This chapter tells how to instantiate an object and invoke methods from within an application program. If you are creating classes, read this and **Chapter 5, SOM Interface Definition Language** on page 115, **Chapter 6, The SOM Compiler** on page 155 and **Chapter 7, Implementing Classes in SOM** on page 171 for information about the SOM Interface Definition Language (SOM IDL) syntax and other details of class implementation.

Programs that use a class are client programs. A client program can be written in C, C++ or another language. This chapter describes how client programs can use SOM classes Using a SOM class involves creating instances of a class, invoking methods on objects and so forth. All methods, functions and macros described can be used by class implementors within the implementation file for a class.

Note: Using a SOM class 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 used without the C or C++ bindings. For example, users of other programming languages can use SOM classes; C and C++ programmers can use a SOM class without using its language bindings. The language bindings offer a more convenient programmer's interface to SOM. Vendors of other languages may offer SOM bindings; check with your language vendor for possible SOM support.

To use C or C++ bindings for a class, a client program must use the #include preprocessor directive to include a header file for the class. 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 depends on the IDL interface definition available 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 (a client program that does not know at compile time what classes it will be using) should include the SOM-supplied bindings for **SOMObject**, provided in the header file **somobj.h** (for C programs) or **somobj.xh** (for C++ programs).

This chapter shows how to do each SOM task:

- With C, using C bindings
- With C++, using C++ bindings
- Not using SOM's C or C++ language bindings.

If neither of the first two approaches is applicable, use the third approach.

Contents

Example Client Program Using A SOM Class

This C program uses the class `Hello` as defined in the Tutorial. The `Hello` class provides one attribute, msg, of type string, and one method, `sayHello`. The `sayHello` method 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());
    _somFree(obj);
    return(0);
}
```

This example is the C++ version:

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

Both client programs produce the output:

```
Hello World Again
```

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. To declare obj to be a pointer to an object that has type *interfaceName*, code:

`interfaceName obj ;` in C programs

`interfaceName *obj ;`	in C++ programs

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
`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`. 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, it might be an instance of some subclass of `Animal`; perhaps, the `Cat` class.

All SOM objects are of type **SOMObject**, even though they may not be instances of the **SOMObject** class. If you do not know, at compile time, what type of object the variable will point to, the following declaration can be used:

`SOMObject obj;`	in C programs
`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*. 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

If a C programmer prefers to use explicit pointers to *interfaceName* types, then the SOM Compiler option -**maddstar** can be used when C binding files are generated. 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** on page 161.

Users of other programming languages must define object variables to be pointers to the data structure representing SOM objects; this 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.

Note: Do not end a class name with the letter "C." The class name that is used with a method name is generated as *classNameClassData*. If a class name ending in "C," such as myprogC, is used, the name generated would be myprogCClassData. Since ClassData and CClassData are different, the compiler would generate the wrong type of export.

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

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

Using classNameNew: 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

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, invoke **somFree Method** to free it:

```
_somFree(obj);
```

After uninitializing the object by invoking **somDestruct Method** on it, **somFree** calls the class object for storage deallocation. 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 classNameRenew: After verifying that the *className* class object exists, the *className***Renew** macro invokes the **somRenew** method on the class object. *className***Renew** is used only when the space for the object has been allocated previously. (Perhaps the space holds an old, no longer needed, uninitialized object.) This macro converts the given space into a new, initialized instance of *className* and returns a pointer to it. You must ensure that the argument of *className***Renew** points to a block of storage large enough to hold an instance of class *className*. You can invoke the **somGetInstanceSize Method** 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.

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 allocating memory separately.

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

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. See **Creating a Class Object** on page 91. Once the class object has been created, the example invokes the **somGetInstanceSize Method** on this class to determine the size of a `Hello` object, uses **SOMMalloc Function** 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 the method used to allocate the storage. The typical method pairs are:

- If an object was originally initialized using the *className***New** macro, the client should use the **somFree Method** on it.
- 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**. However, first invoke **somDestruct Method** to deinitialize the objects in the region to be freed. This allows each object to free any memory that may have been allocated without the programmer's knowledge.

Note: In the **somDestruct** method call above, the first zero indicates that memory should not be freed by the class of the object; you must do it explicitly. The second zero indicates that the class of the object is responsible for overall control of object uninitialization. See **Initializing and Uninitializing Objects** on page 195.

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. See **Initializing and Uninitializing Objects** on page 195**.** 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();
```

For convenience, pointers to SOM objects created using the new operator can be freed using the delete operator. You can also use the **somFree Method** on page 154:

```
delete obj;
obj1->somFree;
```

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 described for the C bindings, first create the class object using the *className***NewClass** procedure or the **somFindClass Method**. See **Using Class Objects** on page 90.

The **somNew** method invoked on the class object creates a new instance of the class, initializes the object using **somDefaultInit Method**, and then returns a pointer to the new object. The C example below 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 */
}
```

Free an object created using the **somNew** method 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. You can use **somGetInstanceSize Method** to determine the amount of memory required. 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. For example, when the name of the class is specified by user input. However, the *className***New** macro (for C) and the **new** operator (for C++) can be used only for classes whose header file is included in the client program at compile time.

An object created using the **somRenew** method should be freed by the client program that allocated its memory, 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 also 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 or C++ and other languages and then describes more specialized situations.

Making Typical Method Calls

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

```
_methodName (receiver, args)
```

The method name is preceded by an underscore (_). 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 other arguments to the method. You can use this expression where 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 and a value of type (environment *). The environment data structure, specified by CORBA, passes environmental information between a caller and a called method. For example, it returns exceptions. For more information, see **Exceptions and Error Handling** on page 100.)

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. 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. When invoking the method, this argument must immediately follow immediately 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 do not supply the (**Environment ***) and **context** arguments when invoking the method. The receiver of the method call is followed immediately by any arguments to the method. Some of the classes supplied in the SOMobjects Developers Toolkit, including **SOMObject**, **SOMClass** and **SOMClassMgr**, are defined in this way to ensure compatibility with previous releases of SOM. *Programmer's Reference for SOM and DSOM* specifies when to use these arguments for each method.

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. Do not use a **somNew** method call or a macro call of *className***New** as the first argument to a C method call because it creates two new class instances rather than one.

Enter any additional arguments required by a method, as specified in IDL following the initial, required arguments to a method (the receiving object, the **Environment**, if any, and the context, if any), as specified in IDL. For a discussion of how IDL **in**, **out** or **inout** argument types map to C/C++ data types, see **Parameter List** on page 130.

Short form versus 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) is not provided by the bindings, because the macro is ambiguous. 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 **Using va_list Methods** on page 80), a you can 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. Use only the long form to write code that can be easily ported to other vendor platforms that support the CORBA standard. 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, you usually know what type an object is expected to have. If you do not know, but the different methods have the same signature, invoke the method using name-lookup resolution, as described in 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 such as exceptions between a caller and a called method. See **Exceptions and Error Handling** on page 100.

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 do not supply the (**Environment ***) and context arguments when the method is invoked. Some of the classes supplied in the SOMobjects Developers Toolkit (including **SOMObject**, **SOMClass** and **SOMClassMgr**) are defined in this way, to ensure compatibility with the previous release of SOM. *Programmer's Reference for SOM and DSOM* 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 **Parameter List** on page 130.

For non-C or C++ programs: 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, you 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 and the **context** argument, if necessary, and the remainder of the method's arguments. The means for calling a procedure, given a pointer returned by **somResolve**, is language-specific. See the section above for C programs. 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.

You must know where to find the method token to use **somResolve** for the desired method. Method tokens are available from class objects that support the method (with the **somGetMemberToken Method**), 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 you can access the method token for method *methodName* introduced by class *className* with:

```
classNameClassData.methodName
```

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

To use offset resolution to invoke methods from a programming language other than C or C++, 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 Function** 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 Function**. 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

Programmer's Reference for SOM and DSOM for more information on **somResolve** and **somClassResolve**.

If you do not know at compile time which class introduces the method to be invoked, or if you cannot directly access method tokens, then use the **somResolveByName Function** 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 in **Method Name or Signature Unknown at Compile Time** on page 90.

Accessing Attributes

In addition to methods, SOM objects can have attributes. An attribute is an IDL shorthand for declaring methods. It 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.

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

```
#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, are described in the documentation of each class in *Programmer's Reference for SOM and DSOM*.

Using va_list Methods

SOM supports methods whose final argument is a **va_list**. A **va_list** is a data type whose representation depends on the operating system platform. To aid construction of portable code, SOM supports a platform-neutral API for building and manipulating **va_lists**. Use of this API is recommended on all platforms because it is both compliant with the ANSI C standard and portable.

A function to create a **va_list** is not provided. Instead, you can declare local variables of type **somVaBuf** and **va_list**.

Use the following sequence of calls to create and destroy a **va_list**:

- **somVaBuf_create**

  Creates a SOM buffer for variable arguments from which the **va_list** will be built.

- **somVaBuf_add**

  Adds an argument to the SOM buffer for variable arguments.

- **somVaBuf_get_valist**

  Copies the **va_list** from the SOM buffer.

- **somVaBuf_destroy**

  Releases the SOM buffer and its associated **va_list**.

- **somvalistSetTarget**

  Modifies the first scalar value on the **va_list** without other side effects.

- **somvalistGetTarget**

  Gets the first scalar value from the **va_list** without other side effects.

Detailed information on these functions is provided in *Programmer's Reference for SOM and DSOM.*

Examples of va_list usage: The following code segments pass a **va_list** to the **somDispatch** method by using the SOMobjects functions that build the **va_list**.

The **somDispatch** method (introduced by **SOMObject)** is a useful method whose final argument is a **va_list**. Use **somDispatch** to invoke some other method on an object when usage bindings for the dispatched method are unavailable or the method to be dispatched is unknown until run time. The **va_list** argument for **somDispatch** holds the arguments to be passed to the dispatched method, including the target object for the dispatched method.

For C:

```
#include <somobj.h>
void f1(SOMObject obj, Environment *ev)
{
     char *msg;
     va_list start_val;
     somVaBuf vb;
     char *msg1 = "Good Morning";
     vb = (somVaBuf)somVaBuf_create(NULL, 0);
     somVaBuf_add(vb, (char *)&obj, tk_pointer);
              /* target for _set_msg */
     somVaBuf_add(vb, (char *)&ev, tk_pointer)
              /* next argument */
     somVaBuf_add(vb, (char *)&msg1, tk_pointer);
              /* final argument */
     somVaBuf_get_valist(vb, &start_val);
     /* dispatch _set_msg on object */
     SOMObject_somDispatch(
```

```
        obj,     /* target for somDispatch */
        0,       /* says ignore dispatched method result */
        somIdFromString("_set_msg"),
                 /* the somId for _set_msg */
        start_val);   /* target and args for _set_msg */
        /* dispatch _get_msg on obj: */
        /* Get a fresh copy of the va_list */
        somVaBuf_get_valist(vb, &start_val);
        SOMObject_somDispatch(
        obj,
        (somToken *)&msg,
            /* address to store dispatched result */
        somIdFromString("_get_msg"),
        start_val); /* target and arguments for _get_msg */
        printf("%s\n",msg);
        somVaBuf_destroy(vb);
    }
```

For C++:

```
    #include <somobj.h>
    void f1(SOMObject obj, Environment *ev)
    {
        char *msg;
        va_list start_val;
        somVaBuf vb;
        char *msg1 = "Good Morning"
        vb = (somVaBuf)somVaBuf_create(NULL, 0);
        somVaBuf_add(vb, (char *)&obj, tk_pointer);
                      /* target for _set_msg */
        somVaBuf_add(vb, (char *)&ev, tk_pointer);
                      /* next argument  */
        somVaBuf_add(vb, (char *)&msg1, tk_pointer);
                      /* final argument */
        somVaBuf_get_valist(vb, &start_val);

        /* dispatch _set_msg on obj:  */
        obj->SOMObject_somDispatch(
        0,   /* says ignore the dispatched method result */
        somIdFromString("_set_msg"),
                      /* the somId for _set_msg */
        start_val);
                      /* the target and arguments for _set_msg */

        /* dispatch _get_msg on obj:  */
```

```
        /* Get a fresh copy of the va_list  */
        somVaBuf_get_valist(vb, &start_val);
        obj->SOMObject_somDispatch(
             (somToken *)&msg,
                      /* address to hold dispatched method result */
             somIdFromString("_get_msg"),
             start_val);
                      /* the target and arguments for _get_msg */
             printf("%s\n", msg);
        somVaBuf_destroy(vb);
   }
```

As a convenience, you can invoke methods whose final argument is a **va_list** from C and C++ by using the short form of method invocation and specifying a variable number of arguments in place of the **va_list**. That is, beginning at the syntax position where the **va_list** argument is expected, SOMobjects interprets all subsequent arguments as being the components of the **va_list**. This is illustrated below, using the **somDispatch** method.

As an example of using the variable-argument interface to **somDispatch**, the following code segments illustrate how an example of attribute access (in **Accessing Attributes** on page 80) could be recoded to operate without usage bindings for the `Hello` class. These code segments are expressed as functions that accept an argument of type **SOMObject** under the assumption that bindings for `Hello` are not available. This requires usage bindings for **SOMObject,** which are also required for calling **somDispatch**.

For C:

```
#include <somobj.h>
void f1(SOMObject obj, Environment *ev)
{
   char *msg;
   /* dispatch _set_msg on obj:  */
   _somDispatch(
       obj,                 /* the target for somDispatch */
       0,     /* says ignore the dispatched method result */
      somIdFromString("_set_msg"),
                               /* the somId for _set_msg */
      obj,                    /* the target for _set_msg */
      ev,           /* the other arguments for _set_msg */
      "Good Morning");
   /* dispatch _get_msg on obj:  */
   _somDispatch(
       obj,
       (somToken *)&msg,
             /* address to hold dispatched meth result */
       somIdFromString("_get_msg"),
       obj,   /* the target for _get_msg */
       ev);   /* the other argument for _get_msg */
   printf("%s\n", msg);
```

```
}
```

For C++:

```
#include <somobj.xh>
void f1(SOMObject *obj, Environment *ev)
{
   char *msg;
   /* dispatch _set_msg on obj:  */
   obj->somDispatch(
       0,  /* says ignore the dispatched method result */
       somIdFromString("_set_msg"),
                                  /* dispatched method id */
       obj,                   /* the target for _set_msg */
       ev,           /* the other arguments for _set_msg */
       "Good Morning");
   /* dispatch _get_msg on obj:  */
   obj->somDispatch(
       (somToken *)&msg,
                 /* address to store dispatched result */
       somIdFromString("_get_msg"),
       obj,
       ev);
   printf("%s\n", msg);
}
```

C programmers must be aware that the short form of the invocation macro that is 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. See **Short form versus long form** on page 77.

Using Name-Lookup Method Resolution

C or C++ programs: Offset resolution is the most efficient way to select the method procedure appropriate to a given method call. However, client programs can invoke a method using name-lookup resolution instead of offset resolution. The C and C++ bindings for method invocation use offset resolution, but methods defined with the **namelookup** SOM IDL modifier result in C bindings where the short form invocation macro uses name-lookup resolution. For C and C++ bindings, a special **lookup_***methodName* macro is defined.

Name-lookup resolution is appropriate when you know at compile time which arguments will be expected by a method (that is, its signature), but do not know the type of the object on which the method will be invoked. For example, use name-lookup resolution when two different classes introduce different methods of the same name and signature, and you do not know 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 **Method Resolution** on page 183.

C only: To invoke a method using name-lookup resolution, when using the C bindings for a method that has been implemented with the **namelookup** modifier, 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 use the first form shown above when supplying a variable number of arguments. Use the second form when supplying a **va_list** argument in place of the variable number of arguments.

C++ only: To invoke a method using name-lookup resolution, when using the C++ bindings for a method that has been defined with the **namelookup** modifier, 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 is used when supplying a variable number of arguments. 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.

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

1. Use the **somResolveByName Function** or any of the **somLookupMethod Method, somFindMethod** or **somFindMethodOk** to obtain a pointer to the procedure that implements the desired method.
2. Then, invoke the desired method by calling that procedure, passing the method's intended receiver, the **Environment** pointer and the context argument if needed, and any method arguments.

The **somLookupMethod, somMethodOK** 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 *Programmer's Reference for SOM and DSOM*.

Note: There are many ways to acquire a pointer to a method procedure. Once this is done, you must make appropriate use of this procedure.

- The procedure should be used only on objects for which it is appropriate. Otherwise, run-time errors are likely to result.
- When the procedure is used, you must give the compiler the 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 NT, SOM method procedures must be called with **__stdcall** linkage. 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. You can use this type name when you want to use a procedure to invoke a method. However, you must have access to the usage bindings for the class containing the method because that is where the type is defined. The type is named **somTD_***className_methodName*. This is illustrated in the following example, and further details are provided in **Obtaining a Method's Procedure Pointer** on page 88.

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, you can always 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 **Figure 3** illustrates the class hierarchy for this example.

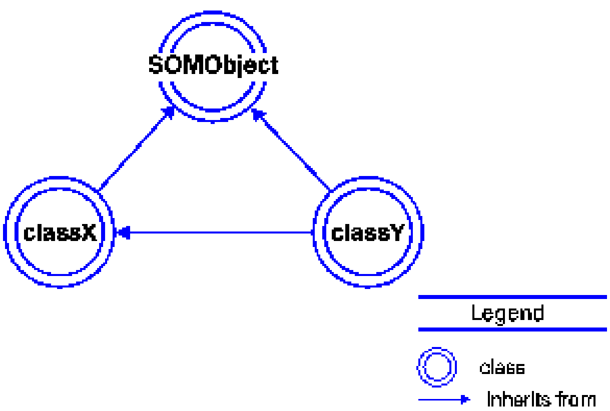

Figure 3. Name-Lookup Resolution

Following is a C++ generic procedure that uses name-lookup method resolution to invoke the reduce method on its argument, that may be either `classX` or `classY`. 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`.

In any case, if the reduce method in `classX` were defined as a **namelookup** method, the following code would be appropriate. 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` does not need to be 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 Function** 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 through 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 **Method Resolution** on page 183 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 Macro** and **SOM_ResolveNoCheck Macro**. 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.

The following C example uses **SOM_Resolve** to obtain a method procedure pointer for method `sayHello`, introduced by class `Hello`, and using it to invoke the method on `obj`. 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 *Programmer’s Reference for SOM and DSOM* 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 Function** (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 *Programmer's Reference for SOM and DSOM*.

Method Name or Signature Unknown 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.

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:

- placing the arguments in a variable of type **va_list** at run time
- using the **somGetMethodData Method** followed by use of the **somApply Function** or 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 *Programmer's Reference for SOM and DSOM*.

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 the **somGetClass Method**. 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. **Getting Information about a Class** on page 96 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 cases, SOM provides the **SOM_GetClass Macro**. In general, the **somGetClass** method and the **SOM_GetClass** macro may have different behavior (if **somGetClass** 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 classNameRenew 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 **somRenew** 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 **Creating Instances of a Class** on page 72). 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 classNameNewClass: 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

`classNameClassData.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 when not using the C/C++ language bindings for the class, or when the class name is not known at compile time:

- 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 Method** 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. 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 invoked on the class manager, 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 Method** to get the name of the library file containing the class. If the class was defined with a **dllname** class modifier, then **somLocateClassFile** 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 **somFindClsInFile** 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.

On AIX and 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 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.

Invoking methods without corresponding class usage bindings: This topic builds on the preceding discussion, and illustrates how a client program can apply dynamic SOM mechanisms to utilize classes and objects for which specific usage bindings are not available. This process can be applied when a class implementor did not ship the C/C++ language bindings. Furthermore, the process allows more programming flexibility, because it is not necessary to know the class and method names at compile time in order to access them at run time. (At run time, however, you must be able to provide the method arguments, either explicitly or with a **va_list**, and provide a generalized way to handle return values.) As an example application, a programmer might create an online class viewer that can access many classes without requiring usage bindings for all those classes, and the person using the viewer can select class names at run time.

As another aspect of flexibility, a code sequence similar to the following C++ example could be re-used to access any class or method. After getting the **somId** for a class name, the example uses the **somFindClass** method to create the class object. The **somNew** method is then invoked to create an instance of the specified class, and the **somDispatch** method is used to invoke a method on the object.

```
#include <stdio.h>
#include <somcls.xh>
int main()
{
  SOMClass *classobj;
  somId tempId;
  somId methId;
  SOMObject *s2;
  Environment * main_ev = somGetGlobalEnvironment();
  tempId = SOM_IdFromString("myClassName");
  classobj = SOMClassMgrObject->somFindClass(tempId,0,0);
  SOMFree(tempId);
  if (NULL==classobj)
  {
    printf(
```

```
      "somFindClass could not find the selected class\n");
   }
   else
   {
      s2 = (SOMObject *) (classobj->somNew());
      methId = somIdFromString("sayHello");
      if (s2->somDispatch((somToken *) 0, methId, s2, ev))
        printf("Method successfully called.\n");
   }
   return 0;
}
```

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** on page 90.

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**. **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.

Note: If you are building an application that uses a combination of C and C++ compiled object modules, then the C++ linker must be used to link them.

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 or Windows NT, for C programmers

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

Under OS/2 or Windows NT, for C++ programmers

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

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 or Windows NT:

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

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.

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.]

somLPrintf Function
: Combines **somPrefixLevel Function** and **somPrintf Function.** 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).

somPrefixLevel Function
: Generates (by **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).

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 through **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.

See *Programmer's Reference for SOM and DSOM* 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 Method
: 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 Method
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 Method
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 Method
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 Method
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 Method
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 Method
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 Method
Obtains the name of a class. The receiver of the method is the class object. The method returns the class name.

somGetNumMethods Method
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 Method
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 Method
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 Method

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 Method

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** returns TRUE if the method is supported, or FALSE otherwise.

See *Programmer's Reference for SOM and DSOM* 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.

somGetClass Method

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 Method

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 Method

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 Method

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 Method

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.

somIsObj Function

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.

somRespondsTo Method

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. The **somRespondsTo** method returns TRUE if the object supports the method, or FALSE otherwise.

See *Programmer's Reference for SOM and DSOM* 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.

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: **SOM_WarnMsg Macro**, **SOM_TestC Macro** and **SOM_Expect Macro**
- **SOM_AssertLevel** controls the behavior of the **SOM_Assert Macro**.

Available macros for generating debugging output are as follows:

```
classNameMethodDebug
```

(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 classNameMethodDebug(c,m)\
        SOM_NoTrace(c,m)
```

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

SOM_TestC Macro

SOM_TestC 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 Macro

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

SOM_Assert Macro
: **SOM_Assert** 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 Macro
: **SOM_Expect** 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.

somDumpSelf Method and **somPrintSelf Method** can be useful in testing and debugging. **somPrintSelf** produces a brief description of an object, and **somDumpSelf** produces a more detailed description. See *Programmer's Reference for SOM and DSOM* 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. 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.

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 ending in a digit 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 ending in a digit 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 ending in a digit 9 belong to this category.

The codes for errors detected by SOM are listed in **Appendix A, Error Codes** on page 397.

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 Macro
: **SOM_Error** 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.)

SOM_Test Macro
: **SOM_Test** 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 *Programmer's Reference for SOM and DSOM* for more information on a specific macro.

Other classes provided by the SOMobjects Developer Toolkit (including those in DSOM, the Interface Repository framework and the utility classes and metaclasses) handle errors differently. Rather than invoking **SOMError** with an error code, their methods return exceptions by 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.

Introduction to Exceptions

SOMobjects follows the CORBA model for exception handling. In this model the method caller receives error information back from the method invocation in a data structure called the Environment. This is different from the *catch/throw* model where an exception is implemented by a long jump or a signal.

CORBA defines two types of exceptions:

USER_EXCEPTION
: Explicitly declared in IDL files. Every method that returns a user exception contains a raises keyword listing the exceptions it may return.

SYSTEM_EXCEPTION
: Implicitly defined. Any method may return these exceptions without listing them on a raises keyword. System exceptions are sometimes called standard exceptions.

User Exceptions

In SOM Interface Definition Language, a method may be declared to return zero or more exceptions. Each type of exception has 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 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.

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 the **SOMMalloc Function** to allocate a `BAD_FLAG` structure, initialize it with the appropriate error information, and make a call to the **somSetException Function** 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.

System 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 **Exception Declarations** on page 125. 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.

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).

Because 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 primarily used to return exception data to the client following a method call.

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 **SOM_InitEnvironment Macro**), 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,

```
typedef enum exception_type {
   NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION,
exception_type_MAX=214783647
} 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 must be allocated by **SOMMalloc**:

The **somSetException Function** expects the params argument to be a pointer to a structure that was allocated using **SOMMalloc Function**. 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 next call to **somSetException**), or when the **somExceptionFree Function** is called.

somSetException simply sets the exception value; it performs no exit processing. If there are multiple calls to **somSetException** before the method returns, the caller sees only 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 Function** call) from an Environment struct with the following functions:

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

The **somExceptionId Function** returns the exception name, if any, as a string. The **somExceptionValue Function** 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 the **somGetGlobalEnvironment Function**.

The **somExceptionFree Function** frees any memory in the Environment associated with the last exception. This function does only a shallow **SOMFree** of the Environment's exception parameters. It does not walk the exception parameters, freeing any nested memory blocks. For information on managing the memory, objects and exceptions used by DSOM applications, see **Memory-Management Functions** on page 256.

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

You can also use the CORBA **exception_free** API to free the memory in an Environment structure.

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 */
```

Example Of Raising an Exception

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 maps 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
      parameter */
   _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 may 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;
   }
   /* ... */
}
```

The Error Log Facility

SOMobjects supports an error log to record exceptions and error conditions that may occur within the SOMobjects services. DSOM and all the Object Services use this facility.

Most of the data recorded in the error log is formatted to help you debug new applications. The remaining data is not formatted and can be used by support personnel to help diagnose more complicated problems.

The error log records only exceptions and errors that occur within code shipped with the SOMobjects Developer Toolkit. If you wish to record error information from the frameworks and applications that you develop, you must provide your own facility to do this.

The error log is implemented as a wrapping log file. Once it has filled up, the oldest entries are destroyed as new entries are added. All processes on the same system should be set up to share a single error log file for greater ease in solving any multi-process interaction problems that might occur. If you are using DSOM at multiple locations, you have multiple error logs. Error log files are located in the directory pointed to by the **SOMDDIR** configuration variable, under the [somd] stanza, in the configuration file.

Configuring the Error Log

There are four variables in the configuration file, under the [somras] stanza, which you can set to customize the operation of the error log.

Name of the Error Log File

The SOMErrorLogFile variable controls the name of the error log file. The default setting of SOMErrorLogFile is SOMERROR.LOG.

Size of the Error Log

The SOMErrorLogSize variable controls the maximum size of the error log file. The default size is 128 which lets the error log file grow to 128 kilobytes before it begins to wrap. The default size allows room for several hundred average log entries.

Type of Information To Record

The SOMErrorLogControl variable lets you filter the information to record in the Error Log. The severity of errors logged, for example, INFO, WARNING, or ERROR can be individually selected. You can use the MAPPED_EXCEPTION value to select whether to record a message each time an object service maps an exception into a different exception. This often occurs as an exception raised by an object service ripples back up through the object system to the application.

The default setting of SOMErrorLogControl is WARNING ERROR MAPPED_EXCEPTION which records errors with severities of WARNING and ERROR, as well as all mapped exceptions. The default setting does not record errors with severity INFO. You may specify any combination of the control values. To specify multiple values, as the default setting does, separate each value by one or more blank spaces.

Display Error Messages

The SOMErrorLogDisplayMsgs variable controls whether to also display the error message to the standard output device each time an Error Log entry is made. The displayed messages do not include any extended log data collected for service personnel. The default setting of SOMErrorLogDisplayMsgs is YES. The default setting logs errors and displays the optional messages. This setting is helpful while you are debugging new applications.

Using The Error Log

Much of the data in the Error Log is formatted. You can use a text editor to display the contents of the error log file. The top lines of the file contain the name of the host system that this error log file is from, the operating system, and the number of the last log entry written in the file.

To find the last error log entry in the file look for the `LASTENTRY`: *nnnn* line in the information at the top of the file. You can then use the search feature of your text editor to locate this entry *nnnn* in the log file. Because the error log wraps around, this entry may be anywhere in the file.

Understanding Error Log Entries

Each error log entry starts with a message area and may contain extended log data. The message area contains the following free-form fields:

- The error log entry number enclosed in brackets. Error log entry numbers can range from 0 to 9999. When the error log entry number reaches 9999 it wraps back to 0 and starts over again.
- A date and time stamp telling the system date and time when the error log entry was made.
- `PID: 0X`*nnnn* is the process identifier of the code that made the error log entry call.
- `TID: 0X`*nnnn* identifies the thread, within the process, that made the call.
- `Request from` *clientX* `on` *hostY* identifies the client name and host name where the operation request originated. Normally this information is retrieved from the Principal object that accompanies each DSOM request. If the Principal object is not available, this portion of the message may say `Request from unknown client on unknown host`. You can use this information to help debug servers where requests are coming from different hosts and clients.

The second part of an error log entry is the optional extended log data. If included, this part of the entry contains hexadecimal data followed by an ASCII translation of the data. Most of this data is useful only to service personnel. However, some of the action lists in **Appendix A, Error Codes** on page 397 may point out information in this data that will be useful to you.

A complete error log entry that contains both a message and extended log data may look like the following.

```
{1234} Thur Oct 12 13:02:46 cst 1995 PID:0XF0A2 TID:0X000B
Raised SYSTEM_EXCEPTION UNKNOWN with severity WARNING at
                                        somutil.c:1254.
Request from clientX on hostY.
Error code is 20199[SOMERROR_BadArgument].
e2 00 00 00 21 00 44 6f 67 43 6C c1 73 73 00 00 d2
[....!.DogClass...]
00 24 00 00 4d 79 46 61 63 74 6f 72 79 00 00 ff ff
[.$..MyFactory....]
```

```
00 00 00 00 00 ed ed 00 00 ef fe 00 00 00 00 00 00
[.................]
00 3e 00 00 40 00 00 2a 00 00 00 00 ff ff ff ff ed
[.>..@..*.........]
```

If the extended log data is quite large, you may see continuation blocks that begin with {+*nnnn*}, where *nnnn* is the same number used at the beginning of the log entry. The plus symbol (+) indicates this is a coniuation block for the same long entry. A continuation block for the above example would begin:

```
{+1234} Thur Cot 12 13?02:46 cst 1995 PID 0XF0A2 TID: 0X000B
```

It is possible for one log entry to have several continuation block.

The Standard Error Messages: The following list shows the four basic message types that may be logged. The examples do not include the optional extended log data.

```
{1234} Thur Oct 12 13:02:46 cst 1995 PID:0X0512 TID:0X0837
Raised SYSTEM_EXCEPTION UNKNOWN with severity WARNING at
                                         somutil.c:1254.
Request from clientX on hostY.
Error code is 20199[SOMERROR_BadArgument].
```

This type of message is logged by an object service when it raises a new exception (that is, records an exceptional condition into the environment structure). This message includes an error code. See **Appendix A, Error Codes** on page 397 to determine its meaning and actions to take to correct the problem. This type of message also contains a level indicator. See **Level of Errors** on page 110 for more information.

```
{4321} Sat Oct 14 13:02:46 cst 1995 PID:0X9909 TID:0X0787
Mapped USER_EXCEPTION WrongTransaction to SYSTEM_EXCEPTION BAD_PARAM
                                         at somtr.c:231.
Request from clientX on hostY.
New Error code is 00001[Exception_Data_Logged].
```

This type of message indicates that an object service received an exception from a sub-service method call and has mapped the original exception into a new exception. This mapping is often required because services can raise only User Exceptions listed on their interface descriptions or Standard CORBA Exceptions. You may follow the trail of mapped exceptions to understand if the exception received by the application code has been re-mapped from the object service exception that was originally raised. If re-mapping has occurred, the action list for the original exception, that is, the one with message type (1) above, may often contain more helpful information for resolving the problem.

```
{0089} Thur Oct 12 19:06:39 cst 1995 PID:0XF0A3 TID:0X001B
Process abnormally terminated at somutil.c:142.
Request from clientX on hostY.
```

This type of message indicates an error situation that prevented the object service from operating reliably.

```
{1523} Sat Oct 14 02:23:16 cst 1995 PID:0XF003 TID:0X0015
Service data collected with severity ERROR at somderr.c:523.
Request from clientX on hostY.
```

The action lists in **Appendix A, Error Codes** on page 397 might provide useful information. This type of message might indicate that the object service logged data for use by service personnel.

Extended Error Messages: Each object service may also supply some unique extended error messages. Object service extended error messages always begin with the text from one of the four standard message types and then append service specific information to the end of the message.

The object service extended message below illustrates an extension of message type 1.

```
{1425} Thur Oct 12 11:48:26 cst 1995 PID:0XF0A3 TID:0X001B
Raised SYSTEM_EXCEPTION BAD_PARAM with severity WARNING at
                                         somp.c:1325.
Request from clientX on hostY.
Error code is 50123[Bad_Persistent_ID].
Persistent ID received from client was CustomerPID0005.
```

In this extended error message example `Persistent Id received from client was CustomerPID0005.` is the object service unique extension that was added to a standard message.

Level of Errors: Some error log entries contain an indication of the error level. The following error levels are used.

- INFO indicates a problem which is usually handled within the SOMobjects code and probably does not affect your application.
- WARNING indicates a problem that may cause your application to not function properly. Warning messages cannot be ignored. You must resolve these errors for your application to function properly.
- ERROR indicates a problem that most likely prevents your application from functioning properly.

Locating the Correct Log File

In order to use the Error Log effectively you must first determine which systems' log file you need to look at. If your application invokes methods on server processes running in remote systems, you may have to examine the log files on the server systems to determine what error occurred. It may be helpful to keep SOMErrorLogDisplayMsgs set to YES on each of the server systems to display error messages while you are debugging your new application. This lets you determine which system originally raised an exception. Once you have identified the system which raised the exception, you can look up the error code in **Appendix A, Error Codes** on page 397 and use the action list to help you resolve the problem.

Memory Management

The SOMobjects Developer Toolkit provides several functions 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 the **SOMError Function**. 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 Function, SOMCalloc Function, SOMRealloc Function and **SOMFree Function** 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.

Clearing Memory for Objects

The memory associated with objects initialized by a client program must also be freed by the client. The SOM-provided **somFree Method** 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:

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

For information on managing the memory, objects and exceptions used by DSOM applications, see **Memory-Management Functions** on page 256.

SOM Manipulations Using somId

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 somIds 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 Function
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 the **SOMFree Function**.

somStringFromId Function
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 Function
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 two **somId** values represent the same string, or FALSE (0) otherwise.

somCheckId Function
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 Function
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 Function
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 Function
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.

somSetException Function
Indicates how many unique somIds 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 **somEnvironmentNew Function** 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 Function
somEndPersistentIds Function
Delimit a time interval for the current thread during which it is guaranteed that:

- any new **somId** values that are created will refer only to static strings
- these strings will not be subsequently modified or freed.

These functions are useful because **somIds** that are registered within a *persistent ID interval* can be handled more efficiently.

See *Programmer's Reference for SOM and DSOM* for more information on a specific function.

Chapter 5. SOM Interface Definition Language

This chapter discusses how to define SOM classes. 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 in **Chapter 7, Implementing Classes in SOM** on page 171.) At this point, the implementation file can be compiled and linked with a client program that uses it (as described in **Chapter 4, Using SOM Classes in Client Programs** on page 69).

Syntax for SOM IDL is presented in this chapter, along with helpful information for using them correctly.

Interface versus 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 attributes and the signatures of its methods. 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 interface to instances of the class: the interface definition file (or **.idl** file)
- Method procedures 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 NT) 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:
  - in the class's implementation file
  - 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 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.

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 **SOM IDL Language Grammar** on page 421. Instructions for converting existing OIDL-syntax files to IDL are given in **Converting OIDL Files to IDL** on page 417. The current section describes the syntax and semantics of SOM IDL using the following conventions:

Literals (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 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.

any	FALSE	readonly
attribute	float	sequence
boolean	implementation	short
case	in	string
char	inout	struct
class	interface	switch
const	long	TRUE
context	module	TypeCode
	octet	

Table 1. Keywords for SOM IDL

A typical IDL specification for a single class, residing in a single **.idl** file, has the following form. (See **Defining Multiple Interfaces in a .idl File** on page 150.) 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 and Constant Declarations (optional)

Exception Declarations (optional)

Interface Declarations (optional)

Module declaration (optional)

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. 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} .. $2^{15}-1$

long -2^{31} .. $2^{31}-1$

unsigned short 0 .. $2^{16}-1$

unsigned long 0 .. $2^{32}-1$

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.

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 9, The Interface Repository Framework** on page 337 describes TypeCodes and their associated functions. For extensive examples, see **Using the IDL Basic Type any** on page 351.

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 just as they are in C and C++, 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;              /*  Tag myStruct is the same */
   double y;            /*  as the type name below;  */
} 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.

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 : SOMObject{
      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 : SOMObject{
      typedef char char_t;
```

```
      };
    };
  };
```

In the C/C++ bindings of the preceding fragment, you can refer to:

`M_long_t` as `long_t`

`M_N_short_t` as `short_t`

`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;        /* Integer ’1’ can be converted */
  case 2: float y;       /* to the switch type long      */
  default: char z;
};
```

is mapped onto the following C struct:

```
typedef struct Hello_Foo {
  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:

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

Note: This example is from *Common Object Request Broker: Architecture and Specification*.

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 number of elements 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 null-terminated character arrays. The length of the string is encoded by the position of the null (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*.

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.

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 more explanation, see **Declaring Object Variables** on page 71.) 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.**Method Declarations** on page 130 describes how to indicate that a particular method may raise a particular exception, and **Exceptions and Error Handling** on page 100 describes how exceptions are handled.

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 in **Standard Exceptions Prescribed by OMG** on page 126. The 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 in **Standard Exceptions Prescribed by OMG** on page 126 are defined in an IDL module called **StExcep**, in the file called **stexcep.idl**, and the C definitions can be found in **stexcep.h**.

Standard Exceptions Prescribed by OMG

OMG publishes many standards, of which CORBA is only one. In version 3.0, we have implemented the Transaction Service from OMG. Therefore, CORBA and Transaction Service exceptions are proper subsets of OMG.

```
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};


//CORBA-defined standard exceptions
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

//Transction Service standard exceptions
exception TransactionRequired ex_body;
                                 //operation requires transaction
exception TransactionRolledBack ex_body;
                                 //current transaction has rolled
back
exception InvalidTransaction ex_body;
                                 //transaction invalid or invalid
state
exception WrongTransaction ex_body;
                                 //reply received for wrong
transaction
};
```

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)
};
```

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 #included 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 a discussion of multiple interface statements, see **Defining Multiple Interfaces in a .idl File** on page 150.

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 discussion of name recognition in SOM, see **Scoping and Name Resolution** on page 151.

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, **_get_***attributeName*) and one to set the value of the attribute (a **set** method, **_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 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 method names (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.

in, out, inout Parameters: 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**). 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();
```

A method should not modify an **in** parameter. This is important, because any changes may be visible to clients and are unexpected, given the **in** designation. If a change must be made, the parameter should first be copied and only the copy modified. The **pass_by_copy_parameters** modifier can be used for this, so that SOMobjects will make a copy automatically. See **Passing Parameters by Copying** on page 146.

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.

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**. (See **Using va_list Methods** on page 80.) 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 **Modifier Statements** on page 133.)

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 **Exception Declarations** on page 125 for information on defining exceptions and for the list of standard exceptions. See **Exceptions and Error Handling** on page 100 for information on using exceptions.)

Context Expression

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, underscores 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 Method** 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. The CORBA standard allows properties in addition to those in the **context** expression to be passed in the **Context** object.

Invoking Methods on Objects on page 76 describes the placement of a context parameter in a method call.

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 information is assumed.

Because the implementation statement is specific to SOM IDL, the implementation statement should be preceded by an **#ifdef __SOMIDL__** directive and followed by an #endif directive. (See **Example 3. Overriding an Inherited Method** on page 60.)

```
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 declaring instance, terminated by a semicolon. These constructs are described below. An interface statement may not contain multiple implementation statements.

Modifier 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 **SOM Compiler Unqualified Modifiers** is associated with the interface 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 **SOM Compiler Qualified Modifiers** is associated with a qualifier (by connecting them with a colon). The qualified modifier has the following 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 a user-defined term. If the qualifier denotes 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. Qualified modifiers can be defined with the *qualifier* and the *modifier*[=*value*] definition on either side of the colon. 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 {
        somDefaultInit : 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 = somDefaultInit
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 lets users 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 *Programmer’s Reference for SOM Emitter Framework*) and from the Interface Repository (see **Chapter 9, The Interface Repository Framework** on page 337).

SOM Compiler Unqualified Modifiers: Unqualified modifiers include the SOM Compiler-defined identifiers:

abstract
: Specifies that the class is intended for use as a parent for subclass derivations, but not for creating instances.

abstractparents =“parentName, ...”
: Specifies that no implementation will be inherited from the indicated parent class into the new subclass being defined, for all the interfaces inherited from the parent class. The implementations not inherited are instance variables and method implementations. The subclass being defined may inherit these implementations from some other non-abstract parent, but, otherwise, the subclass is responsible for providing implementations for the inherited methods by overriding these methods and providing an appropriate implementation.

At run time, when a class is constructed, abstract inheritance from a parent is requested using the first argument to **somBuildClass**, which is a bit mask with bit *n* set to `0` only if parent *n* is abstract. The implementation bindings generate this argument based on the IDL for a class, and indicate abstract inheritance when the IDL includes an abstractparents modifier statement.

The *parentName* can be one or a comma-separated series of simple names or C_Scoped names. To verify that the implementation bindings emitter correctly recognized the modifier and the parentNames, you can inspect the call to **somBuildClass** in the generated implementation bindings file.

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 **Customizing the Default Base Proxy Class** on page 322.)

Modifiers that name classes that DSOM loads using **somFindClass** (such as **baseproxyclass** and **factory**) must be specified in a form that somFindClass can accept. For classes that are defined within modules, the modifier value must include the module name.

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 complete the initialization of a class object after it is created. The **classinit** modifier is needed if something should happen exactly once when a class is created. (That is, you want to define an action that will not be inherited when subclasses are created. One example of this is for **staticdata** variables.) When the **classinit** modifier is specified in the **.idl** file for a class, the implementation file generated by the SOM Compiler provides a template for the procedure, which includes a parameter that is a pointer to the class. The class implementor can then fill in the body of this procedure template.

directinitclasses
directinitclasses = "*ancestor1, ancestor2, ...*"

Specifies the ancestor class 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** on page 195.

dllname
dllname=*filename*

Specifies the name of the library file that will contain the class's implementation. If *filename* contains special characters, 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 NT is used to locate the file.

When the **def**, **exp**, or **imod** emitter is run, the **dllname** modifier is overridden by use of the SOM Compiler's **-m** option **dll**, described under **Running the SOM Compiler** on page 161.

factory = *className*
Specifies the name of the class's factory. The specified factory will be used to create instances of the target class in a DSOM server. If no factory is specified, the SOM class object will be used. For more information, see **Customizing Factory Creation** on page 302.

**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
functionprefix = *prefix*

Directs the SOM Compiler to construct method 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.

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
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^{32}-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 (using either **object_owns_result** or **object_owns_parameter**). See **Memory-Management Functions** on page 256 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
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^{32}-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 Method** is invoked.

somdeallocate = procedure
Specifies a user-written procedure that will be executed to deallocate memory for class instances when the **somDeallocate 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*].

Classes: **releaseorder**

Attributes: **impldef_prompts**, **indirect, nodata, noget, noset**, **persistent**

Method: **caller_owns_parameters**, **caller_owns_result**, **const**, **dual_owned_parameters**, **dual_owned_result**, **init**, **maybe_by_value_parameters**, **maybe_by_value_result**, **method**, **migrate**, **mplan**, **namelookup**, **nocall**, **noenv**, **nonstatic**, **nooverride**, **noself**, **object_owns_parameters**, **object_owns_result**, **offset**, **override**, **pass_by_copy, procedure**, **reintroduce**, **select**, **suppress_inout_free**

Variables: **staticdata**

Types: **impctx**, **length**, **pointer**, **struct**

caller_owns_parameters

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 the **release Method** on the parameter, rather than the **somFree Method**).

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 the **release Method** on the result, rather than the **somFree Method**).

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.

dual_owned_parameters

dual_owned_parameters="p1, p2, ..., pn"

When invoking the method remotely with DSOM, this modifier indicates that a copy of the named parameters of the specified method is owned by each of the caller and the object, and each is responsible for releasing its own copy, including introduced pointers for **inout** and **out** parameters. When invoking the method on a local (same-process) object, the result of the specified method is owned by the object and should not be freed by the caller. See **Advanced Memory-Management Options** on page 257.

dual_owned_result
When invoking the method remotely with DSOM, this modifier indicates that the result of the specified method is owned by each of the caller and the object, and each is responsible for releasing its own copy. When invoking the method on a local (same-process) object, the result of the specified method is owned by the object and should not be freed by the caller. See **Advanced Memory-Management Options** on page 257.

impctx
Supports types that cannot be fully defined using IDL. The information provided by this modifier is built into the **TypeCode** constructed for the type. See **Using tk_foreign TypeCode** on page 350.

One use of **impctx** is by DSOM to marshal SOMFOREIGN types. For information regarding DSOM's use, see **Passing Foreign Data Types** on page 262

impldef_prompts
Indicates that the DSOM **regimpl** tools should prompt the user to supply a value for an attribute. This modifier can only be used to modify attributes of type string. It is used in the IDL for a subclass of the DSOM **ImplementationDef**. For more information, see **Customizing ImplementationDef Objects** on page 41.

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** on page 195.

length
length=*n*

Specifies the size in bytes of the top-level contiguous storage of a foreign type. The default is 4 bytes. The value of this modifier must be nonzero. This modifier is used by DSOM to marshal SOMFOREIGN types. For more information, see **Passing Foreign Data Types** on page 262.

maybe_by_value_parameters
maybe_by_value_parameters ="p1, p2, ..., pn"

Indicates that, for the named parameters of the specified method, objects passed as parameters are to be passed to the remote site by copy (rather than by reference) if this is possible. See **Passing Objects by Copying** on page 261.

maybe_by_value_result
Indicates that, for the specified method, its returned object is to be passed back to the client by copy (rather than by reference) if this is possible. See **Passing Objects by Copying** on page 261.

method
Indicates the category of method implementation. See **Four kinds of SOM Methods** on page 184 for an explanation of the meanings of these different method modifiers. If none of these modifiers is specified, the default is method. Methods with a procedure modifier cannot be invoked remotely by DSOM.

migrate
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 **Name Usage in Client Programs** on page 151 for an explanation of C-scoped names.

mplan
mplan=none

Directs the SOM Compiler not to generate a marshal plan for the specified method in any emitted **.ih** or **.xih** file. A marshal plan tells DSOM how to invoke the method remotely. By default, the SOM Compiler attempts to generate marshal plans for all methods. It silently abandons the attempt if the signature of the method makes it impossible for the method to be invoked remotely. Specifying **mplan=none** can marginally improve SOM compilation time and DLL size, if you are certain that a method will never be invoked remotely. See **Registering Class Interfaces** on page 30

namelookup
See **offset**.

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** 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** or **.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**.

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** or **.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.

The **set** method procedure that the SOM Compiler generates by default for an attribute in the **.h** or **.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_parameter
object_owns_parameters = "p1, p2, ..., pn"

Specifies the names of the method's parameters whose ownership will be transferred to the target object, which takes responsibility for the parameter storage. The in parameters must be allocated by the client with SOMMalloc; in a remote method call, DSOM "stands in" and frees them with SOMFree. For a remote target object, the inout and out parameters in the client's address space are released when the proxy is released. On the server side, the target object's implementation determines when the associated memory will be freed after the method completes. Object ownership sometimes applies to introduced pointers. For more information, see **Advanced Memory-Management Options** on page 257.

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 of the result sometime before the object is destroyed. For more information, see **Advanced Memory-Management Options** on page 257.

offset
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 **Invoking Methods on Objects** on page 76.) 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**.

pass_by_copy_parameters
pass_by_copy_parameters ="p1, p2, ..., pn"

Indicates that, for the named parameters of the specified method, each parameter will be passed by copy (rather than by reference). See **Passing Objects by Copying** on page 261.

pass_by_copy_result

Indicates that, for the specified method, its returned object will be passed back to the client by copy (rather than by reference). See **Passing Objects by Copying** on page 261.

procedure

See **method**.

pointer

Indicates that a SOMFOREIGN type has the same storage class as a pointer, which influences when pointers are introduced by the mapping from IDL to C/C++. The default is **pointer** unless **struct** is specified. For additional information, see **Passing Foreign Data Types** on page 262. In the same chapter, see **Introduced Pointers** on page 255 showing, by data type, when pointers are introduced by the mapping from IDL to C/C++.

reintroduce

Indicates that this interface will hide a nonstatic or direct-call method introduced by some ancestor interface, and will replace it with another implementation. Only methods introduced as direct-call procedures or nonstatic methods can be reintroduced. Static methods (the default implementation category for SOM methods) cannot be reintroduced.

releaseorder

releaseorder: a, b, c, ...

Specifies the order in which the SOM Compiler will place methods or variables 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 **releaseorder** statement should contain every method name introduced by the class, but should not include any inherited methods. The **get** and **set** methods defined automatically for each new attribute (**_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

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. Using select guarantees inheritance of the selected parent's method implementation, in case some metaclass implementation may have overridden the default inheritance from the left-most parent, or when inheritance from a parent further to the right is desired. The parent is specified using the C-scoped name. For example, use `Module_InterfaceName`, and not `Module::InterfaceName`. See **Name Usage in Client Programs** on page 151 for an explanation of C-scoped names.

staticdata

Indicates that a data variable is not instanced (that is, is not stored within objects), but, instead, that it will be accessed through an external pointer to which client code can be linked. This is similar in concept to C++ static data members, but with one level of indirection. (The indirection is provided to allow SOM objects to be **staticdata**.)

A class implementor has responsibility for allocating the staticdata variable and for loading the external pointer to the staticdata variable during class initialization. The external pointer is located in the **ClassData** structure for the implementing class, in the field: *className***ClassData.***variableName*.

The implementor's responsibility for loading the external pointer(s) can be facilitated by writing a special class initialization function and indicating its name using the **classinit** unqualified modifier (see **Example 3** on page 144).

Attributes can be declared as **staticdata**. This is an important implementation technique that allows classes to introduce attributes whose backing storage is neither instanced nor inherited by subclasses. (See **Example 1** on page 143 and **Example 3** on page 144) **staticdata** attributes are valuable for other reasons as well: they hide the pointer indirection required for their data access, and they are the only *DSOM-safe* mechanism for accessing staticdata variables. For this reason, it is recommended that the **staticdata** modifier be restricted to attributes.

struct

Indicates that a SOMFOREIGN type has the same storage class as a **struct**, which influences when pointers are introduced by the mapping from IDL to C/C++. For additional information, see **Passing Foreign Data Types** on page 262 and see **Introduced Pointers** on page 255 showing, by data type, when pointers are introduced by the mapping from IDL to C/C++.

suppress_inout_free

suppress_inout_free="p1, p2, ..., pn"

Directs DSOM to suppress (for the named parameters of the specified method) the freeing of any part of an **inout** parameter in the caller's address space. This modifier is meaningful only for remote method calls. See **Advanced Memory-Management Options** on page 257.

Example 1: The following example illustrates the specification of qualified modifiers:

```
implementation
{
  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 and z are attributes
  _set_z: object_owns_parameters = "name";
  _get_z: object_owns_result;
  mymod: a, b;
```

```
  releaseorder: op1,op3,op2,op5,op6,x,y,_set_z,_get_z,
                _set_y,_get_y;
};
```

As shown above for attribute `z`, separate modifiers can be declared for an attribute's **_set** and **_get** methods, using method modifiers. This capability may be useful for DSOM applications. (See the DSOM sample program `animal` that ships with SOMobjects.)

Example 2: For this example, class `B`, which is derived from class `A`, originally introduced methods `foo1`, `foo2` and `foo3`. If method `foo2` were migrated to class `A`, the modified class `B` implementation would be as shown:

```
interface B : A {
    void foo1();
                    /* <<-- foo2() has been moved to class A */
    void foo3();
    implementation {
        releaseorder: foo1, foo2, foo3;
        majorversion = 1; minorversion = 2;
        foo2: migrate = A;
    };
};
```

Example 3: This example for classes `X` and `Y` illustrates the use of a **staticdata** modifier, along with its corresponding **classinit** modifier and the template procedure generated for **classinit** by the SOM Compiler.

```
/* IDL for staticdata and classinit example: */
#include <somobj.idl>
interface X : SOMObject {
    attribute long staticAttribute;
    attribute long normalAttribute;
    implementation {
        staticAttribute: staticdata;
        classinit = Xinit;
        releaseorder: staticAttribute,
                      _get_staticAttribute,
                      _set_staticAttribute,
                      _get_normalAttribute,
                      _set_normalAttribute;
    };
};
interface Y : X { };

/* Template procedure for classInit: */
#ifndef SOM_Module_classinit_Source
#define SOM_Module_classinit_Source
#endif
#define X_Class_Source
```

```
#include ”classInit.ih”
static long holdStaticAttribute = 1234;
void SOMLINK Xinit(SOMClass *cls)
{
    XClassData.staticAttribute = &holdStaticAttribute;
}

. . .
main()
{
    X *x = XNew();
    Y *y = YNew();
    somPrintf(”initial staticAttribute = x(%d) = y(%d)\n”,
                               __get_staticAttribute(x,0),
                               __get_staticAttribute(y,0));
    __set_staticAttribute(x,0,42);
    __set_staticAttribute(y,0,4321);

    somPrintf(”changed staticAttribute = x(%d) = y(%d)\n”,
                               __get_staticAttribute(x,0),
                               __get_staticAttribute(y,0));
    __set_normalAttribute(x,0,10);
    __set_normalAttribute(y,0,20);

    somPrintf(”after setting normalAttribute, x(%d) != y(%d)\n”,
                                     __get_normalAttribute(x,0),
                                     __get_normalAttribute(y,0));
}
/* Program output:

   initial staticAttribute = x(1234) = y(1234)
   changed staticAttribute = x(4321) = y(4321)
   after setting normalAttribute, x(10) != y(20)
*/
```

Declaring Instance Variables and Staticdata Variables

Declarators are used within the body of an implementation statement (described in **Implementation Statements** on page 132) 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** on page 118). 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.

Passing Parameters by Copying

Under normal circumstances, the **in** parameters to a method must not be modified by the method. This is important because any changes made by the method's implementation (or callee) may be visible to callers of the method. Moreover, such changes would not be expected, given the **in** designation of the parameter.

For situations where a method does need to modify an **in** parameter, however, the method can receive a copy of the parameter. The IDL modifier **pass_by_copy_parameters** is used to identify parameters that should be copied when passed from the caller of a method to the method's implementation. This parameter-passing style is similar to *pass by value* in C++, and is generally used to ensure that changes made to parameters by the callee are not visible to callers.

The following example demonstrates both parameter passing styles:

```
interface A;
interface B : SOMObject {
    void op(in A a1, in A a2);
    implementation {
        op: pass_by_copy_parameters =a2;
    };
};
```

In this example, method `op` takes two in parameters, both of type `A`. The default parameter passing semantics for objects is by reference, so `a1` is passed to `op` by reference. Parameter `a2`, however, is passed by copy, because modifier **pass_by_copy_parameters** is used to override the default call semantics.

C and C++ usage bindings generated by the SOM compiler automatically make copies of **pass_by_copy_parameters** parameters; thus, callers that use these bindings need not construct copies explicitly. However, if a method is called through the Dynamic Invocation Interface or through a method procedure pointer (as returned by the **somResolve**

Function, for example), then the caller is responsible for copying **pass_by_copy_parameters** parameters.

The designation of **pass_by_copy_parameters** for a method argument does not affect its type in the corresponding C or C++ bindings. For example, clients of `op` should pass, for each parameter, a pointer to an object of type `A`.

C and C++ usage bindings copy **pass_by_copy_parameters** of an object type via the copy constructor **somDefaultCopyInit Method**, as supported by the formal parameter class. The copying of all other types is done via a shallow, top-level copy. For example, the top level of a structure parameter is copied, but no copying is done of any objects that are referenced from fields within the structure.

Modifier **pass_by_copy_parameters** can only be used only with **in** parameters, as it is incompatible with the callee-to-caller passing of parameter values that takes place with **out** and **inout** parameters.

Passthru Statements

A passthru statement (used within the body of an implementation statement, described above) lets a class implementor 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 SOM compiler ignores the contents of the passthru lines which can contain anything that needs to be placed near the beginning of a header file for a class. 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 and so forth) 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.

For any given target file (as indicated by *language_suffix*), only one passthru statement may be defined within each implementation section. You may, however, define multiple #include statements in a single passthru. For legibility, each #include should begin on a new line, optionally with a blank line to precede and follow the #include list.

Introducing non-IDL Data Types or Classes

You may want a new **.idl** file to reference some element that the SOM Compiler would not recognize, such as a user-defined class or an instance variable or attribute with a user-defined data type. You can reference such elements if they already exist in **.h** or **.xh** files that the SOM Compiler can #include with your new **.idl** file, as follows:

- To introduce a non-IDL class, insert an interface statement that is a forward reference to the existing user-defined class. It must precede the interface statement for the new class in the **.idl** file.
- To declare an instance variable or attribute that is not a valid IDL type, declare a dummy **typedef** preceding the interface declaration.
- In each case above, in the implementation section use a passthru statement to pass an #include statement into the language-specific binding files of the new **.idl** file
  - for the existing user-defined class
  - for the real **typedef**

In the following example, the generic SOM type **somToken** is used in the **.idl** file for the user's types `myRealType` and `myStructType`. The passthru statement then causes an appropriate #include statement to be emitted into the C/C++ binding file, so that the file defining types `myRealType` and `myStructType` will be included when the binding files process. In addition, an interface declaration for `myOtherClass` is defined as a forward reference, so that an instance of that class can be used within the definition of `myCurrentClass`. The passthru statement also #includes the binding file for `myOtherClass`:

```
typedef somToken myRealType;
typedef somToken myStructType;

interface myOtherClass;
interface myCurrentClass : SOMObject {
. . .
  implementation {
     . . .
     myRealType myInstVar;
     attribute myStructType st1;
     passthru C_h =
       ""
       "#include <myTypes.h>"
       "#include <myOtherClass.h>"
       "";
```

```
    };
  };
```

See **Using tk_foreign TypeCode** on page 350.

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 "*/". 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 **Chapter 3, Tutorial for Implementing SOM Classes** on page 49.

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, 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. Use throw-away comments 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** command when running the SOM Compiler (See **Chapter 6, The SOM Compiler** on page 155). 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, place the following declaration in 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.

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, or for use when implementing a class library containing private methods, the **-p** option should be included when running the SOM Compiler. The resulting header files will then include bindings for private, as well as public, methods and attributes. Both the implementation bindings (**.ih** or **.xih** file) and the usage bindings to be #included in the implementation (**.h** or **.xh** file) should be generated under the **-p** option. The **-p** option is described in **Running the SOM Compiler** on page 161.

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 className ;
```

The actual definition of the **interface** for *className* 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 moduleName { definition+ };
```

where each definition is a type declaration, constant declaration, exception declaration, interface statement or nested module statement. Modules are used to scope identifiers.

Alternatively, multiple interfaces can be defined in a single **.idl** file without using a module to group the interfaces. Whether 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.

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:

```
scope-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 *scope-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 unnamed 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 **Running the SOM Compiler** on page 161.)

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**. (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 **Implementation Statements** on page 132.

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** on page 161.)

Chapter 6. The SOM Compiler

The SOM Compiler translates the IDL of a SOM class into a set of binding files for the language that implement the class's methods and the languages that use the class. These bindings make it more convenient to implement and use SOM classes. The SOM Compiler produces binding files for the C and C++ languages. However, 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 that analyzes an OIDL or IDL class definition.
- An emission phase where one or more emitter programs produce binding files.

An emitter program generates each binding file. Setting the SMEMIT environment variable determines the emitters.

Note: In binding files, 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** on page 133).

When changes to definitions in the **.idl** file become necessary, rerun the SOM Compiler to update the current implementation template file. The **c** or **xc** emitter must be specified either with the **-s** option or the **SMEMIT** environment variable. Additional information on generating updates is in **Running Incremental Updates of the Implementation Template File** on page 193.

Binding Files Created By The C Emitters

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 updates 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. The **.c** file contains an **#include** directive for the **.ih** file, described below.

The content 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 information on changing the template file see *Emitter Framework Guide and Reference*.

*filestem***.h**
(produced by the **h** emitter)

This is the header file to be included by C client programs (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 header file to be included in the implementation file (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** on page 189.
- a *className***NewClass** procedure for constructing the class object at run time
- any IDL types and constants defined in the IDL interface

Binding Files Created By The C++ Emitters

The emitters for the C++ language produce the following binding files:

*filestem***.C (for AIX) or** *filestem***.cpp (for OS/2 and Windows NT)**
(produced by the **xc** emitter)

This is a template for a C++ source program that implements a class's methods. This becomes 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.

The C++ implementation file contains an **#include** directive for the **.xih** file, described below.

The content 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 *Emitter Framework Guide and Reference*.

*filestem***.xh**
(produced by the **xh** emitter)

This is the header file to be included by C++ client programs 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 header file to be included in the implementation 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. See **Stub Procedures for Methods** on page 189.
- a *className***NewClass** procedure for constructing the class object at run time
- any IDL types and constants defined in the IDL interface

Other Files the SOM Compiler Generates

*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 are removed. (An item is marked as private by surrounding it with **#ifdef __PRIVATE__** and **#endif** directives.) Thus, the Public Definition Language (**pdl)** emitter can generate a "public" version of a **.idl** file. See **The pdl Facility** on page 167 for information about the **pdl** emitter.

*filestem***.def**
(produced by the **def** emitter) (for OS/2)

This file is used by the linker to package a class as a library. You can combine several classes into a single **.def** file by running the def emitter for each of the **.idl** files that contain classes in the library. Each interface in the **.idl** files should contain the same **dllname** modifier in its implementation section. Or, you can specify the **.def** file's name with the global **dll** modifier on the SOM Compiler command line using the **-m** option. See the **dllname** modifier under **Modifier Statements** on page 133. For additional information of the **dll** modifier and the **-m** option, see **Running the SOM Compiler** on page 161.

When packaging multiple classes in a single library, you must also have a C procedure named **SOMInitModule** to initialize the class library. This procedure should call the routine *className***NewClass** for each class packaged in the library. The **SOMInitModule** procedure can be generated automatically with the **imod** emitter. **SOMInitModule** is called by the SOM Class Manager when the library is dynamically loaded.

*filestem***.exp**
(produced by the **exp** emitter) (for AIX)
This file is used by the linker to package a class as a library. You can combine several classes into a single **.exp** file by running the **exp** emitter for each of the **.idl** files that contain classes in the library. Each interface in the **.idl** files should contain the same **dllname** modifier in its implementation section. Or, you can specify the **.exp** file's name

with the global **dll** modifier on the SOM compiler command line using the **-m** option. See the **dllname** modifier under **Modifier Statements** on page 133. For additional information on the dll modifer and the **-m** option, see **Running the SOM Compiler** on page 161.

When packaging multiple classes in a single library, you must also have a C procedure named **SOMInitModule** to initialize the class library. This procedure must be exported and should call the routine *className***NewClass** for each class packaged in the library. The **SOMInitModule** procedure can be generated automatically with the **imod** emitter. **SOMInitModule** is called by the SOM Class Manager when the library is dynamically loaded.

*filestem***.nid**

(produced by the **def** emitter) (for Windows NT)
This file is used by the linker to package a class as a library. You can combine several classes into a single **.nid** file by running the **def** emitter for each of the **.idl** files that contain classes in the library. Each interface in the **.idl** files should contain the same **dllname** modifier in its implementation section. Or, you can specify the **.nid** file's name with the global **dll** modifier on the SOM compiler command line using the **-m** option. See the **dllname** modifier under **Modifier Statements** on page 133. For additional information on the dll modifer and the **-m** option, see **Running the SOM Compiler** on page 161.

When packaging multiple classes in a single library, you must also have a C procedure named **SOMInitModule** to initialize the class library. This procedure must be exported and should call the routine *className***NewClass** for each class packaged in the library. The **SOMInitModule** procedure can be generated automatically with the **imod** emitter. **SOMInitModule** is called by the SOM Class Manager when the library is dynamically loaded.

*filestemi***.c**

(produced by the **imod** emitter)

This is a C source program that implements a class library's initialization and termination function. This source file should be compiled and linked along with all the class implementation source files. The contents of this file are described in more detail in **Specifying the Initialization and Termination Function** on page 215.

The output source file is named based on the value of the **dllname** modifier in the implementation section of the **.idl** file. Otherwise, the output source file is the value of the **.idl** file's filestem by default. The global modifier **dll** specified with the SOM Compiler's **-m** option can also be used to set the output source file name. See the **dllname** modifier under **Modifier Statements** on page 133. For additional information on the dll modifier and the **-m** option, see **Running the SOM Compiler** on page 161.

When you run the **imod** emitter again for the same set of **.idl** files, any additional classes that have been defined are added to the output source file. Any existing information in the output source file is *not* disturbed.

The content of the C source program is controlled by the Emitter Framework file:

```
<SOMBASE>/include/imod.efw
```

This file can be customized to change the initialization and termination function produced by the emitter. For detailed information on changing the **.efw** file, see *Emitter Framework Guide and Reference*.

The Interface Repository

(produced by the **ir** emitter)

See **Chapter 9, The Interface Repository Framework** on page 337 for a discussion on the Interface Repository and the **ir** emitter.

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 *Programmer's Reference for SOM and DSOM*.

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 SOM Compiler command, as follows:

sc [-*options*] *files*

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** command is issued. The applicable environment variables are as follows:

SMEMIT

Determines which output files the SOM Compiler produces. Its value consists of a list of items separated by semicolons for OS/2 and Windows NT, or by semicolons or colons for AIX. Each item designates an emitter to execute. For example, the statement:

```
SET SMEMIT=c;h;ih             (for OS/2 and Windows NT)
export SMEMIT="c;h;ih"        (for AIX)
```

directs the SOM Compiler to produce the C binding files hello.c, hello.h, and hello.ih from the hello.idl input specification. By comparison,

```
SET SMEMIT=xc;xh;xih          (for OS/2 and Windows NT)
export SMEMIT="xc;xh;xih"     (for AIX)
```

directs the SOM Compiler to produce C++ binding files `hello.C` (for AIX) or `hello.cpp` (for OS/2 and Windows NT), `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.

SMINCLUDE

Specifies where the SOM Compiler should look for **.idl** files **#include**d by the **.idl** file being compiled. Its value should be one or more directory names separated by a semicolon when using OS/2, 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 and Windows NT)

```
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 not coincide with the directory of the input or output files. For AIX, the default setting of SMTMP is /tmp; for OS/2 and Windows NT, the default setting of SMTMP is the root directory of the current drive.

```
SET SMTMP=..\MYSCDIR\GARBAGE
```

tells the SOM Compiler to place the temporary files in the `GARBAGE` directory.

```
SET SMTMP=%TMP%                    (OS/2 and Windows NT)
```

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).

AIX examples:

```
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 and Windows NT)
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. For additional information on the -u option, see **Running the SOM Compiler** on page 161.

SMADDSTAR

When defined, causes all interface references to have a "*" added to them for the C bindings. The command line SOM Compiler options **-maddstar** and **-mnoaddstar** supercede and override the **SMADDSTAR** setting, however.

Environment variables that affect the SOM Compiler can be set for any **-m** options of the SOM Compiler command. The **-E** option can be used to set an environment variable. For additional information on the **-maddstar** and **-mnomaddstar** option and the **-m** and **-E** options, see **Running the SOM Compiler** on page 161.

Running the SOM Compiler

The syntax of the command for running the SOM Compiler takes the forms:

```
sc  [-options]  files
```

The *files* specified in the **sc** 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. For additional information on the **filestem** modifier, see **Modifier Statements** on page 133.

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***]**
: Same 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. When a class contains private methods or attributes, both the implementation bindings and the usage bindings to be **#included** in the implementation should be generated using the **-p** or **-D __PRIVATE__** option.

-E *variable***=***value*
: Sets an environment variable. For additional information on the environment variables: **SMEMIT**, **SMINCLUDE**, **SMTMP**, **SMKNOWNEXTS**, **SOMIR** and **SMADDSTAR**, see **Environment Variables Affecting the SOM Compiler** on page 159.

-I *dir*
: When looking for **#include**d 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. (See also the following Note about the **-m** option, which explains how to convert any "-m name" modifier to an environment variable.)

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 option **-maddstar**, set corresponding environment variables as follows:

```
set SMADDSTAR=1
```

On AIX:

```
export SMNOTC=1
export SMADDSTAR=1
```

Currently Supported **-m** *name*[=*value*] modifier options:

addcmt This option directs the **ir** emitter to emit IDL comments into the Interface Repository file. For example, a method in an IDL file may be immediately followed by a comment that describes what the method does. When the **addcmt** modifier is used, any IDL comments are added as modifiers for the IR objects to which the comments are related. The modifier name is **comment**, and the value of the modifier is the comment string, including line breaks. For example, assume an operation is defined in an IDL file as follows:

```
void testMethod(in string arg);

// This is a comment.
```

Specifying **addcmt** creates an **OperationDef Class** object in the Interface Repository with a **comment** modifier equal to the string `This is a comment.`

addprefixes Adds a functionprefix 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** modifier. 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
```

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.) For additional information onf the **functionprefix** modifier, see **Modifier Statements** on page 133

addstar This option causes all interface references to have a '*' added to them for the C bindings. See **Object Types** on page 124 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. 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.

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**.

dll=*filestem*

> **Warning:** Do not use this modifier on the Windows NT platform. Due to the pecularities of the Windows NT, you must rely on the dllname modifier to be in the idl file itself.

This option specifies the name of the class library DLL to the **exp** emitter, the **def** emitter and the **imod** emitter. In each case, the filestem value defines the filestem name of the output file. In addition, for the **imod** emitter only, the C output file has an "i" appended to the filestem. For example, consider the following SOM Compiler command, which runs both the **def** and **imod** emitters:

```
sc -s"def;imod" -mdll=abc a.idl b.idl c.idl
```

The **def** emitter produces a **.def** file named `abc.def` that contains the exports for the classes defined in the three IDL files `a.idl`, `b.idl` and `c.idl`. The **imod** emitter produces a C file named `abci.c` that contains the class library initialization code for the classes in `a.idl`, `b.idl` and `c.idl`.

Under the **exp, def** or **imod** emitter, the **dll** modifier overrides the value of any **dllname** modifier specified in the implementation section of an IDL file. Thus, for example, if the `a.idl` file contains an interface specifying `dllname="test.dll"`, the **def** emitter would still output the exports of a class defined in `a.idl` into file `abc.def`. (For additional information on the **dllname** modifier, see **Modifier Statements** on page 133.)

The DLL name specified with the **dll** modifier is also used within the source file that is generated by an **imod** emitter. (For full information on the **imod** emitter, see **Specifying the Initialization and Termination Function** on page 215. As described in **Generating Binding Files** on page 155, the linker uses the **def** emitter on OS/2 and the **exp** emitter on AIX to package a class as a library.)

emitappend This option causes emitted files to be appended at the end of existing files of the same name.

imod This option specifies the name of the output C file that will contain the class library initialization code produced by the **imod** emitter. For example, the following SOM Compiler command directs the **imod** emitter to produce a C file with the name `initterm.c`:

```
sc –simod –mimod=initterm a.idl b.idl c.idl
```

The file name given with the **imod** modifier is always used as the output file name, even when the **dll** modifier is used at the same time. For full information on the **imod** emitter, see **Specifying the Initialization and Termination Function** on page 215.

noaccessors This option turns off the automatic creation of OperationDef entries in the Interface Repository for attribute accessors (that is, for an attribute's **_set** and **_get** methods).

noaddstar This option ensures that interface references will not have a "*" added to them for the C bindings. This is the default setting; it is the opposite of the **-m** compiler option **addstar**.

noannot This option specifies to the ir emitter that no file, line and emit modifiers should be added to the output Interface Repository file. This option is used to save some space in the IR file.

By default, the **ir** emitter automatically generates several modifiers: **file**, **line** and **emit** — for the objects contained in an Interface Repository. For example, for any given **InterfaceDef** in an IR, modifiers are automatically associated with it, to define the file containing the interface and the line number on which the interface is defined. The **emit** modifier indicates whether a globally defined **typedef, struct, exception, enum, union,** or constant is to be emitted. Including an **emit** modifier means that one of these data types has been defined within a **#pragma somemittypes** section of the IDL file.

noimod This option specifies to the AIX **exp** emitter that the **imod** emitter was not used to create the class library initialization function and **exp** should therefore not export an initialization function named *dll_stem_name***SOMInitTerm**. By default, the **exp** emitter generates the *dll_stem_name***SOMInitTerm** function name into the exports file. For full information on the **imod** emitter, see **Specifying the Initialization and Termination Function** on page 215.

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** to leave the IR unlocked when updates are made that 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. For additional information, see **Chapter 9, The Interface Repository Framework** on page 337.

nomplans Directs the SOM Compiler not to generate a *marshal plan* for any method of the current class in any emitted **.ih** or **.xih** file.

```
sc –sh;ih –mnomplans xidl
```

Note, `somcsr.h` is still included in `x.ih`, but no marshal plans are emitted.

Specifying **nomplans** has the same effect as specifying `mplan=none` as a method modifier on every method introduced by the class. For more information, see the **mplan** modifier and see **Registering Class Interfaces** on page 30.

nopp This option directs the SOM Compiler not to run the SOM preprocessor on the **.idl** input file.

notc A value of 2 must be specified (that is, `-mnotc=2`) to direct the compiler not to generate typecode information. This option is only used when compiling converted **.csc** files (that is, OIDL files originally) 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.

pass This option directs the **ir** emitter to emit passthru statements into the IR file. Passthrus are added to **InterfaceDef** objects in the IR as modifiers whose value equals the passthru string, including line breaks. The modifiers are:

passthruC.h, passthruC.ih

passthruC.xh, passthruC.xih

passthruC.h_after, passthruC.ih_after

passthruC.xh_after, passthruC.xih_after

pbl This option directs the SOM Compiler that, in declarations containing a linkage specifier, the "*" will appear before the linkage specifier. This is required when using any C++ compiler (Watcom is a known example) that cannot handle declarations in the default format where the "*" follows the linkage specifier. A default example is the declaration:

```
typedef void (SOMLINK * somTD_SOMObject_somFree)
        (SOMObject *somSelf);
```

Under the **-mpbl** option of the SOM Compiler command, the same example would be declared as:

```
typedef void (* SOMLINK somTD_SOMObject_somFree)
        (SOMObject *somSelf);
```

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. See **TypeCode Constants** on page 351 for additional information on **TypeCode** constraints.

updateir[**=***filestem*] This option specifies that the Interface Repository data file will be updated when the **ir** emitter is run. The **updateir** modifier can include a *filestem* value to designate the IR file that will be updated. For example, the following command updates the IR file `test.ir` in the current directory:

```
sc -sir -mupdateir=test.ir a.idl
```

Using the **-sir** option and the **updateir** modifier without a value is equivalent to using the **-u** flag on the SOM Compiler command. Thus, there is no need to use both the **-u** flag and the **-mupdateir** flag together; simply choose one or the other, as convenient.

-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). If **-p** is used, it must be applied for both the implementation bindings (**.ih** or **.xih** file) and the usage bindings (**.h** or **.xh** file) to be #included in the implementation.

-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 SOM Compiler command. This determines which emitters will be run and, hence, which output files will be produced.

The **-s** option is a convenient way to override the **SMEMIT** environment variable. In OS/2 and Windows NT, 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%
```

For additional information on the **SMEMIT** environment variable, see **Environment Variables Affecting the SOM Compiler** on page 159.

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
```

-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. For additional information on **SMEMIT**, see **Environment Variables Affecting the SOM Compiler** on page 159.

-v

Uses verbose mode to display information 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 SOM Compiler command:

`sc -sc hello.idl` Generates file `hello.c`.

`sc -h`

```
     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.
sc –sxc –doutdir hello.idl
     Generates hello.xc in directory outdir.
```

The pdl Facility

The SOM Compiler provides a Public Definition Language **pdl** emitter that generates a file equivalent to the **.idl** file from which it is produced, except that it removes all items within the **.idl** file that are marked as private. To mark an item as private surround it with **#ifdef __PRIVATE__** and **#endif** directives. You can use the **pdl** emitter to generate a public version of a **.idl** file. Generally, client programs need only the public methods and attributes of an interface.

The SOMobjects Toolkit also provides a stand-alone program, **pdl**, capable of generating not just a public version of a **.idl** file, but also arbitrary, user-defined versions. For example, **pdl** can generate release-specific versions of a **.idl** file, a version with legacy support, and so on. Using **pdl** can simplify the management of **.idl** source files by generating multiple versions of the file from a common source, rather than maintaining multiple copies.

Using pdl To Maintain Common Versions of an IDL File

For example, assume that the file **window.idl** contains the interface to a window class, and that maintenance of the current version 1.1 of the class is to be overlapped with development of extensions for the 2.0 version. To support these concurrent activities you could maintain independent copies of **window.idl**, one with version 1.1, the other with 2.0. This approach is error prone and you must make changes to common code in two separate files.

Here's how to use **pdl** to generate two versions from a common source file. If extensions for version 2.0 of the window class are made in **window.idl**:

```
#if VERSION >= 200
     //extensions specific to version 2.0 and higher
#endif
```

then you can generate the version you want by invoking:

```
pdl –DVERSION=num window.idl
```

where *num* is 110 for version 1.1 or 200 for version 2.0. Changes to common code are made in only one file, minimizing the likelihood of error.

The **pdl** program is somewhat similar to **somcpp**, the preprocessor invoked by the SOM compiler. **pdl**, however, preprocesses a **.idl** file only partially, resolving some preprocessor directives, while deferring others to **somcpp** which completely resolves all preprocessor directives.

pdl examines all **#if**, **#ifdef**, and **#ifndef** directives in an input **.idl** file and evaluates the associated conditional expressions to determine if subsequent text up to the matching **#endif** or **#else** should be emitted or skipped. For example:

```
#ifdef ABC
     void op();
```

```
#else
     void op (in long arg);
#endif
```

pdl could produce the following results:

- If `ABC` were defined on the **pdl** command line with `–DABC` then `void op();` would be emitted into the output file.
- If `ABC` were explicitly undefined with `–UABC`, then `void op (in long arg);` would be emitted.
- If `ABC` were unknown by being neither explicitly defined or undefined on the command line, then all five lines would be emitted. **somcpp** and standard C or C++ processors would handle this case differently, emitting `void op(in long arg);` instead of all five lines.

pdl Simplification of Conditional Expressions

If possible, **pdl** simplifies a conditional expression whose value is unknown before it is emitted. Consider a variation of the previous example:

```
#if defined(ABC) && defined(DEF)
     void op();
#else
     void op(in long arg);
#endif
```

If `ABC` were defined with `–DABC` and `DEF` is unknown by being neither explicitly defined or undefined, then **pdl** would emit the following:

```
#if defined(DEF)
     void op();
#else
     void op(in long arg);
#endif
```

because `defined(ABC) && defined(DEF)` can be simplified to `TRUE && defined(DEF)` which can be further simplified to `defined(DEF)`.

pdl resolves as many conditional compilation directives as possible, based on identifier definitions and undefinitions specified on the command line with the **-D** and **-U** options. An identifier can be defined or undefined only on the command line, and not by **#define** or **#undef** directives in the **.idl** file. This gives you fine control over which conditional compilation directives get resolved in a specific invocation of **pdl**, and which ones get deferred to subsequent preprocessing.

Syntax of the pdl Command

The syntax for the **pdl** command is:

pdl [-c *cmd*] [-d *dir*] [-f] [-h] [-s *smemit*] [-D *id* [=*val*]] [-U *id*] [-/ *id*] *files*

The **pdl** command supports the following options. 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.

-c *cmd*
Specifies that the **pdl** program is to run the specified system command for each **.idl** file. 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 issuing the **sc** command with the **-sh** option.

-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.

-h
Displays this description of the **pdl** command syntax and options.

-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 **pdl** is to invoke the SOM Compiler with the **SMEMIT** variable.

-/ *id*
See **-U** option.

files
Specifies one or more **.idl** files to be processed. Filenames must be completely specified with the **.idl** extension.

-D *id* **[=** *val* **]**
Defines id to val, if specified, or 1 otherwise.

-U *id*
Undefined id.

If no **-/**, **-U** or **-D** options are specified, then **-U __PRIVATE__** is assumed.

For example, to install public versions of the **.idl** files in the directory **pubinclude**, type:

```
pdl -d pubinclude *.idl
```

Chapter 7. Implementing Classes in SOM

This chapter is a more in-depth discussion of SOM concepts and the SOM run-time environment than **Chapter 3, Tutorial for Implementing SOM Classes** on page 49. Subsequent sections 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. Refer to **Chapter 5, SOM Interface Definition Language** on page 115 for reference information or the full syntax of topics discussed in this chapter. This chapter also describes customizing SOMobjects.

SOM Run-Time Environment

The SOMobjects Developer Toolkit provides:

- The **SOM Compiler**, used when creating SOM class libraries.
- 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 SOM classes are represented by run-time objects; 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 the following advantages:

- Application programs can access information about a class at run time, including its relationships with other classes, the methods it supports and the size of its instances.
- Much of the information about a class is established at run-time.
- 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 class objects (**SOMObject**, **SOMClass** and **SOMClassMgr**), and one is an instance 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.

In addition to creating the four primitive SOM objects, initialization of the SOM run-time environment also involves initializing global variables to hold data structures 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, the SOM run-time environment is automatically initialized the first time any object is created. 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 behavior common to all SOM objects. All user-defined SOM classes are derived, directly or indirectly, from this class. Every SOM class is a subclass or derived subclass of **SOMObject** and has no instance variables. 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 all run-time objects are instances of some class, it follows that a SOM class object must be an instance of some class. The class of a class is called a metaclass. 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, also called factory methods or constructors, are performed by class objects. 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 supervisory operations. Also, class methods facilitate inheritance of instance methods from parent classes.

See **Figure 4** for the distinction between instance methods and class methods, as well as that between objects, classes, and metaclasses. For the distinction between parent classes and metaclasses, see **Parent Class versus Metaclass** on page 174.

SOMClass is the root class for all SOM metaclasses. All SOM metaclasses must be subclasses or derived metaclasses of **SOMClass** that defines the essential behavior common to all SOM class objects. **SOMClass** provides:

- Class methods for creating new class instances: **somNew**, **somNewNoInit**, **somRenew**, **somRenewNoInit**, **somRenewNoZero** and **somRenewNoInitNoZero**.
- Class methods that dynamically obtain or update information about a class and its methods at run time, including:
  - **somAddDynamicMethod Method** to introduce new dynamic methods
  - **somGetInstanceSize Method** to obtain the size of an instance of this class
  - **somDescendedFrom Method** to test if a specified class is derived from this class

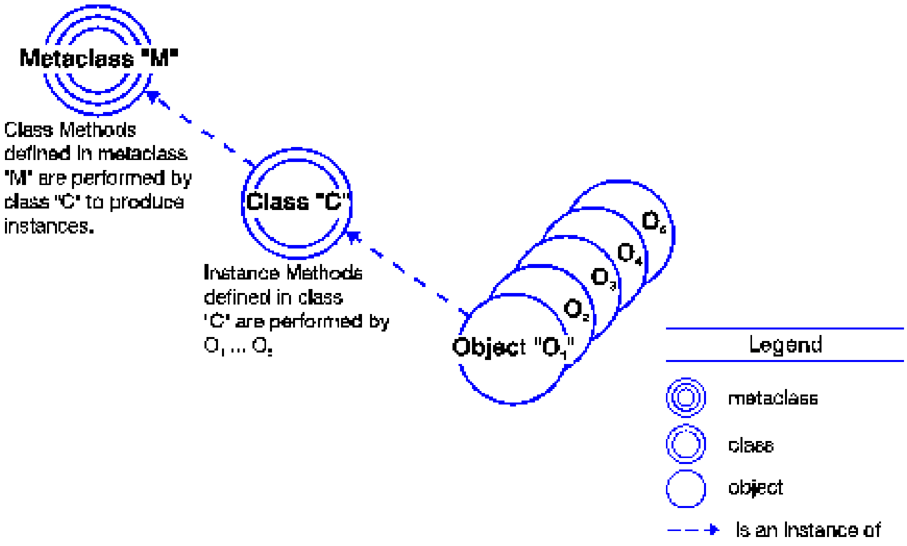

Figure 4. Class methods versus instance methods.

SOMClass is a subclass of **SOMObject**. Hence, SOM class objects can perform the same set of basic instance methods common to all SOM objects. Instance methods allows SOM classes to be real objects in the SOM run-time environment. **SOMClass** has the distinction of being its own metaclass.

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**. An instance of the **SOMClassMgr** class is created during SOM initialization. This instance is the **SOMClassMgrObject** because it is pointed to by that global variable**.** The object **SOMClassMgrObject:**

- Maintains a registry, or run-time directory, of all SOM classes within the current process
- Assists in the dynamic loading and unloading of class libraries

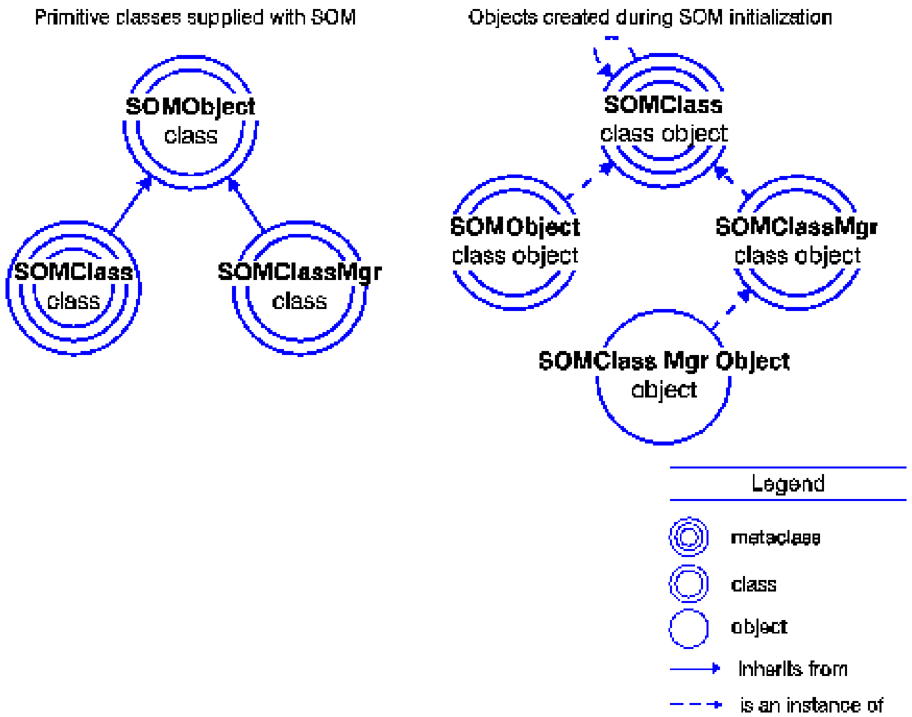

Figure 5. The SOM run-time environment provides four primitive objects, three of them class objects.

SOM classes can be defined locally within a program or can be packaged in a class library. For a class located in a class library, **SOMClassMgr** provides a method, **somFindClass Method**, for directing the **SOMClassMgrObject** to load the library file for the class and to create its class object. However, programs that use the C/C++ language bindings to create and invoke methods are linked so that the operating system will automatically load the appropriate libraries when the program is loaded.

Relationships among the four primitive SOM run-time objects are illustrated in **Figure 5**. The primitive classes supplied with SOM are **SOMObject, SOMClass** and **SOMClassMgr** with the latter class generating an instance called **SOMClassMgrObject.** The left-hand side of **Figure 5** shows parent-class relationships among 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, **SOMClass** is the metaclass of itself, **SOMObject** and **SOMClassMgr**, which are all class objects at run time. **SOMClassMgr** is the class of **SOMClassMgrObject**.

Parent Class versus Metaclass

There is a distinct difference between the notions of parent class and metaclass. Both notions are related to the fact that a class defines the methods and variables of its instances which become instance methods and instance variables.

A parent of a given class is a class from which the given class is derived. Thus, the given class is the child or 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`. 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 are the methods and variables of class objects. For this reason, a metaclass defines class methods: the methods a class object performs. For example, the metaclass `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).

It is important to distinguish the methods of a class object from the methods that the class defines for its instances.

To summarize, the parent of a class provides inherited methods that the class 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 illustrated in **Figure 6**.

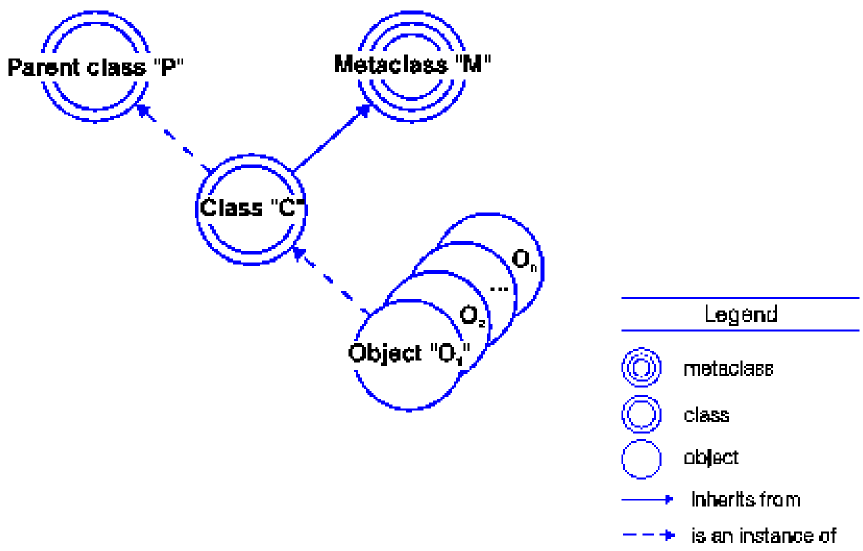

Figure 6. Characteristics of Parent Class versus Metaclass.

To summarize **Figure 6** any class C has both a metaclass and one or more parent classes.

- The parent classes of C provide the inherited instance methods that individual instances (objects `O1`) of class C can perform. Instance methods that an instance `O1` performs might include:
  - initializing itself
  - performing computations using its instance variables
  - printing its instance variables
  - returning its size
- 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
  - inherit its parents's instance methods and instance variables
  - tell its own name
  - create new instances
  - 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 classes 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 7**, a sequence of classes is defined, stemming from **SOMObject**. The or subclass at the end of this line, `C2` inherits instance methods from all of its ancestor classes (here, **SOMObject** and `C1`). An instance created by `C2` can perform any of these instance methods.

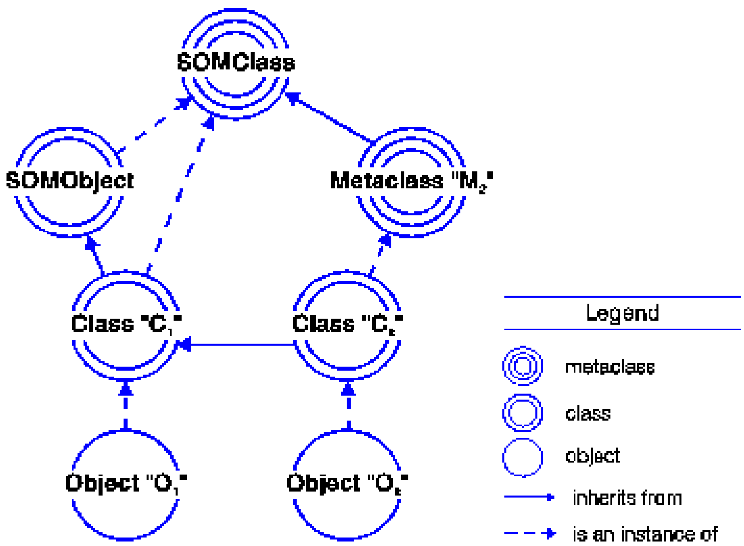

Figure 7. Derivation of Parent Classes and Metaclasses

In an analogous manner, a line of metaclasses is defined, stemming from **SOMClass.** Just as a new class is derived from an existing class, a new metaclass is derived from an existing metaclass. 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**.

Inheritance

One of the defining aspects of an object model is its support for inheritance. This section describes SOM's model for inheritance and explains how this relates to subclassing.

A class in SOM defines an implementation for objects that support a specific interface:

- The interface defines the methods supported by objects of the class, and is specified using SOM IDL.
- The implementation defines the instance variables that implement an object's state and the procedures that implement its methods.

Techniques for Deriving Subclasses

New classes are derived (by subclassing) from previously existing classes through inheritance, specialization (or overriding), and addition, as follows.

Deriving Classes through Inheritance

Subclasses always inherit interface 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). In addition, subclasses generally inherit implementation (that is, method procedures that implement inherited methods, and instance data that supports these procedures).

Deriving Classes through Specialization

Inherited method procedures can be overridden (or redefined). This is often characterized as specializing the implementation of an inherited method so that it is appropriate for objects of the new subclass. With this technique, the class implementor can either completely replace the inherited implementation or (by using parent-method calls) invoke the inherited implementation (method procedures) as part of the overall behavior of the new implementation.

Deriving Classes through Addition

Finally, a subclass can introduce new methods and new instance variables. New instance variables are generally introduced only when necessary to support the implementation of newly introduced methods or of overridden inherited methods. These new additions will, in turn, be inherited by any subclasses of the current class (along with methods and instance data inherited from more distant parents).

Multiple Inheritance

SOM supports multiple inheritance. That is, a class may be derived from (and may inherit interface and implementation from) multiple parent classes. Multiple inheritance is not available in SOM's earlier interface definition language, OIDL. See **Appendix B, Converting OIDL Files to IDL** on page 417 for information on how to automatically convert existing OIDL files to IDL.

Resolving Problems with Multiple Inheritance

It is possible under multiple inheritance to encounter potential conflicts or ambiguities. All multiple inheritance models must face these issues and resolve them in some way. The following topics discuss some of these problems and describe SOM's solutions.

Problem 1: Having alternative meanings for the same name:

One conflict that may arise with multiple inheritance occurs when two ancestors of a class define different methods (in general, with different signatures) using the same name. For example, consider Figure 8, "Multiple Inheritance can Create Naming Conflicts." . 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 8**. (They are implemented using two different method-table entries, each of which is associated with its introducing class.)

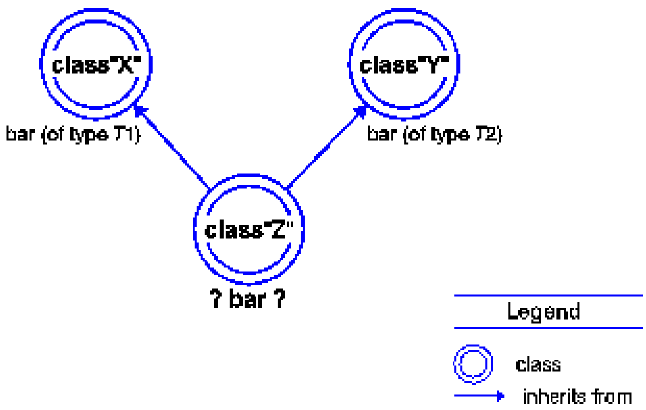

Figure 8. Multiple Inheritance can Create Naming Conflicts.

However, the interface to instances of such classes cannot 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 from name-lookup method resolution (discussed in **Method Resolution** on page 183), since, in this case, a method name alone does not identify a unique method. For these reasons, statically declared multiple-inheritance classes in SOM are restricted to those whose interfaces can be defined using IDL. Thus, the preceding example cannot be constructed with the aid of the SOM Compiler.

This kind of problem can be very irritating when it prevents programmers from using multiple inheritance to combine the functionality from different classes. A good guideline for preventing this problem is that, when introducing new methods in a class, you should try to avoid using method names that other classes might use independently. For example, you can use method names that have identifying prefixes not likely to be used by other classes. All methods introduced by the SOM kernel classes have “som” as a prefix.

Problem 2: Using alternative implementations for the same inherited method:

When multiple inheritance is used to define a class, the class may inherit the same method or instance variable from different parents (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 is basically the location within an object where it is stored. There is no ambiguity here, since classes cannot override the layout of inherited instance data. But classes do override method procedures, so different parents might have different implementations for the same method. The following illustration addresses the question of which method procedure would be inherited when there is an ambiguity with respect to an inherited method implementation.

Consider the situation in **Figure 9**. 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.

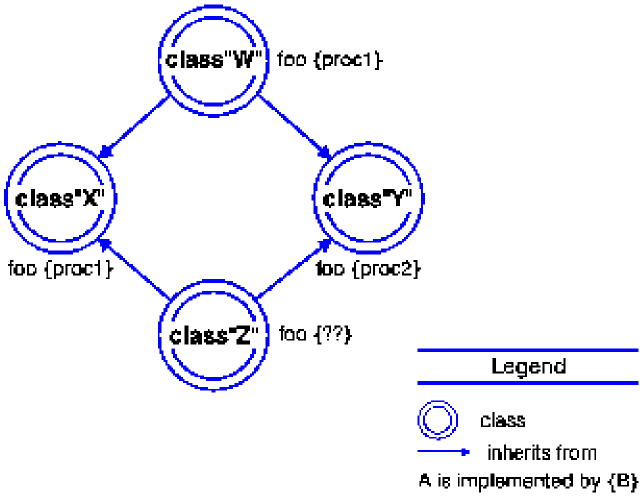

Figure 9. Resolution of Multiple-Inheritance Ambiguities.

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 the **SOMClass** class 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 the one used by the leftmost ancestor from which the method is inherited.

This 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 9**, 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.

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. For additional information on the **select** modifier, see **Modifier Statements** on page 133.

To summarize, defining a multiple-inheritance class requires a class designer to be aware of the potential for alternative inherited implementations of a method. When this happens, the class designer can explicitly choose the desired inherited implementation. The next multiple-inheritance issue deals with problems that may arise when overriding a method whose implementation is inherited from multiple parents.

Problem 3: Making multiple parent-method calls:

In a common OOP paradigm, subclasses override an inherited method with code that:

- provides specialized handling of the method invocation appropriate to the subclass
- performs parent-method calls to allow ancestor classes to participate in the execution of the method. Whether this is appropriate depends on the method involved.

When documenting newly introduced methods, you should always indicate whether the implementation of a method is intended to be shared among different classes. For such shared methods, however, multiple inheritance can pose serious questions.

To illustrate, imagine that class `Z` in the preceding example overrides the `foo` method to provide a specialized implementation. When an overridden method such as `foo` is inherited from multiple parents, the SOMobjects implementation bindings define multiple parent-call macros: one for each (non-abstract) parent from which the method is inherited. Unfortunately, however, calling more than one of these macros normally causes the implementations at and above the "diamond top" (for example, class `W` in the previous example) to be executed multiple times. Depending on the particular method involved, this may or may not create a problem. But it should always be cause for concern.

Given the different ways that multiple inheritance can be used in SOM, there is no good, overall solution to this problem. One way to handle it (assuming you have control over all the classes involved), is to only make parent-method calls to the diamond top from one of its subclasses. This may be possible in special cases, but it is not a general solution. In other situations, it may be appropriate to make only one parent-method call from a multiple-inheritance class, even though the method is inherited from more than one parent.

More fundamentally, however, you can avoid creating diamond tops in the first place. The **SOMObject** class is often a diamond top above multiple inheritance classes, but this is not a problem. Only a few of **SOMObject**'s methods are intended to be overridden with implementations that make parent-method calls. The **somInit** and **somUninit** methods originally fell into this category, but these methods now execute under the overall control of the **somDefaultInit Method** and **somDestruct Method**, which are specially designed to avoid multiple executions at diamond tops. The only other **SOMObject** method that is meant to be overridden with implementations that make parent-method calls to all parents is **somDumpSelfInt Method**. This method causes no problems because **SOMObject** implementation of the method does nothing.

The best approach is to avoid multiple inheritance when it would create more than one inheritance path to any class other than **SOMObject**.

Multiple inheritance requires careful thought. If you create a multiple-inheritance class, and if you override a method that is inherited from multiple parents, you should give careful consideration before making parent-method calls to more than one of these parents, if they have a common ancestor other than **SOMObject**.

Although multiple inheritance can be problematic, it is nevertheless a valuable and important part of SOM. Multiple inheritance is essential in order to provide reliable support for explicit metaclasses.

SOM-Derived Metaclasses

As discussed in **Parent Class versus Metaclass** 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 to introduce new class methods. **Chapter 10, The Metaclass Framework** on page 357 discusses metaclass programming.

The following factors are essential for effective use of metaclasses in SOM:

- Every class in SOM is an object that is implemented by a metaclass.
- You can define and name new metaclasses and can use these metaclasses when defining new SOM classes.

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 also will execute without method-resolution errors on instances of any subclass 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 10**. 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 `O1`), this in turn invokes `bar` on class `A` itself.

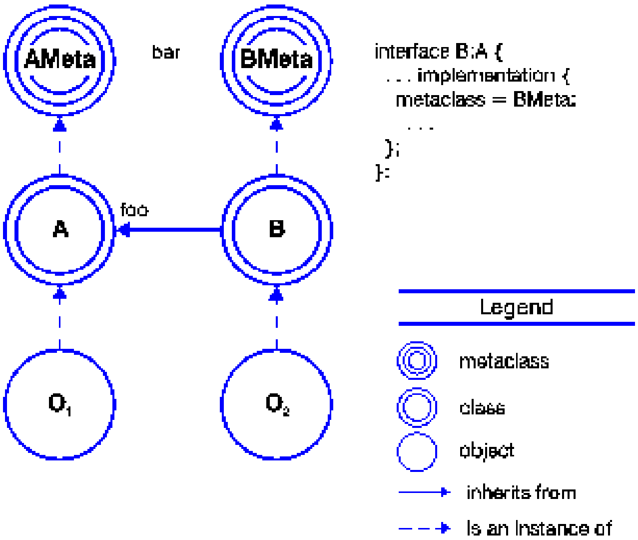

Figure 10. Example of Metaclass Incompatibility

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 10**. If the class hierarchy were formed as in **Figure 10**, then an invocation of `foo` on `O2` 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 guarantee 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 11**, 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`.

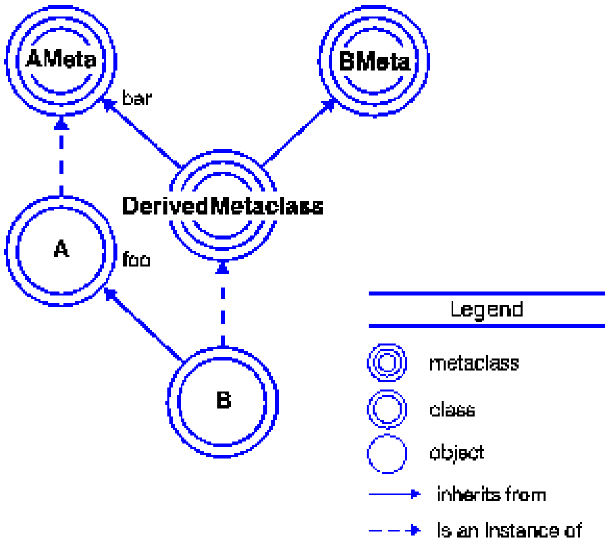

Figure 11. Example of a Derived Metaclass.

There are three important aspects of SOM's approach to derived metaclasses:

1. 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.
2. 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).
3. 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 12**, 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.

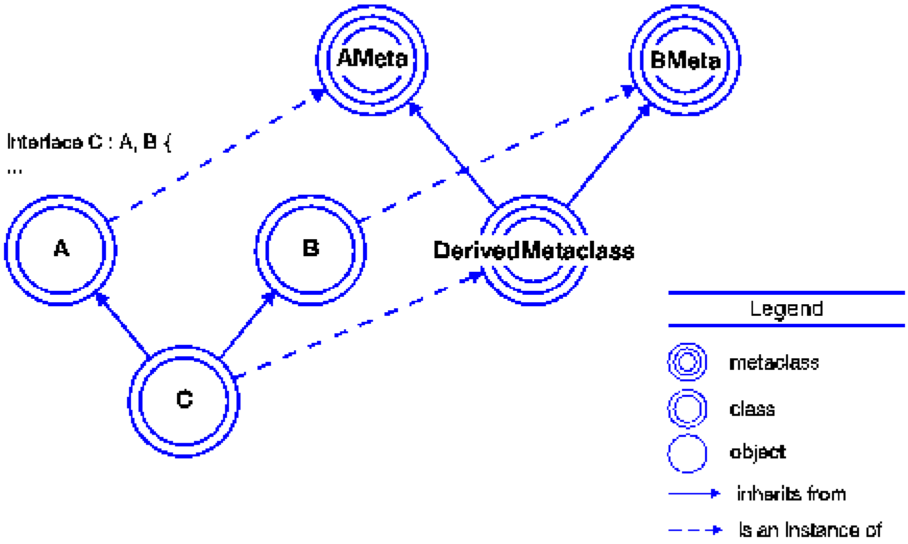

Figure 12. Multiple inheritance in SOM requires derived metaclasses.

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 know the metaclasses of all ancestor classes of a new subclass and 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 implementations are changed. Derived metaclasses allow SOM to make this guarantee. A SOM programmer doesn't have to worry about metaclass incompatibility. Instead, explicit metaclasses can be used to “add in” whatever behavior is desired for a new class. SOM handles anything else that is needed. **Chapter 10, The Metaclass Framework** on page 357 provides useful examples of metaclasses. A SOM programmer can find uses for the techniques illustrated there.

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`.
- A subclass of `Dog`, called `BigDog`, overrides `bark`.
- 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 is an instance of Dog or `BigDog`.

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. These different kinds of method resolution are described after first introducing the four different kinds of methods in SOMobjects.

Four kinds of SOM Methods

SOM supports four different kinds of methods: static, nonstatic, dynamic and direct-call. 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 in **Offset Resolution** on page 185, 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 on which the method is invoked determines the method procedure that is selected.

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. 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 **somAddDynamicMethod Method**.

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 are not overridden in subclasses but are *hidden* by subclasses in which a dynamic method of the same name is added. This provides much the same effect that overriding provides for static methods. Specifically, method resolution for dynamic methods typically begins with the class of the object on which the method is invoked, and works upward in the class hierarchy searching for a class that supports the indicated method. See **Name-Lookup Resolution** on page 186 for a description of the search order.

Direct-Call Procedures

These are similar in concept to C++ static member functions. Direct-call procedures are not given positions in SOM method tables and are not known to SOM class objects. Instead, language bindings are generated to call them *directly* without method resolution.

Strictly speaking, none of the previous method-resolution approaches (offset resolution, name-lookup resolution, or dispatch-function resolution) applies 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.

Offset Resolution

When using SOM's C and C++ language bindings, offset resolution is the default way of resolving static and nonstatic methods, because it is the fastest. For those familiar with C++, this is roughly equivalent to the C++ *virtual function* concept. Offset resolution cannot be used to resolve dynamic methods or direct-call procedures.

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
- The method to be invoked must be part of the introducing class static interface definition

To perform offset method resolution, SOM first obtains a method token. 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 the method's token is stored, offset resolution is quite efficient.

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, and it can be used for all but direct-call procedures. Name-lookup resolution is considerably 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.
- The method is added to the class interface at run time.
- 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. **Figure 13** illustrates this search order.

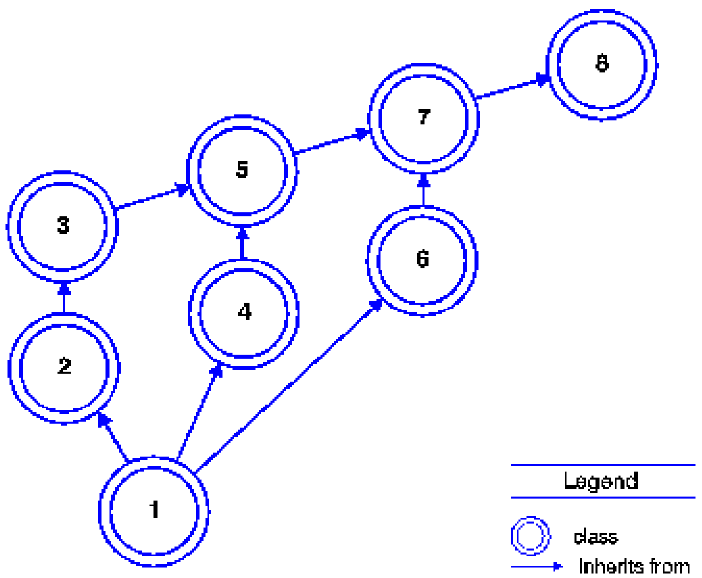

Figure 13. Search Order for Name-Lookup Resolution.

For static and nonstatic methods, 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 4, Using SOM Classes in Client Programs** on page 69 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 based on arbitrary rules associated with the class 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. The use of dispatch-function resolution requires customizing techniques.

Customizing Method Resolution

Customizing method resolution requires the use of metaclasses that override **SOMClass** methods. This is not recommended without use of a Cooperation Framework that guarantees correct operation of SOMobjects in conjunction with such metaclasses. SOMobjects users who require this functionality should request access to the experimental Cooperation Framework used to implement the SOMobjects Metaclass Framework. Metaclasses implemented using the Cooperation Framework may have to be reprogrammed in the future when SOMobjects introduces an officially supported Cooperation Framework.

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.

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 methods, 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** on page 193.

For C programmers, the SOM Compiler generates a **.c** file. For C++ programmers, the SOM Compiler generates a **.C** file (for AIX) or a **.cpp** file. 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 **Chapter 6, The SOM Compiler** on page 155.) Be aware that bindings for both C and C++ cannot be produced by the same compiler execution

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, that the C++ class defined by the C++ usage bindings for a SOM class cannot be subclassed in C++ to create new C++ or SOM classes.

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!".
};
```

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 #included 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");
    }
```

The first line 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 *className*_Class_Source" is replaced by the directive "#define SOM_Module_*moduleName*_Source".

The second line (`#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)
```

Unless it is already defined, the "SOM_Scope" symbol is defined in the implementation header file as *static*. The term "void" signifies the return type of method `sayHello`. The SOMLINK symbol is defined by SOM; it represents the keyword needed to specify a function-linkage convention 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 name 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 formal parameter list for the method procedure. Because each SOM method always receives at least one argument (a pointer to the SOM object that responds to the method), the first parameter name in the prototype of each stub procedure is called **somSelf**. (The macros defined in the implementation header file rely on this convention.) The **somSelf** parameter is a pointer to an object that is an instance of the class 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 **Exceptions and Error Handling** on page 100.

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 by the programmer 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. These macros also use **somThis**.

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 1 in the calling program), the macro produces a message each time the method procedure is entered. (See **Chapter 4, Using SOM Classes in Client Programs** on page 69 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 following the #include directive for the implementation header file, as illustrated below. (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)
```

Alternatively, using -DRETAIL as a C/C++ compiler option when compiling a class implementation achieves the same effect.

The way in which the stub procedure ends is determined by whether the method is a new or an overriding method:

- For non-overriding methods, the stub procedure ends with a return statement. 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 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. See **Making Parent Method Calls** on page 192.

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:

```
SOM_Scope void  SOMLINK HInit(SOMClass *cls)
{
}
```

For a direct-call procedure, the stub appears as follows:

```
SOMEXTERN <rettype> SOMLINK
                 __somp_<className>_<procedureName>(...);
{
}
```

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 4, Using SOM Classes in Client Programs** on page 69 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 accessing instance variables of the object responding to the method and making parent method calls.

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.

Note: 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.

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.)

Making Parent Method Calls

In addition to macros for accessing instance variables, the implementation header file 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 needs to augment the functionality of the existing implementations. 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 as needed. The parent method call can occur anywhere within the overriding method. See **Example 3. Overriding an Inherited Method** on page 60.

The SOM-generated implementation header file defines the following two macros for making parent-method calls from within an overriding method:

*className*_**parent**_*parentClassName_methodName* and

*className*_**parents**_*methodName*

A macro of the first kind is defined for each parent class of the class overriding the method. 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(arg1, 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.
- 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 **If you change the parents of a class** on page 194).
- 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 properly updates method procedure prototypes in the implementation file, class implementors should avoid the following:

- A method procedure name should not be enclosed in parentheses in the prototype.

- A method procedure name must appear in the first line of the prototype, excluding comments and whitespace.

Error messages occur while updating an existing implementation file if it contains non-ANSI C syntax. For example, "old" method definitions below generate errors:

```
Invalid "old" syntax     Required ANSI C
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++. If update errors occur, either the **.idl** file or the implementation file may be at fault. To track down the problem, run the implementation file through the C/C++ compiler. Another option is to move the existing implementation file to another directory, generate a 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 in the implementation file can be another source of errors. The SOM Compiler does not invoke the preprocessor. The programmer should be careful when using conditional compilation, such as in the situation below:

```
Invalid syntax          Required matching braces
#ifdef FOOBAR           #ifdef FOOBAR
 {                        {
  ...                       ...
#else                    }
 {                      #else
  ...                     {
#endif                      ...
 }                        }
                        #endif
```

With two open braces and one closing brace, the emitter reports an unexpected end-of-file.

If you change the parents of a class: The implementation-file emitters never change code in a generated implementation file, changing the parents of a class requires extremely careful attention by the programmer. For example for overridden methods, changing a class parents may invalidate previous parent-method calls provided by the template and require new calls. Neither issue 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 details on ancestor initializer calls, see **Initializing and Uninitializing Objects** on page 195. 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. When the parents of a class are changed, the ancestor-initializer calls 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. Each example provides code where the client program and implementation file is in C and C++.

Note: If you are building an application that uses a combination of C and C++ compiled object modules, you must use the C++ linker.

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

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

OS/2 and WIndows NT in C:
```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include -Fe hello \
      main.c hello.c somtk.lib
```

OS/2 and Windows NT in C++:
```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include -Fe hello \
      main.cpp hello.cpp somtk.lib
```

If the class definition in the **.idl** file changes, re-run the SOM Compiler to generate new header files and update the implementation file to include:

- New comments
- Stub procedures for any new methods
- Revised method procedure prototypes for methods whose signatures 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.

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 initializer 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.

Note: All code written prior to SOMobjects Release 2.x 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.

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 the **directinitclasses** list 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 initializers, with ancestor classes always taken in left-to-right order.

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 SOMobjects Developer Toolkit, however, the **somInitCtrl** structure prevents this from happening.

For example, **Figure 14** 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**. The class numbered **3** is at the top of a diamond.

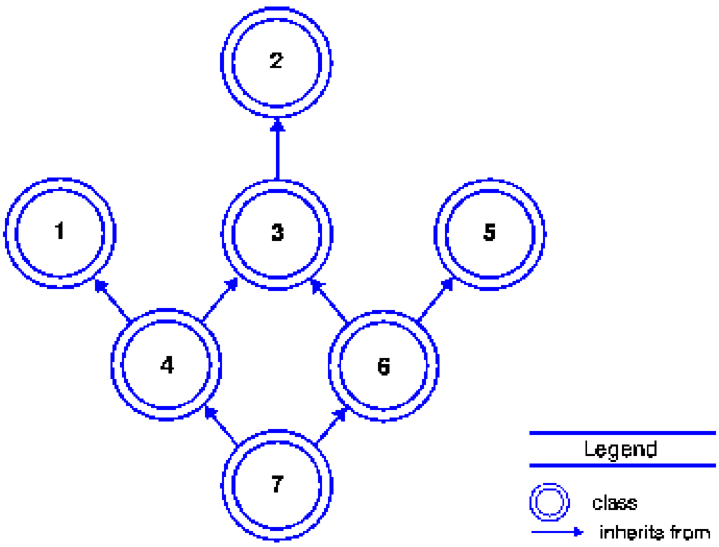

Figure 14. A Default Initializer Ordering of a Class's Inheritance Hierarchy.

In this example, the **somInitCtrl** data structure for class **7** is what tells node **6** in **Figure 14** 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).

Declaring New Initializers in SOM IDL

When defining SOMobjects classes, programmers can declare and implement new initializers. Classes can have as many initializers as desired and subclasses can invoke whichever 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.
- All initializers return **void**.

Accordingly, the **somDefaultInit** initializer introduced by **SOMObject** takes a **somInitCtrl** structure as its only argument. Following is the IDL syntax for this method:

```
void  somDefaultInit (inout somInitCtrl ctrl);
```

When introducing a new initializer, it is necessary to specify the **init** modifier in the implementation section. The modifier tells emitters the method is an initializer, so the method can be supported from the language bindings. This support includes the generation of special initializer stub procedures in the implementation template file and bindings containing ancestor-initialization macros and object constructors that invoke the class implementor's new initializers.

You should begin the names of initializer methods with the name of the class. 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;
};
};
```

Interface `Example1` declares three new initializers. Notice the use of **inout somInitCtrl** as the first argument of each initializer and that the **init** modifier. Both are mandatory to declare initializers. A class can declare any number of initializers. `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:

- **somDefaultInit** is the only initializer that you can override.
- SOMobjects arranges that this initializer will be available on any class.

Historically in SOMobjects Developer 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 the preferred form of initialization because it offers improved efficiency.

Even if no specialized initialization is requisite for a class, you should override the **somDefaultInit** method for efficiency. If you do not override **somDefaultInit**, then a generic and less efficient **somDefaultInit** method procedure will be used for your class.

When you override **somDefaultInit**, the emitter's implementation template file will include a stub procedure similar to those used for other initializers. You can fill it in as appropriate. Default initialization for your class will run much faster than with the generic method procedure. Examples of initializer stub procedures and customizations are 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 somInit Initialization from Earlier SOM Releases

All code before SOMobjects Release 2.1 using documented guidelines for the earlier initialization based on the **somInit** method and all class binaries are fully supported.

Prior to SOMobjects 2.1, initializer methods chained 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 one of the idioms available to OOP users in SOM.

SOM did not constrain initialization any particular way or require the use of any particular ordering of the method procedures of ancestor classes. SOM did provide an overall framework that class designers could easily use 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 chain parent-method calls upward through parent classes. Many class methods that perform object creation automatically called **somInit**.

Note: These will call **somDefaultInit** which call **somInit** for legacy code.

Because it takes no arguments, **somInit** served the purpose of a default initializer. SOM programmers had the option of introducing additional "non-default" initialization methods that took arguments. 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. Because the **init** modifier identifies initializers, usage-binding emitters can now provide these constructors resulting 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. You can use **somInit** on these systems, although this is less efficient than using **somDefaultInit**.

However, you cannot use both methods. In particular, if a class overrides **somDefaultInit** and **somInit**, its **somInit** code will never be executed. You should always override **somDefaultInit** for object initialization. It is likely that when SOMobjects is ported to new systems, **somInit** and **somUninit** may not be supported on those systems. Code written using these obsolete methods will be less portable.

Implementing Initializers

When new initializers are introduced by a class the implementation template file generated by the SOM Toolkit C and C++ emitters 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 the parents of the new class are the ancestors whose initializers are called. When something else is desired, the ID **directinitclasses** modifier can be used to explicitly designate the ancestors whose initializer methods should be invoked by a new class's initializers.

You should not change the number or the ordering of ancestor initializer calls in the first section of an initializer stub procedure. 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 and re-emit the implementation template with a new initializer stub.) The ancestor initializer calls can be modified.

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. 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).

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** on page 201.

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 required by the replacement initializer. Under no circumstances should you change anything else in the first section. If the parents or the **directinitclasses** are changed, then a new implementation stub should be generated.

The example below 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 object initialization. Clients can use the SOM API directly rather than using the usage bindings). 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 *className***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;
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.

```
/*
  * 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);
}
```

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;
}
```

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 metaclass). So, **somDefaultInit** can be overridden by metaclasses to initialize class variables.

Creating SOM Class Libraries

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 implementation of one or more classes and can be dynamically loaded and unloaded as needed by applications. Importantly, class libraries can also be replaced independently of the applications that use them and, provided that the class implementor observes simple SOM guidelines for preserving binary compatibility, can evolve and expand 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 that adds new interfaces, 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 **Modifier Statements** on page 133 for explanations of the **releaseorder, minorversion** and **majorversion** modifiers.)
4. 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, where on Windows they appear as dynamically-linked libraries (or DLLs). The term “DLL” is sometimes used to refer to any form of 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 **Modifier Statements** on page 133.)

It is also important to note that, whereas a client program may have been written to use only the static SOM bindings, it may in fact use SOM frameworks like DSOM which employ dynamic SOM mechanisms for finding classes and invoking methods.

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/termination function to create and destroy 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 and Windows NT to produce the necessary exported symbols for each class. For example, to generate the necessary exports for a class “A”, issue the **sc** command with one of the following **-s** options. (For a discussion of the **sc** command and options, see **Running the SOM Compiler** on page 161.)

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 NT, this command generates an “a.nid” file:

```
sc –sdef a.idl
```

Typically, a class library contains multiple classes. To produce an appropriate export file for all the classes that the library will contain, you should set the **dllname** modifier of each class equal to the name of the loadable library file (**.dll** file), and then run the SOM Compiler with the **-s** option, as shown above, for each IDL file containing a class in the class library. The **exp** and **def** emitters will update the export file with the same filestem name as the given **dllname**. These emitters can sense whether an export file already exists, and, if one does exist, the classes in the given IDL file are added to the existing export file.

To illustrate, assume that the SOM Compiler command shown above invokes the **exp** and **def** emitters on the following IDL file:

```
#include <somobj.idl>
interface A : SOMObject
{
  void methodA();
#ifdef __SOMIDL__
  implementation {
    dllname   = "abc.dll";
  };
```

```
#endif /* __SOMIDL__ */
};
```

The SOM Compiler command creates the following output files:

AIX `abc.exp` file:

```
#! abc.dll
ACClassData
AClassData
ANewClass
abcSOMInitTerm
```

OS/2 `abc.def` file:

```
LIBRARY abc INITINSTANCE
DESCRIPTION 'A Class Library'
PROTMODE
DATA MULTIPLE NONSHARED LOADONCALL
SEGMENTS
    SOMCONST CLASS 'DATA' SHARED READONLY
EXPORTS
  AClassData
  ACClassData
  ANewClass
```

Windows NT `adc.nid` file:

```
LIBRARY abc
EXPORTS
  _AClassData
  _ACClassData
  _ANewClass@8
```

The name of the output file that is created or modified by the **exp** and **def** emitters is determined by the **dllname** modifier specified in the **implementation** section of the IDL file. Therefore, in the example above, the output file name is `abc.exp` (on AIX). If the **dllname** is not specified for a class, the output file created by the emitter is the same as the IDL filestem. So, if the **dllname** had not been specified in the preceding example, the resulting file would have been `a.exp (on AIX).`

To add additional classes to the class library export file, run the SOM Compiler with the **exp** or **def** emitter for the IDL files containing the additional classes. For example, to add the following classes `B` and `C` to the class library, first make sure each additional class specifies the same **dllname** modifier as shown:

```
#include <somobj.idl>
interface B : SOMObject
{
  void methodB();
#ifdef __SOMIDL__
  implementation {
    dllname   = "abc.dll";
  };
#endif /* __SOMIDL__ */
```

```
};
#include <somobj.idl>
interface C : SOMObject
{
  void methodC();
#ifdef __SOMIDL__
  implementation {
    dllname   = "abc.dll";
  };
#endif /* __SOMIDL__ */
};
```

Then, run the SOM Compiler and emitter again, specifying the additional IDL files:

For AIX, this command updates the `abc.exp` file:

```
sc –sexp b.idl c.idl
```

For OS/2, this command updates the `abc.def` file:

```
sc –sdef b.idl c.idl
```

For Windows NT, this command updates the `abc.nid` file:

```
sc –sdef b.idl c.idl
```

The modified output file now appears as follows:

AIX `abc.exp` file:

```
#! abc.dll
ACClassData
AClassData
ANewClass
abcSOMInitTerm
BClassData
BCClassData
BNewClass
CClassData
CCClassData
CNewClass
```

OS/2 `abc.def` file:

```
LIBRARY abc INITINSTANCE
DESCRIPTION 'A Class Library'
PROTMODE
DATA MULTIPLE NONSHARED LOADONCALL
SEGMENTS
  SOMCONST CLASS 'DATA' SHARED READONLY
EXPORTS
  AClassData
  ACClassData
  ANewClass
  BClassData
  BCClassData
  BNewClass
  CClassData
```

```
    CCClassData
    CNewClass
```

Windows NT `abc.nid` file:

```
LIBRARY abc
EXPORTS
  _AClassData
  _ACClassData
  _ANewClass@8
  _BClassData
  _BCClassData
  _BNewClass@8
  _CClassData
  _CCClassData
  _CNewClass@8
```

The recommended way to name the export file is to specify the **dllname** modifier in IDL files. There is, however, an optional mechanism that forces a file name to be used. The **exp** and **def** emitters support a command-line modifier, **dll**. This modifier is specified with the **-m** option of the SOM Compiler command, as shown in the example below:

For AIX:

```
sc -sexp -mdll=abc2 a.idl b.idl c.idl
```

For OS/2 and NT:

```
sc -sdef -mdll=abc2 a.idl b.idl c.idl
```

In the example above, the file `abc2.def` (or `abc2.exp` on AIX) is created and the loadable library named within the file is also `abc2`. (If the **dll** command-line modifier is used when running the **def** emitter, it overrides any settings of the **dllname** modifier within an IDL file. For more information on global modifiers, see the **-m** option in **Running the SOM Compiler** on page 161.) Do not use the **-mdll** modifier on the Windows NT platform.

The data structures and entry points added to the exports file by the **exp** or **def** emitter are required for the correct operation of the SOM run time. On AIX, the emitter also exports the initialization function of the class library. Other symbols in addition to those generated by the emitters can be included if needed, but this is not required by SOM. One convenient feature of SOMobjects is that a class library requires no more than three exports per class. (By contrast, many OOP systems require externals for every method as well.)

Specifying the Initialization and Termination Function

An initialization and termination function for a 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 or unloads a class library, the initialization/termination function will register or unregister all of the classes contained in the library. These classes are managed as a group (called an affinity group).

One class in the affinity group has a privileged position — namely, the class that was specifically 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.

The SOM initialization and termination functions for a class library must be called when the class library is loaded and unloaded. How initialization and termination is handled for SOM libraries is platform specific, because dynamic library loading is unique to an operating system. On OS/2, an init/term function must be called from the class library DLL's general purpose init/term function. When the VisualAge C++ compiler is used, the general DLL init/term function is named **_DLL_InitTerm**. On AIX, the initialization function must be specified as the entry point for the library when the library is linked.

It is the responsibility of the class-library creator to supply the initialization and termination function. There is, however, an **imod** emitter provided with the SOM Compiler to construct a C source file with an appropriate initialization/termination function for your class library. It is recommended that you use the **imod** emitter to create the init/term function for your library rather than constructing one manually.

For AIX: SOMInitModule gets called provided that the **SOMInitModule** function has been specified as the entry point when the shared library is linked.

Running the imod Emitter

This topic describes how to run the **imod** emitter to generate an init/term function for a class library. This topic also provides a detailed explanation of the contents of the source file generated by the **imod** emitter. To illustrate the construction of an init/term function for a class library, consider the following classes `A`, `B`, and `C` in files `a.idl`, `b.idl` and `c.idl`. Notice that each class contains a **dllname** modifier to specify the name of the library file that will contain the class's implementation.

```
a.idl:
#include <somobj.idl>
interface A : SOMObject
{
  void methodA();
#ifdef __SOMIDL__
  implementation {
    dllname   = "abc.dll";
  };
#endif /* __SOMIDL__ */
};
b.idl:
#include <somobj.idl>
interface B : SOMObject
{
  void methodB();
#ifdef __SOMIDL__
  implementation {
    dllname   = "abc.dll";
  };
#endif /* __SOMIDL__ */
};
c.idl:
#include <somobj.idl>
```

```
interface C : SOMObject
{
  void methodC();
#ifdef __SOMIDL__
  implementation {
    dllname   = "abc.dll";
  };
#endif /* __SOMIDL__ */
};
```

A SOM class library init/term function can be constructed by running the SOM Compiler for the IDL files that define the classes in the library. Specify the **imod** emitter with the **-s** flag of the SOM Compiler command. For example, to run the **imod** emitter for the classes above, issue the following command:

```
sc -simod a.idl b.idl c.idl
```

This command creates (or updates) a source file named *dllname_stem***i.c**. This **.c** file is used for a SOM class library programmed either in C or C++. Therefore, in this example the output source file is `abci.c`, because the **dllname** modifier is `abc.dll`. On OS/2 and Windows NT, the *filename* stem of the generated source file is limited to 8 characters. Consequently, if a specified **dllname** *filename* stem is 8 characters or more, it will be truncated to 7 characters to accommodate the "i" that must be appended. Also, if the generated file name would conflict with another source file name, you should use the **imod** global modifier described in the next paragraph.

As an alternative way to run the **imod** emitter, the SOM Compiler can be run using a command-line **-m** global-modifier **imod** option that explicitly names the output source file for the initialization routine. For example, running the following command creates (or updates) a source file named `initterm.c`. This is useful to consistently name a SOM class library init/term source file for every class library you build.

```
sc -simod -mimod=initterm a.idl b.idl c.idl
```

Creating the Class Library

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. Produce an init/term source file for the class library.

   For AIX:

   ```
   sc -simod -mimod=initfunc a.idl b.idl c.idl
   ```

   For OS/2 and Windows NT:

   ```
   sc -simod -mimod=initfunc a.idl b.idl c.idl
   ```

2. Compile all of the implementation files for the classes that will be included in the library. Include the initialization source file generated by the **imod** emitter 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
```

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.c
```

For Windows NT written in **C**:

```
icc -DSOM_DLL_abc -I. -I%SOMBASE%\include -Ge- -Gd+ -Gm+ -c a.c
icc -DSOM_DLL_abc -I. -I%SOMBASE%\include -Ge- -Gd+ -Gm+ -c b.c
icc -DSOM_DLL_abc -I. -I%SOMBASE%\include -Ge- -Gd+ -Gm+ -c c.c
icc -DSOM_DLL_abc -I. -I%SOMBASE%\include -Ge- -Gd+ -Gm+ -c initfunc.c
```

For Windows NT written in **C++**:

```
icc -DSOM_DLL_abc -I. -I%SOMBASE%\include -Ge- -Gd+ -Gm+ -c a.cpp
icc -DSOM_DLL_abc -I. -I%SOMBASE%\include -Ge- -Gd+ -Gm+ -c b.cpp
icc -DSOM_DLL_abc -I. -I%SOMBASE%\include -Ge- -Gd+ -Gm+ -c c.cpp
icc -DSOM_DLL_abc -I. -I%SOMBASE%\include -Ge- -Gd+ -Gm+ -c initfunc.c
```

Note: For OS/2 and NT, the “-Ge” option is used only with the IBM compiler. It indicates that the object files will go into a DLL.

Note: When compiling the file produced by the **imod** emitter on Windows NT, also use the **-IC:\IBMCPPW\SDK\WINH** option, where **C:\IBMCPPW** is the root directory of the VisualAge compiler.

3. Produce an export file for each class.

   For AIX:

   ```
   sc -sexp a.idl b.idl c.idl
   ```

   For OS/2 and Windows NT:

   ```
   sc -sdef a.idl b.idl c.idl
   ```

   Provided the IDL files all include the **dllname=**“`abc.dll`” modifier in their implementation section, the above command will generate `abc.def` on OS/2 and `abc.nid` on Windows NT.

   If the IDL files do not include the **dllname** modifier, you can force all the exported names into the same output file by specifying the **-mdll=***dll_filestem* option along with the **def** emitter on the SOM Compiler command line. Do not use the **-mdll** modifier on the Windows NT platform.

4. 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. The **SOMInitModule** procedure should not be exported. Instead, the function <*class_library_dllname_stem*>**SOMInitTerm** must be exported.

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.

For Windows NT:

```
ilib /geni:abc.lib /DEF:abc.nid
```

The first filename (`abc.lib`) specifies the name for the import library and should always have a suffix of **.lib**. The second filename (`abc.nid`) specifies the exported symbols to include in the import library.

5. 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.

Note: **SOMInitModule** is specified as the initialization function with the **-e** option, even though the **imod** emitter is used to generate a <*dllname_stem*>**SOMInitTerm** function. This is done because the <*dllname_stem*>**SOMInitTerm** function requires an initialization *flag* as input, but the SOM run time — for compatibility with existing class libraries — calls the class library entry point with two version numbers and a *className* string.

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%
    ilink /dll /noe a.obj b.obj c.obj initfunc.obj \
           /OUT:abc.dll somtk.lib abc.def
```

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).

Note: If your class library uses dynamically linked C/C++ runtime libraries, you may receive a linker error message saying that _CRT_term is undefined. Should this occur, you need to add the option -DDYNA_LINK_C to the compilation of the initialization source file, recompile and relink. Do not use the -DDYNA_LINK_C option if your class library uses statically linked C/C++ runtime libraries because this will cause your class library to operate incorrectly.

For Windows NT:

```
set LIB=%SOMBASE%\lib;%LIB%
    ilink /dll /noe a.obj b.obj c.obj initfunc.obj \
           /OUT:abc.dll somtk.lib abc.exp
```

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).

Note: If your class library uses dynamically linked C/C++ runtime libraries, you may receive a linker error message saying that _CRT_term is undefined. Should this occur, you need to add the option -DDYNA_LINK_C to the compilation of the initialization source file, recompile and relink. Do not use the -DDYNA_LINK_C option if your class library uses statically linked C/C++ runtime libraries because this will cause your class library to operate incorrectly.

Building a SOM Library Implemented with C++ on AIX

On AIX, if you use the **somFindClass** or **somFindClsInFile** method or the AIX "load" system call on shared libraries (that is, on SOM DLLs) that have been implemented using C++, this may result in a program crash. (This problem does *not* appear when the shared library is linked to the application at compile time.)

If this problem occurs, you can use the command **makeC++SharedLib**, provided by the AIX XL C++ Compiler/6000, to correctly build a shared library implemented in C++ on AIX. This command must be used instead of the **ld** command when you build a SOM library.

Note: When building a multithreaded shared library, use **makeC++SharedLib_r** instead of **makeC++SharedLib**.

Do the following to build and initialize dynamically loaded SOM libraries that are implemented using XL C++:

Use the **makeC++SharedLib** command (a shell script) to create the DLL, as shown in the sample Makefile commands in the next example. Be aware of the following considerations when creating a Makefile:

- **makeC++SharedLib** expects $BIN and $LIB to be set to the "bin" and "lib" directories for the xlC product. (Or, if BIN and LIB are unset, default paths of `/usr/lpp/xlC/bin` and `/usr/lpp/xlC/lib` are used.) The following Makefile example shows how to explicitly set BIN and LIB before you call **makeC++SharedLib**.
- **make** on AIX requires a <tab> in front of each line of the **makeC++SharedLib** command in the Makefile stanza.
- **makeC++SharedLib** is sensitive to the order of its parameters. The parameter order in the sample below is appropriate for xlC Version 3.1.4.
- See the **makeC++SharedLib** documentation in the AIX XL C++ documentation.

When you use the following sample commands in your Makefile, the values of DLL, EXP, OBJS, LIBDIRLIST and LIBLIST should be tailored for your application:

```
DLL        = foo.dll               # DLL file name
EXP        = foo.exp               # export file name
OBJS       = foo.o fooinit.o       # object (.o) files
LDFLAGS    = -H512 -T512           # ld flags
LIBDIRLIST = -L. -L$(SOMBASE)/lib  # list of library dirs
LIBLIST    = -lsomtk               # list of libraries
<etc.>

$(DLL): $(EXP) $(OBJS)
BIN=/usr/lpp/xlC/bin
LIB=/usr/lpp/xlC/lib
makeC++SharedLib -o $@ \
-n SOMInitModule \
    -bhalt:4 \
$(LIBDIRLIST) $(LIBLIST) \
-p 1024 \
-E $(EXP) -bM:SRE $(OBJS)
```

During the link in step 2, you may receive some warnings from the **nm** command, similar to those below. These warnings can be ignored.

```
nm: libsomtk.a[somc.exp]: 0654-203 Specify an XCOFF object module.
```

Exporting Variables on Windows NT

To export a variable outside of the DLL, instead of declaring it as:

```
SOMEXTERN int SOMLINK var1;
```

You need to declare it as:

```
SOMEXTERN int
#if defined(_WIN32) && !defined(SOM_DLL_myLib)
  SOMDLLIMPORT
     #endif
     var1;
```

Note that **SOM_DLL_myLib** contains the name of the DLL as the suffix. If you have too many variables to export, you can save some typing and make your code look more legible as follows:

```
#if !defined(SOM_IMPORTEXPORT_myLib)
#if defined(_WIN32) && !defined(SOM_DLL_myLib)
#define SOM_IMPORTEXPORT_myLib SOMDLLIMPORT
#else
#define SOM_IMPORTEXPORT_myLib
#endif
#endif
SOMEXTERN int SOM_IMPORTEXPORT_myLib SOMLINK var1;
SOMEXTERN int SOM_IMPORTEXPORT_myLib SOMLINK var2;
....
```

Here, **SOM_IMPORTEXPORT_myLib** is a unique symbol that you introduce. You should make up one that fits the nature of your .h file. Because the emitters also generate their own unique symbols, use a unique symbol of the form “**SOM_IMPORTEXPORT_dllname**”, where *dllname* is the name of your DLL.

The same technique also applies if you want to export a variable within a “Passthru” modifier. For example, if you started with the following OS/2 IDL file:

```
interface foo {
        implementation {
            dllname = "myLib.dll";
            passthru c_h_after = ""
                "SOMEXTERN int SOMLINK var3;";
```

```
                              "SOMEXTERN int SOMLINK var4;";
                         ...
                      }
                  };
```

You would provide the following IDL file for NT:

```
interface foo {
   implementation {
      dllname = "myLib.dll";
      passthru c_h_after = ""
         "#if !defined(SOM_IMPORTEXPORT_myLib)";
         "#if defined(_WIN32) && !defined(SOM_DLL_myLib)";
         "#define SOM_IMPORTEXPORT_myLIb SOMDLLIMPORT";
         "#else";
         "#define SOM_IMPORTEXPORT_myLib";
         "#endif";
         "#endif";
         "SOMEXTERN int SOM_IMPORTEXPORT_myLib SOMLINK var3;";
         "SOMEXTERN int SOM_IMPORTEXPORT_myLib SOMLINK var4;";
      ...
   }
};
```

Other Considerations

Do not set the byte alignment to anything other than 8 bytes.

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. This section and subsequent ones contain advanced topics describing how to customize the various aspects of SOM behavior. For information on DSOM customization, see **Chapter 8, Distributed SOM** on page 229.

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 2**.

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 2. 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. 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: */

#ifdef __OS2__
#pragma linkage(myFree, system)
#endif

void SOMLINK myFree (somToken memPtr)
{
    (Customized code goes here)
}
...
SOMFree = myFree;
```

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.

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 **SOMClassInitFuncName** returns the string **"SOMInitModule"**. The interface to the library initialization function is described under **Creating SOM Class Libraries**.

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:

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 classes 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 relies on the library's automatic initialization routine to perform the appropriate class construction/registration function, as described in **Specifying the Initialization and Termination Function** on page 215.

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** on page 49 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 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;
```

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** on page 99 and **Exceptions and Error Handling** on page 100). 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 Function** (because **SOMOutCharRoutine** would remain in effect for subsequent tasks).

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
- 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);
}
```

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** on page 100 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
- 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.

The actual mutex service function prototypes and global variable declarations are found in file somthrd.h.

Chapter 8. Distributed SOM

This chapter describes the Distributed SOMobjects framework, called DSOM, that enables SOMobjects applications to execute across distributed processes or across a network of machines. The following sections tell how to use DSOM.

DSOM Definition

Whereas the power of SOMobjects technology derives 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 provides support for TCP/IP on AIX and Windows NT. In addition to TCP/IP, DSOM on OS/2 supports NetBIOS through AnyNet.

DSOM can be viewed as:

- An extension to SOM that allows a program to invoke methods on SOM objects in other processes
- An Object Request Broker (ORB), a standardized transport for distributed object interaction. In this respect, DSOM complies with the Common Object Request Broker Architecture (CORBA) 1.1 specification, published by the Object Management Group (OMG) and X/Open™

This chapter describes DSOM from both perspectives.

DSOM Features

The following is a quick summary of some of important features of DSOM:

- 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 LAN transport facilities for workgroup communications.
- Provides support for writing multi-threaded servers and event-driven programs.
- Provides a default object server program, 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.
- Complies with the CORBA 1.1 specification, which is important for portability of applications to other CORBA-compliant ORBs.

Complies with the CORBA Internet Inter-ORB Protocol 1.0 specification, which allows interoperability with other CORBA-compliant ORBs.

DSOM Usage

DSOM is for 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, the clients, access the object via remote method invocations, made transparently by DSOM.

DSOM should be used for applications that require objects to be isolated from the main program. This is done where reliability is a concern; either to protect the object from failures in other parts of the application or to protect the application from an object.

Chapter Outline

This chapter is divided into logical and functional sections.

DSOM Tutorial

DSOM Tutorial on page 233 shows a complete example of how an existing SOM class implementation can be used, without modification, with DSOM to create a distributed application. Using a SOM class implementation as a 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 manages objects
- Implementing classes: writing code that implements objects

Basic Client Programming on page 243, **Basic Server Programming** on page 286 and **Implementing Classes** on page 304 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. Additional examples are provided in these sections to illustrate DSOM services.

Running DSOM Applications

Running DSOM Applications on page 308 explains what is necessary to run a DSOM application, once it has been built and configured.

Advanced Topics

Advanced Topics on page 310 covers:

- **Peer versus Client-Server Processes** on page 310 demonstrates how peer-to-peer object interactions are supported in DSOM.
- **Dynamic Invocation Interface** on page 311 details DSOM support for the CORBA dynamic invocation interface to dynamically build and invoke methods on local or remote objects.
- **Building a Client-Only stub DLL** on page 318 shows how a programmer can build a stub DLL for a remote object so that the DSOM runtime can build a proxy without having access to the remote object's complete DLL.
- **Creating User-Supplied Proxies** on page 319 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** on page 322 discusses how the **SOMDClientProxy** class can be subclassed to define a customized base class that DSOM will use during dynamic proxy-class generation.

Error Reporting and Troubleshooting
: **Error Reporting and Troubleshooting Hints** on page 323 discusses facilities to aid in problem diagnosis.

DSOM and CORBA
: Those readers interested in using DSOM as a CORBA-compliant ORB should read **DSOM as a CORBA-Compliant Object Request Broker** on page 327. This section answers the question: How are CORBA concepts implemented in DSOM?

Deprecated DSOM Methods
: DSOM generally provides backward compatibility for objects and methods supported in DSOM 2.x or before. However, the programming model evolved to incorporate new standards and provide greater flexibility and extensibility. See **Deprecated DSOM Objects and Methods** on page 335 for details on deprecated methods.

DSOM Overview

A DSOM application typically consists of at least four processes running on a single machine or across multiple machines:

- The client program, written by the application developer.
- The server program, which may be the default server program provided by DSOM, or a customized server program written by the application developer. The default server program simply runs in a loop, listening for and servicing client requests. It hosts a well-known server object, which responds to generic methods for loading and instantiating application-specific class libraries, and it hosts the application objects created in it.
- The DSOM location-service daemon, **somdd**, running on the same machine as the servers. The daemon establishes the initial connection between client and server, and starts the server program dynamically on the client's behalf, if necessary.
- The name server providing a Naming Service used by DSOM applications directly and used by DSOM to provide a Factory Service. The DSOM Factory Service is used by client programs to create remote objects.

The DSOM application uses the following files at run-time:

- The SOMobjects configuration file that defines run-time environment settings for DSOM. Each of the above DSOM processes can have unique configuration-file settings, or they can share a common configuration file. SOMobjects provides a default configuration file, which can be customized using any text editor.
- The Interface Repository files that primarily load class libraries dynamically in both client and server processes. These files are created and updated using the SOM Compiler. See **Registering Class Interfaces** on page 30 for more information.
- Implementation Repository files that contain information required only on the server used by the daemon to start servers and by servers to initialize themselves. This repository is created and updated by registering servers using **regimpl**. See **The regimpl Registration Utility** on page 32 for additional information.
- Naming Service files that store information from the Naming Service and the DSOM Factory Service persistently on disk. This includes information about which application

classes are supported on each registered server, collected when the servers are registered. See **Naming Service Concepts** on page 27 for more information.

The typical sequence of events that occurs when configuring and running a DSOM application is as follows:

- Customize environment settings by editing the default configuration file.
- Configure the Naming Service and Security Service using the **som_cfg** utility. This is a one-time step.
- Update the Interface Repository to include application IDL.
- Start **somdd** on the server machines.
- Register application servers and classes, using the **regimpl** tool.
- Run the client application.

At runtime, DSOM clients and servers communicate via proxy objects, a kind of object reference. A proxy object 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 do not execute 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.

For the most part, a client program treats a proxy object exactly as it would treat a local object. The proxy takes responsibility for forwarding requests to and yielding results from the remote object.

Limitations

The following list indicates known limitations of DSOM at the time of this release.

1. 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, allows 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. DSOM has a single server activation policy, that 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_object** 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, **somdd**, 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. The OUT_LIST_MEMORY, IN_COPY_VALUE and DEPENDENT_LIST flags used with the Dynamic Invocation Interface are not supported.

Other important notes concerning DSOM are documented in the product README file.

DSOM Tutorial

The DSOM tutorial presents a sample `stack` application as an introduction to DSOM. This tutorial demonstrates that for simple examples, like `stack`, the class can be used to implement remotely accessed distributed objects. The tutorial presents the DSOM information in these units:

- The application components
- The implementor steps before running the application
- The run-time activity

The source code for this example is provided with the DSOM samples in the SOMobjects Developer Toolkit.

Application Components

The application components used to illustrate DSOM basics consist of the `stack` IDL interface provided by the SOMobjects Developer Toolkit, client coding examples of SOM to DSOM stack changes, stack server implementation and application compiling.

The Stack Interface

The DSOM example assumes that the class implementor built a SOM class library DLL, called `stack.dll`, in the manner described in **Creating SOM Class Libraries** on page 210. 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
};
```

The class implementor could have built this DLL without knowing it would be accessed remotely. Some DLLs require changes in the way their classes pass arguments and manage memory for remote clients. (See **Implementation Constraints** on page 305 for additional information.)

The stack class example assumes that all implementation was performed in a reasonable manner.

Changing a Client Program from a Local to a Remote Stack

The following program uses DSOM to create and access a `stack` object somewhere in the system. The location of the object does not matter to the client program; it just wants a `stack` object. System configuration determines the location of the object.

In local and remote stacks, the stack operations are identical. The main differences lie in program initialization and stack creation. The pertinent portions of the program are the additions and changes required to modify a client program from using a local stack to using a remote stack. Following the program is an explanation of those portions.

```
#include <somd.h>
#include <stack.h>
boolean OperationOK (Environment *ev);
int main(int argc, char *argv[])
{
  Environment ev;
  Stack stk;
  long num = 100;
  SOM_InitEnvironment(&ev);
  SOMD_Init(&ev);
  stk = somdCreate(&ev, "Stack", TRUE);
  /* 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));
      if ( OperationOK(&ev) )
      {
         somPrintf("Stack test successfully completed.\n");
      }
   }
   _somFree(stk);
   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);
```

```
    }
}
```

See **Memory-Management Functions** on page 256 for more information on allocating and freeing memory. **Stack Example Run-Time Scenario** on page 241 describes the run time operations of the previous application.

Code Differences and Similarities:

- Every DSOM program must #include the file **somd.h** for C, or **somd.xh** for C++. This file defines constants, global variables and run-time interfaces used by DSOM. This file is sufficient to establish all necessary DSOM definitions.
- DSOM requires its own initialization call.

  ```
  SOMD_Init(&ev);
  ```

  The call to **SOMD_Init** initializes the DSOM run-time environment, including allocation of global objects. **SOMD_Init** must be called before any DSOM run-time calls are made.
- The local stack creation statement,

  ```
  stk = StackNew();
  ```

  is replaced by the remote stack creation statement,

  ```
  stk = somdCreate(&ev, "Stack", TRUE);
  ```

- The **somdCreate** function creates a remote Stack object in an unspecified server that implements that class. If no object could be created, NULL is returned and an exception is raised. Otherwise, the object returned is a Stack proxy. 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 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);
  ```

  This same call is made in the client program above, but is invoked on a `Stack` proxy. When invoked on a proxy, **somFree** will destroy both the proxy object and the remote target object. 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 **release** method instead of **somFree**.
- The client must shut down DSOM, so that any operating system resources acquired by DSOM for communications or process management can be returned. The `SOMD_Uninit(&ev);` call must be made at the end of every DSOM program.

Locate and Create Method: Creating a remote object is a two-step process. First, the client must locate a suitable factory. Once an appropriate factory has been found, the client must ask the factory to create an instance of the desired class. In the preceding example, the **somdCreate** function performed both steps.

somdCreate Function: The **somdCreate** function places no constraints on how or where the remote `Stack` object should be created. Applications can exercise more control over the criteria by which a factory is chosen by explicitly selecting the factory and then invoking an object-creation method, such as **somNew**.

Naming Service: The Naming Service is a general directory service that allows an object, along with optional properties, to be bound to a name. The Naming Service supports searching for an object based on either the name or specific properties. DSOM provides an extension of the Naming Service, a *factory service*, for selecting factories by specifying the selection criteria as property values. When server implementations are registered with DSOM, information about which classes are associated with each server alias is stored in the Naming Service.

When the **somdCreate** function is used, the only property specified to the factory service is a class name. In general, the client may specify any number of other properties to determine what kind of factory to use. The preceding client program can be modified to create a remote `Stack` object in a specific server whose name, or *alias,* is `StackServer`. The lines below show the changes that were made:

```
#include <somd.h>
#include <stack.h>

int main(int argc, char *argv[]) {
    Stack stk;
    Environment e;
    ExtendedNaming_ExtendedNamingContext enc;
    SOMObject factory;

    SOM_InitEnvironment(&ev);
    SOMD_Init(&ev);

    enc = (ExtendedNaming_ExtendedNamingContext)
           _resolve_initial_references(SOMD_ORBObject, &ev,
           "FactoryService");
    factory = _find_any(enc, &ev,
           "class == 'Stack' and alias == 'StackServer'", 0);
    stk = _somNew(factory);

    _push(stk,&ev,100);
    _push(stk,&ev,200);
    _pop(stk,&ev);
    if (!_empty(stk,&ev)) somPrintf("Top: %d\n", _top(stk,&ev));

    _somFree(stk);
    _release(factory, &ev);
    _release(enc, &ev);
    SOMD_Uninit(&ev);
    SOM_UninitEnvironment(&ev);

    return(0);
```

```
}
```

This version of the program replaces the **somdCreate** operation with calls to the methods **resolve_initial_references, find_any**, **somNew** and **release**. The **resolve_initial_references** method is invoked on a global DSOM object created as a side-effect of calling **SOMD_Init**. **SOMD_ORBObject** contains an instance of class ORB that provides run-time support for both the client and server. The string `"FactoryService"` instructs the method **resolve_initial_references** to return a proxy to the Naming Context where information about object factories is stored. A context is a node in the Naming Service graph, which resides on a DSOM server.

The **find_any** method queries the Naming Service for a factory that meets the input criteria. In the preceding example, **find_any** will return a proxy to a factory that creates `Stack` objects and resides on the DSOM server whose name is `StackServer`.

Once the client has the factory proxy, it can create objects of the desired class. Since there is no standard interface for a factory, this example assumes that the factory for the stack class is the `Stack` class object and simply invokes **somNew** to instantiate the remote object. The **somNew** call creates an object of class `Stack` in the same server as the selected factory.

Calls to the **release** method have been added to destroy the proxies to the factory context and factory object.

Finding Existing Objects: The previous examples show how a remote object can be created by a client for its exclusive use. It is likely that clients will want to find and use objects that already exist. The Naming Service can be used for this purpose. For example, the **find_any** method could be invoked on a well-known Naming Context that contains objects advertised by a collection of servers. The basic mechanisms that DSOM provides for identifying and locating remote objects are discussed in **Basic Client Programming** on page 243.

Stack Server Implementation

The process that manages a remote object is called the object's *server*. A server consists of four parts:

- 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 on behalf of its clients.
- A server object, derived from the **SOMDServer** class, provides methods used to manage objects in the server process.
- One or more class libraries provide implementations for the objects the server manages. Usually these libraries are constructed as dynamically linked libraries (DLLs), so they can be loaded and linked by a server program dynamically.
- The DSOM run time provides the ability for the server to receive incoming messages, demarshal the messages, marshal the responses and send return messages to clients.

This simple example uses the default DSOM server program that is already compiled and linked. The default server behaves as a simple server, in that it continually receives requests and executes them. The default server creates its server object from the default class **SOMDServer**. The default server loads any class libraries it needs upon demand. By using the default server program, the default server object and the existing `Stack` class library, a simple `Stack` server can be provided without any additional programming.

The `Stack` class library, `stack.dll`, can be used without modification in the distributed application; there are no methods that implicitly assume the client and object are in the same address space. See **Implementing Classes** on page 304 for a discussion of how to build class libraries that are readily distributed.

An application may require more functionality in the server program or in the server object than the default implementations provide. A discussion on how to implement customized server programs and server objects is in **Basic Server Programming** on page 286.

Compiling the Application

DSOM programs and class libraries are compiled and linked like any SOM program or library. The header file **somd.h** for C, or **somd.xh** for C++ 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 and Windows NT.

For more information, see **Compiling and Linking** on page 95 and **Compiling and Linking** on page 195.

Preparing to Run an Application

Before running an application, the environment must be prepared, the stack class registered in the Interface Repository, the DSOM daemon started, and the server registered in the Implementation Repository. Once this is done, the application can be run.

Preparing the Environment

DSOM uses several environment settings from the SOMobjects configuration file. Most of these settings appear in the [somd] stanza. When not set explicitly, DSOM assumes default values for all settings except **SOMIR**, which are usually sufficient for workstation applications. The environment variable **SOMENV** should be set to the files containing the current environment settings. For complete information about the configuration file, see **Chapter 2, Configuration and Startup** on page 11.

DSOM provides a tool, **somdchk**, for checking current environment settings. The **somdchk** tool is described in **Step 4. Issue somdchk** on page 23.

Registering the Stack Class in the Interface Repository

Before the application can be run, the contents of the **stack.idl** file must be registered in the SOM Interface Repository on the server machine. The Interface Repository (IR) is a collection of files that collectively make up a database of information about the classes that a DSOM application uses. DSOM servers typically consult the IR to find the name of the DLL for a dynamically loaded class. This is why the DLL name for the `Stack` class must be specified using the `dllname="stack.dll"` modifier in the implementation statement of the `Stack` IDL.

If the client application is not statically linked to the class library, then the IR on the client machine should be updated. The primary use of the IR in DSOM is for loading class libraries dynamically via the **dllname** modifier. For a description of other ways in which DSOM relies on the IR, see **Registering Class Interfaces** on page 30.

Note: DSOM 2.x relied heavily on the IR to obtain method signature information when making remote calls. In the current release, DSOM no longer uses the IR to make remote method calls, if the class implementation (**.ih** or **.xih** file) was compiled with the latest SOM Compiler. If a class implementation was last compiled with a 2.x SOM Compiler and has not been compiled into the IR, DSOM will generate a run-time error when an attempt is made to invoke a method from that class. When this happens, it is usually sufficient to rebuild the class's DLL and its stub, if any, using the latest SOM Compiler. Rebuilding causes the necessary information to be included with the class implementation. The class can be registered in the IR as an alternative, though less efficient, way of accomplishing the same result.

The interface to the `Stack` class is registered in the Interface Repository by running the SOM Compiler and the **ir** emitter against the **stack.idl** file:

```
sc –u –sir stack.idl
```

Before updating the IR, you should set the **SOMIR** environment variable or the corresponding entry in the configuration file. **SOMIR** specifies a list of files that together constitute the IR. 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. For example,

```
On AIX:   export SOMIR=$SOMBASE/etc/som.ir:/u/shared/my.ir
On OS/2 and Windows NT:  set SOMIR=C:\SOM\ETC\SOM.IR;SOM.IR
```

On OS/2, When the **ir** emitter is run, only the last file specified by **SOMIR** is updated. At run time, however, the sequence of files is examined from left to right. See **Using the SOM Compiler to Build an Interface Repository** on page 337 and **Managing Interface Repository Files** on page 338 for more information on the **ir** emitter and **SOMIR**.

Starting the DSOM Daemon

The DSOM daemon, **somdd,** must be started on each server machine, including the machine on which any Naming Service servers reside, if not already running. Client-only machines do not require an active DSOM daemon. The **somdd** daemon is responsible for establishing a *binding* between a client process and a server. It will activate the desired server automatically.

The daemon can be started manually from the command line, or it can 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 and Windows NT. You need not issue any parameters, but you can use the **-q** option to suppress messages.

Registering the Server in the Implementation Repository

It is necessary to register a description of a server implementation in the DSOM Implementation Repository on the server machine, but not on the client machine. The Implementation Repository contains information, such as the name of the program, that the server will execute and how to communicate with the server. DSOM uses this information to activate server processes on demand and to initialize servers.

Registering a server involves specifying which classes the server will support. This information is stored in the Naming Service and is used by DSOM client processes to create remote objects.

The **SOMDDIR** setting in the configuration file gives the name of a directory on the server machine used to store the files that make up the DSOM Implementation Repository and other DSOM-related files. **SOMDDIR** should be set before you register the server.

If **SOMDDIR** is not set, then the default directory is used: On AIX, the default directory is $SOMBASE/etc/dsom. On OS/2 and Windows NT, the default directory is `%SOMBASE%\etc\dsom`. Before you register the server, the Naming Service and Security Service should be configured, and the DSOM daemon must be running. For workgroup configurations, **somdd** must also be running on the machine on which the global root of the Naming Service resides and the machine on which the security server resides. For more information on configuring the Naming Service and Security Service using the **som_cfg** tool, see **Naming Service Concepts** on page 27.

For this example, where the default server program and the default communications protocol are used, it is only necessary to choose a name (alias) for the server and the classes that this server will support. This is accomplished using the **regimpl** utility. The following commands define a default server, named `StackServer`, which supports the `Stack` class:

```
regimpl -A -i StackServer
regimpl -a -i StackServer -c Stack
```

These commands, executed on the server machine, update the Implementation Repository stored in the directory identified by **SOMDDIR**. They also register with the Naming Service the information that client processes need to create remote objects in the server.

Running the Application

Once the DSOM daemon is running on each server machine and the repositories have been updated, the application can be started 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 use it. After the client program ends, the server and daemon continue to run, accepting connections from new clients.

Stack Example Run-Time Scenario

The run-time scenario introduces several of the key architectural components of DSOM. The following scenario steps through the actions taken by the DSOM run time in response to each line of code in the Stack client program presented previously.

- Initialize an environment for error passing:

  ```
  SOM_InitEnvironment(&ev);
  ```

- Initialize DSOM:

  ```
  SOMD_Init(&ev);
  ```

  The **ORB** object, referred to by the global variable **SOMD_ORBObject** is created as a side effect of this call. This global variable provides run-time support for both clients and server.

- Find the Factory Naming Context in the Naming Service and assign its proxy to the variable `enc`:

  ```
  enc = _resolve_initial_references(SOMD_ORBObject, &ev,
        "FactoryService");
  ```

  In response to this call, DSOM uses information in the configuration file SOMNMOBJREF to construct a proxy to the root of the Naming Service; the server in which this object resides need not be running at this point, because name-context objects are persistent. DSOM then gets the name of the factory naming context and resolves this name on the root Naming Context to get a proxy to the Factory Naming Context. This causes the server in which the root Naming Context resides to be started automatically by the DSOM daemon residing on that machine. It also causes the necessary Naming Context objects to be activated. The proxy to the Factory Naming Context is returned to the client program.

  DSOM locates and activates servers automatically using a combination of the information in a proxy, the **somdd** daemon, and the Implementation Repository.

- The proxy tells the client DSOM run time in which server the remote object resides, the location, host and port, of the daemon for that server, and the communications protocols that can be used to contact it.
- DSOM sends a message to that daemon requesting the location of the server.
- The daemon examines its Implementation Repository to find the name of the executable program that implements the requested server and starts that program.
- After initialization, the server notifies the daemon of its port. The daemon then relays this information to the client program.

Thereafter, the client and server communicate directly. Once the server is running, the proxies returned to clients contain all the information needed for clients to contact the server directly instead of using the DSOM daemon.

- Search the Factory Naming Context for an appropriate factory and assign its proxy to the variable `factory`:

```
factory = _find_any(enc, &ev,
            "class == 'Stack' and alias == 'StackServer'", 0);
```

In response to the **find_any** method, the Factory Naming Context searches the name bindings it contains, including those added when the server was registered via **regimpl**, for one whose properties match those specified. For name bindings established by **regimpl**, the name is bound to a NULL object, indicating that the factory object does not yet exist. An additional property associated with the name binding gives the information necessary to construct a proxy to the server object in the server where that factory resides or will reside, when created.

The Factory Naming Context invokes a method on this server object to create a factory for the specified class, in this case, the `Stack` class object. The server loads the `Stack` DLL dynamically, creates the `Stack` class object and returns its proxy to the Factory Naming Context. The Factory Naming Context returns this proxy to the client program.

- Ask the remote factory to create a `Stack` and assign its proxy to the variable `stk`:

```
stk = _somNew(factory);
```

Invoking the **somNew** method on the factory proxy causes a message representing the method call to be marshaled and sent to the server process. In the server process, DSOM demarshals the message and locates the target object on which it invokes the **somNew** method. The result is passed back to the client process in a message. In this case, the result is an object. DSOM automatically creates a new proxy in the client process.

- Invoke methods on the remote `Stack` object, via the proxy:

```
_push(stk,&ev,100);
_push(stk,&ev,200);
_pop(stk,&ev);
if (!_empty(stk,&ev)) t = _top(stk,&ev);
```

- Destroy the proxies and the remote `Stack` object:

```
_somFree(stk);
_release(factory, &ev);
_release(enc, &ev);
```

The factory and the Factory Naming Context objects should not be deleted, since they may be used by other client processes.

- Uninitialize DSOM:

  ```
  SOMD_Uninit(&ev);
  ```

  The **ORB** object stored in **SOMD_ORBObject** will be destroyed as a side effect of this call.

- Free the error-passing environment:

  ```
  SOM_UninitEnvironment(&ev);
  ```

Summary

This example introduced the key concepts of building, installing and running a DSOM application. It also introduced some key components that comprise the DSOM application run-time environment, see **Figure 15**.

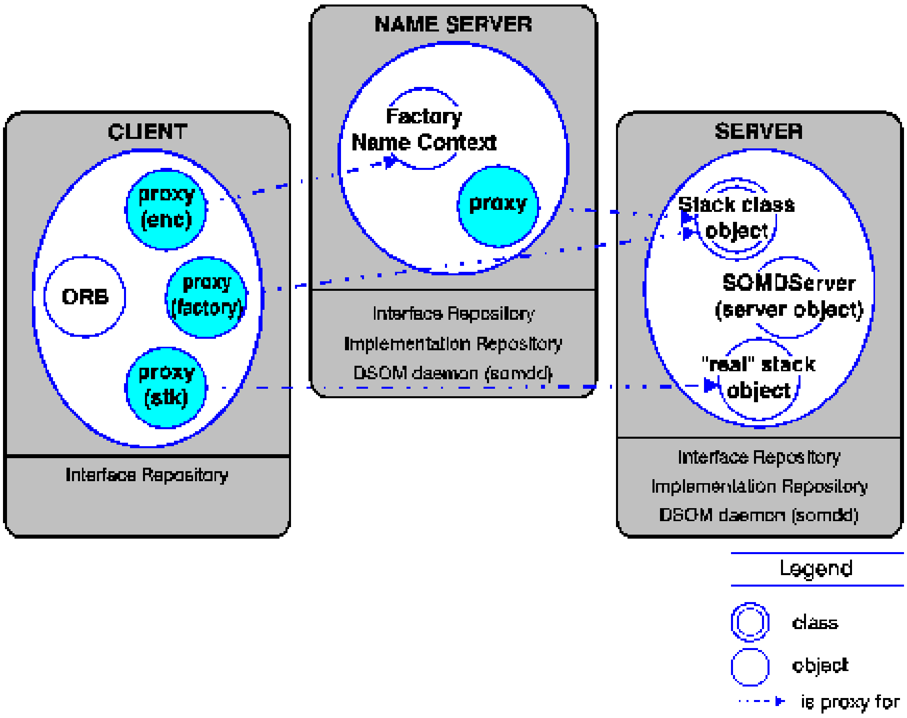

Figure 15. DSOM run-time environment

Basic Client Programming

Client programming in DSOM is generally 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 needs to know how to create, locate, save and destroy remote objects. This is not necessarily done using the usual SOM bindings.

The DSOM run-time environment provides these services to client programs primarily through the DSOM Object Request Broker (ORB) object. These run-time services are described in detail in this section. Examples of how an application developer uses these services are provided throughout the section.

Initializing a Client Program

This topic introduces the ORB object, describes how it is initialized by the **SOMD_Init** procedure and presents two ORB methods that are useful for listing initial object services and for obtaining a reference to a service.

The ORB Object

DSOM provides a CORBA-compliant implementation of the ORB interface. DSOM creates an instance of an ORB object during initialization. The ORB class provides several basic run-time services for both clients and servers; specifically, it defines methods for:

- listing and obtaining initial object references for basic object services
- obtaining object identifiers (string IDs)
- Finding objects, given their identifiers
- creating argument lists to use with the **Dynamic Invocation Interface** on page 311.

The SOMD_Init Function

Calling **SOMD_Init Function** causes various DSOM run-time objects to be created and initialized. Client programs should include the main DSOM header file (**somd.h** for C, **somd.xh** for C++) to define the DSOM run-time interfaces. Since **Environment** parameters are used for passing error results between method and caller, an Environment variable must be declared and initialized for this purpose.

Finding Initial Object References

An object reference is a handle to a remote object in method calls. The ORB class provides methods to list and to obtain initial object references for essential DSOM run-time objects. The IDL prototypes for these two operations are:

```
typedef ObjectId string;
typedef sequence<ObjectId> ObjectIdList;
ObjectIdList list_initial_services ();
SOMObject resolve_initial_references (in ObjectId Identifier);
```

The **list_initial_services Method** can be called to list the run-time objects that are available by calling the **resolve_initial_references Method**. This list is returned as a sequence of well-known strings that each specify a basic object service. Each string can be referred to as an **ObjectId**.

resolve_initial_references takes a single **ObjectId** and returns an appropriate object reference for the requested object service. Since **resolve_initial_references** returns different types of objects, it is prototyped to simply return a SOMObject. The caller of the method **resolve_initial_references** may need to cast the return value to the actual object class.

Three object identifiers are available: InterfaceRepository, NameService and FactoryService. Calling **resolve_initial_references** with **ObjectId** passed as:

- InterfaceRepository returns an object reference of type **Repository**: a local instance of the Interface Repository.
- NameService returns an object reference of type **ExtendedNaming::ExtendedNamingContext**: the root context of the local name tree.

- FactoryService returns an object of type **ExtendedNaming::ExtendedNamingContext**: the naming context where references to SOM object factories are stored.

For example, client code to get a reference to the Interface Repository might look like this:

```
Repository repo;
repo = (Repository) _resolve_initial_references(
        SOMD_ORBObject, &ev, "InterfaceRepository");
```

Creating Remote Objects

The two basic steps in creating a remote object are:

- finding a suitable remote factory object
- invoking an object-creation method on the proxy to the factory object that returns a proxy to the newly created remote object

This section describes these two steps and the **somdCreate Function** that DSOM provides to combine these steps into a single function call.

Advantages of the DSOM Factory Service include:

- The DSOM Factory Service requires less configuration and administration. All configuration is done using the DSOM **regimpl** tools.
- The DSOM Factory Service allows pure-client, non-server, processes to use the same interfaces to create both local, in process, objects and remote objects.
- The DSOM Factory Service generally requires fewer processes to be running and provides better performance.

Finding a SOM Object Factory

The first step in basic object creation is finding a SOM *object factory*, any object that creates a new object. Instances of class SOMClass that support methods **somNew** and **somNewNoInit** are examples of SOM object factories. Applications may provide their own classes, with creation methods, to serve as SOM object factories.

Factory Well-Known: SOM object factories can be advertised in the Naming Service and can be retrieved using standard Naming Service methods. If the name of the factory is well-known, the client can use a method defined by **CosNaming::NamingContext** with the IDL prototype:

SOMObject resolve (in Name *n*);

The **resolve Method** returns the object bound to the input name.

Factory not Well-Known: If the factory is not well-known, the client can query the Naming Service for an appropriate factory by specifying various properties of the desired factory. When server implementations are registered with DSOM using the **regimpl** tool, information about the classes associated with each server is stored in the Naming Service as properties. Server registration creates an entry in the Naming Service with the following three default properties to help identify factories:

class
: specifies the class name of the object to be created, as specified when the class/server association was registered

alias
: specifies the server alias where the factory is located

serverId
: specifies the Implementation Id of the server where the factory is located

Using these three properties, and others added by the system administrator, clients can query the Naming Service. Clients can perform this query using a method defined by **ExtendedNaming::ExtendedNamingContext** with the IDL prototype:

SOMObject find_any (in Constraint *c*, in unsigned long distance);

The **find_any** method returns the first object found whose properties satisfy the input constraint. The type **Constraint** is a string that consists of an expression, including operators, that involves properties and desired values. The **distance** parameter specifies the search depth. When searching for entries created by the DSOM server registration code, a value of `0` is sufficient for the distance parameter. See **BNF for Naming Constraint Language** on page 31 of *Programmer's Guide for Object Services* for a complete description of the naming service constraint language.

For example, client code to find a factory to create an instance of class `Car` looks like this:

```
factory = _find_any(enc, &ev, "class == 'Car'", 0);
```

When a server is registered to create any class, DSOM uses the keyword **_ANY** as the value of **class**. If you want to find a factory to create a specific class or any class, you must explicitly request this. For example:

```
factory = _find_any(enc, &ev, "class=='Car' or class=='_ANY'", 0);
```

Client code to find a factory on server `myCarServer` looks like this:

```
factory = _find_any(enc, &ev,
              "class=='Car' and alias=='myCarServer'", 0);
```

The **find_any** or **resolve** method can return a local or remote factory object. The `Stack` example in **DSOM Tutorial** on page 233 assumed that the factory object and the object to be created were remote. However, the object returned from either method can be a local or a remote factory. The location of the factory object is determined by the information registered in the Naming Service and is transparent to the client. Client programs can explicitly request a local factory object, if necessary, by calling method **find_any** with a constraint where **alias** or **serverId** is set to keyword **_LOCAL**. For example, client code to find a local factory to create an instance of class `Car` looks like this:

```
factory = _find_any(enc, &ev,
              "class == 'Car' and alias == '_LOCAL'", 0);
```

If your client is also a server, besides requesting **alias** as **_LOCAL**, you should request **alias** or **serverId** as the value of the particular server. For example, client code to find a local `Car` factory, executed from a server with **alias** `myCarServer` might look like this:

```
factory = _find_any(enc, &ev,
              "class == 'Car' and
              (alias == '_LOCAL' or alias == 'myCarServer')", 0);
```

The DSOM Factory Service performs special processing for naming bindings having the property `class= _ANY`, indicating that the server can create instances of any class. If a server is registered in the Factory Naming Context with the property `class=_ANY`, then when a **resolve** is performed on that name binding, DSOM will return the server object, an instance of SOMDServer or a subclass thereof, in the registered server. Usually, DSOM returns a factory for the class specified by the class property, but since **_ANY** is not a real class name, DSOM instead returns the server's server object, to act as a default factory.

Similarly, when a **find_any** is performed on the Factory Naming Context and the found name binding has the property `class=_ANY`, DSOM will attempt to find a valid class name in the user-supplied constraint and create a factory for that class in the server. For example, if `class=='Car'` or `class=='_ANY'` and the **find_any** search find a name binding for which `class==_ANY`, then DSOM will create a `Car` factory in the server. If no valid class name is found in the user-supplied constraint, then DSOM will return the server object. DSOM does not analyze the user-supplied constraint when looking for a valid class name; it simply selects the first `class==` clause found in the constraint. In some situations, this may not be appropriate. For example if the constraint is `class=='Car'` or `((class=='Dog' or class='_ANY') and alias=='myServer')` and the found server has properties `alias==myServer` and `class='_ANY'` then DSOM will attempt to create a `Car` factory in `myServer` rather than a `Dog` factory.

Factory Naming Context: When DSOM servers are registered, information about which classes are associated with each server is stored in the Naming Service as properties. This information is recorded in a specialized Naming Context called the *Factory Naming Context*.

For each server/class pair registered, **regimpl** generates a name of the form *<serverUUID><className>* and associates properties **class**, **alias** and **serverId** with that name stored in the Factory Naming Context. Although names and properties in the Naming Service are usually bound to non-NULL object references, the names and properties that **regimpl** store in the Factory Naming Context are associated with NULL object references. NULL object references indicate that the factory object does not exist. In fact, the server where the factory object will reside is probably not running.

The Factory Naming Context provides specialized implementations of some methods. When **resolve** or **find_any** is invoked on the Factory Naming Context, and the specified name or property is associated with a NULL object reference, the Factory Naming Context dynamically starts the server process, creates the factory object in the server and returns a proxy to that factory object. See **Customizing Factory Creation** on page 302 for more information on how factories are created within servers.

The **find_all** method, when invoked on a Factory Naming Context, does not perform a recursive search, regardless of the depth specified in the request. This is done so **find_all** will not result in servers being automatically activated. Other Naming Context methods are also supported but are not customized beyond the default Naming Service functionality.

Creating an Object from a Factory

Once the client finds a SOM object factory, it can ask the factory to create instances of the desired class. In compliance with the OMG COSS Life Cycle Service Specification, there is no standard interface for a SOM object factory. The signature of the creation method depends on the factory instance. The factory instance will be either a SOM class object or an application-specific factory. The SOM IDL **factory** modifier designates if a class provides an application-specific factory class. In either case, it is the application writer's responsibility to know what object-creation method to invoke on it.

For example, when the client receives a factory that is a SOM class object, an instance of SOMClass or one of its subclasses, the client will need to know if **somNew** is appropriate for creating instances of that class, or if it should invoke **somNewNoInit** followed by the appropriate initializer.

Here is an example of a how a `Car` object might be created. This example assumes that the factory is the class object and that a valid instance of the `Car` class can be created using **somNew**.

```
#include <somd.h>
#include <Car.h>
```

```
main()
{
    Environment ev;
    ExtendedNaming_ExtendedNamingContext enc;
    SOMObject factory;
    Car car;
    SOM_InitEnvironment(&ev);
    SOMD_Init(&ev);

   /* get the context where factory objects are stored */
    enc = (ExtendedNaming_ExtendedNamingContext)
           _resolve_initial_references(SOMD_ORBObject, &ev,
                                       "FactoryService");
    /* find a factory that creates "Car" objects */
    factory = _find_any(enc, &ev, "class == 'Car'", 0);

    /* create a "Car" object */
    car = _somNew(factory);
    ...
           _somFree(car);
}
```

Classes that define non-default initializer methods might have corresponding factory classes that have corresponding create methods with parameters to be passed to the non-default initializer.

Here is an example of a how a `Car` object might be created using an application-specific factory:

```
#include <somd.h>
#include <Car.h>

main()
{
    Environment ev;
    ExtendedNaming_ExtendedNamingContext enc;
    SOMObject factory;
    Car car;

    SOM_InitEnvironment(&ev);
    SOMD_Init(&ev);
    /* get the context where factory objects are stored */
    enc = (ExtendedNaming_ExtendedNamingContext)
           _resolve_initial_references(SOMD_ORBObject, &ev,
                                       "FactoryService");
```

```
    /* find a factory that creates "Car" objects */
    factory = _find_any(enc, &ev, "class == 'Car'/ef", 0);

    /* create a "Car" object */
    car = _makeCar(factory, &ev, "Toyota", "red", "2-door");
    ...
    _somFree(car);
}
```

Using the somdCreate Function

The **somdCreate** function is provided to simplify object creation. The function prototype is as follows. The className parameter must match the class name as specified when the class was associated with some server, for example, via **regimpl**.

```
For applications using C "somstars" or C++ bindings:
SOMObject * somdCreate (Environment *ev,
                        Identifier className,
                        boolean init);
For applications using C "somcorba" bindings:
SOMObject somdCreate (Environment *ev,
                      Identifier className,
                      boolean init);
```

The **somdCreate** function calls **find_any** requesting that property **class** be set to the input *className*. If the **init** parameter is TRUE, method **somNew** is called to create the new target object. If the **init** parameter is FALSE, method **somNewNoInit** is called to create the object. This function is useful for simple applications that use only the **somNew** or **somNewNoInit** object-creation methods and use only the **class** property when searching for factories.

Client code to create an instance of class `Car`, using **somNew**, might look like this:

```
#include <somd.h>
#include <Car.h>

main()
{
    Environment ev;
    Car car;
    SOM_InitEnvironment(&ev);
    SOMD_Init(&ev);

    /* create a "Car" object */
    car = somdCreate(&ev, "Car", TRUE);
    ...
}
```

Finding Existing Objects

In addition to creating new objects, it is likely that clients will want to find and use previously created objects. The Naming Service advertises any type of object. For example, DSOM servers use the Naming Service to advertise factories they support. A print service might use the Naming Service to advertise print queues. See **Naming Service Concepts** on page 27 for a description of the Naming Service.

Making Remote Method Calls

As far as the client program is concerned, when a remote object is created, a pointer to the object is returned. What is returned is a pointer to a proxy object, a local representative for the remote target object.

A proxy is responsible for ensuring that operations invoked get forwarded to the target object it represents. The DSOM run-time creates proxy objects automatically, wherever an object is returned from some remote operation. The client program will have at least one proxy for each remote target object on which it operates. Proxies are described further in **DSOM as a CORBA-Compliant Object Request Broker** on page 327 and **Advanced Topics** on page 310.

In the previous example, assuming a remote factory is returned from the Naming Service query, a pointer to a `Car` proxy is returned and put in the variable `car`. Subsequent methods invoked on `car` will be forwarded and executed on the corresponding remote `Car` object.

Remote Object Invocation Methods

DSOM proxies are local representatives of remote objects and can be treated like the target objects. Method invocation on remote objects is in the same manner as if the object is local. This is true for method calls using static bindings or for dynamic dispatching calls. In dynamic dispatching calls, SOM facilities, such as the **somDispatch** method, construct method calls at runtime. CORBA defines a Dynamic Invocation Interface (DII) that is implemented by DSOM as described in **Dynamic Invocation Interface** on page 311.

Note: The **NamedValues** used in DII calls use the **any** fields for argument values. When transmitting the **any** argument to a method call, ensure that the *_value* field is a pointer to a value whose type is described by the *_type* field. To have DSOM transmit an **any** whose *_value* is NULL, set the *_type* field of the **any** to **tk_null**.

The DSOM run-time is responsible for transporting the input method argument values supplied by the caller 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, unless an exception occurs. If the target object raises an exception or a system exception occurs, no return values or **out** parameters will be returned to the caller. DSOM returns to the client only a USER_EXCEPTION declared in an IDL **raises** expression for the method being invoked. If a method attempts to return a USER_EXCEPTION not declared in the method's **raises** expression, DSOM returns a system exception.

DSOM can make remote invocations of methods whose parameter types are any completely defined SOM IDL type. A type is completely defined if it contains no **void*** or **somToken** types, including SOMFOREIGN types. To be transmitted completely, SOMFOREIGN types may need to be declared with an associated user-defined marshaling function, as described in **Passing Foreign Data Types** on page 262.

When a method parameter is an object, a client program making a remote invocation of that method must pass a proxy for that parameter rather than passing a local SOMObject. If the client program is also a DSOM server program, DSOM automatically generates a proxy for the object in the receiver's address space. See **Object Reference Passing in Method Calls** on page 251 for additional information.

Most methods invoked on a default proxy are forwarded and invoked on the remote object. All methods introduced by the target class are forwarded, but some other methods have special behavior. These methods are not forwarded to the remote object because their definition makes better sense in the local context. Examples of non-forwarded methods are:

- debugging methods, such as **somDumpSelf**
- methods that inquire about class properties, such as **somGetSize**
- methods that are specific to proxies, such as **release**

Methods having the SOM IDL **procedure** modifier cannot be invoked remotely using DSOM. These methods are directly called functions and are not sensitive to target object location.

Local Proxy Methods: A complete list of methods executed on the local proxy:

> **create_request**, **create_request_args**, **duplicate, is_proxy**, **release**, **somGetSize**, **somClassDispatch**, **somDumpSelf**, **somDumpSelfInt**, **somIsA**, **somIsInstanceOf**, **somPrintSelf**, **somRespondsTo, somdProxyGetClass**, **somdProxyGetClassName**, **somdReleaseResources**,

Remote Object Methods: SOMDClientProxy introduces methods that forward methods to the remote object:

- **somdTargetFree** invokes **somFree** on the target object.
- **somdTargetGetClass** invokes **somGetClass** on the target object.
- **somdTargetGetClassName** invokes **somGetClassName** on the target object.

Local Proxy and Remote Object Methods: A small number of methods execute on the proxy and on the remote object. Most of these methods deal with proxy destruction:

- **somDefaultInit:** If the proxy has not been initialized, **somDefaultInit** initializes the proxy. If the proxy is initialized, **somDefaultInit** is forwarded to the target object. Proxy objects that DSOM creates will automatically be initialized.
- **somDestruct:** If the proxy is initialized, **somDestruct** is forwarded to the target object. and then the proxy is uninitialized and destroyed. If the proxy is not initialized, **somDestruct** destroys the proxy.
- **somFree:** Invokes **somFree** on the target object then calls **release** to uninitialize and destroy the proxy.

Object Reference Passing in Method Calls

When pointers to objects are returned as method output values, DSOM converts the object pointers in the server to object proxies in the client. Likewise, when a client passes proxy-object pointers as input arguments to a method, DSOM converts the proxy argument in the client to an appropriate object reference in the server. If the proxy being passed from client to server is for an object in that server, DSOM, as part of demarshaling the request in the server, gives the object reference to the server's server object for resolution to a local SOMObject pointer.Otherwise, DSOM leaves the proxy alone, since the proxy must refer to an object in some process other than the target's server.

Local objects, objects that are not proxies, can only be passed as arguments in a remote method call if the process making the call is a server; client-only processes cannot pass local objects as parameters in remote method calls. Servers can make remote calls provided they have executed their **impl_is_ready** initialization call.

In the server, DSOM will not do anything to your object. DSOM creates a SOMDObject to reference your object, marshals that SOMDObject, then releases it. This is equivalent to what happens in the client process: client code is expected to release the proxy when it is finished with it, but this does not necessarily perform a **somFree** on the remote object on the server. There is no need to copy an object before returning it. However, if you return a proxy to an object in another server, you should duplicate it before returning it if your method uses corba memory management ("caller owns"). When DSOM releases the object after marshaling, the copy of the proxy remains.

Memory Allocation and Ownership

When making remote method invocations, you must understand the memory allocation responsibilities of:

- the client program
- the target object in the server
- the DSOM runtime within the client and server process

When using the default memory-ownership policy, the DSOM runtime within the client process "stands in" for the target object: performing all the memory allocation and deallocation that the target object would do during a local call. Likewise, the DSOM runtime within the server process "stands in" for the caller: allocating and deallocating all memory as the caller would do during a local call. If an application adheres to the default memory-management policy, the target object behaves the same way for local and remote clients.

The default memory-ownership policy specifies that the caller is responsible for freeing all parameters and the return result after the call is complete. The default policy states that parameters are uniformly *caller-owned* for local or remote invocation and for caller or target object memory allocation.

The default attribute-accessor, **get** or **set**, method implementations generated by the SOM Compiler do not adhere to the default memory-ownership policy. The default **get** method code does not return storage that the caller can free, and the default **set** method does not make a copy of the input storage. Instead, they do simple assignments and fetches of the object's own instance data. As a result, these implementations, for attributes of non-scalar types, will not work properly when invoked remotely with DSOM, for DSOM assumes all methods are implemented according to the caller-owned policy unless the IDL designates otherwise. Class implementors should therefore either:

- provide their own implementations of **get** and **set** for attributes of non-scalar types using the **noget** and **noset** SOM IDL modifiers in which the **set** method implementation makes a copy of the input storage and the **get** method implementation returns a copy of the object's instance data
- use the **object_owns_result** and **object_owns_parameters** SOM IDL modifiers to designate that the default memory-ownership policy should not be assumed by DSOM when these methods are invoked remotely.

Note: The default memory-ownership policy differs from the default policy used in DSOM 2.x. The new policy takes effect when the class library is recompiled with the current release of SOM Complier. See **The DSOM 2.x Memory-Management Policy** on page 260 for information on how the current default policy differs from the DSOM 2.x policy and how to retain the DSOM 2.x behavior.

The referenced SOM IDL modifiers and other available memory-ownership policies are discussed in **Advanced Memory-Management Options** on page 257.

Default Memory Allocation Responsibilities: Whether the parameter or return result should be allocated by the caller or target object depends on the type of the parameter and its mode (**in**, **inout**, **out** or *return*).

- In local and remote calls, the client program is responsible for providing storage for **in** and **inout** parameters and for initializing pointers used to return **out** parameters.
- The target object is responsible for allocating any other storage necessary to hold the **out** parameters and all storage for return results. For a remote call, DSOM allocates corresponding storage in the client's address space.
- The storage DSOM allocates for **out** parameters and return results from a remote call is the responsibility of the caller and may need to be freed with **ORBfree** or **SOMFree.**

Ownership of memory allocated in the above cases is the responsibility of the client program. For remote method calls, when a remote object allocates memory for a parameter or return value, DSOM allocates memory in the client's address space for the parameter or result. For a parameter or result that is an object, DSOM creates 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. See **Memory-Management Functions** on page 256 for additional information.

On the server side of a remote call, DSOM allocates and initializes the **in** and **inout** parameters and the top-level pointers for **out** parameters before invoking the method on the target object. The target object is responsible for allocating, with **SOMMalloc**, any other storage necessary for **out** parameters and all the storage for the return value.

The target object is responsible for allocating storage as follows:

- strings and *_buffer* field of sequences, when used as **out** arguments or as return results or, in some cases, when used as **inout** arguments
- pointer types, objects, **TypeCode**s, or the *_value* field of **any**s, when used as **inout** or **out** arguments or return results
- arrays when used as return results

If the target object changes any storage within an **inout** parameter, it is the object's responsibility to free, with **SOMFree**, the original orphaned storage. After the method completes, DSOM frees the storage associated with all parameters and return value.

The sequence of memory-management events that occurs during a single remote method invocation, using the default memory-management policy, is:

1. The client application allocates and initializes all **in** and **inout** parameter and return result memory and initializes the pointers used to return **out** parameters. The client then makes the remote method invocation.
2. After sending the method request to the server, the DSOM run-time in the client process releases any **inout** parameter storage that will not be reused.
3. When the method request is received by the server, the DSOM run-time allocates and initializes memory in the server process corresponding to the memory allocated and initialized in the client process by the client.

4. The target object in the server allocates memory as needed for **out** parameters and return results and frees any **inout** parameter and return result memory not reused.
5. As part of returning the response to the client, the DSOM run time in the server process deallocates all parameter and return result memory, including all introduced pointers for **inout** and **out** parameters. See **Introduced Pointers** on page 255 for a discussion of which **inout** and **out** parameters have additional pointers.).
6. When the response is received in the client, the DSOM run-time allocates memory in the client process corresponding to the memory allocated in the server process by the target object. It then returns control to the client application.
7. The client application then has the responsibility to deallocate all parameter and return result memory, including the introduced pointers for **inout** and **out** parameters.

This sequence holds for the default memory-management policy with all data types, but there are slight variations in parameter-memory management for different data types. The main differences in handling different data types are:

- Whether pointers are introduced by the mapping from IDL to C or C++.
- Whether the storage for **inout** parameters is reused or freed and reallocated.
- The function or method that deallocates parameter and return result memory.
- What portions of the storage are to be allocated by the caller and by the target object, for **out** parameters and return results,. The general rule is the caller provides top-level storage, the target object allocates the rest. For example, an **out struct** whose members are strings, the caller allocates the **struct** but the target object allocates the strings. More specifically, the client allocates storage for the following portions of **out** parameters and return results:
  - scalar - all storage
  - object reference and TypeCode - the pointer but not the reference itself; for **out**, the introduced pointer.
  - string and pointer - a pointer; for **out**, the introduced pointer.
  - structs and unions - the top-level storage only, for **out**, the introduced pointer
  - sequence - the sequence storage but not the *_buffer* storage, for **out**, the introduced pointer
  - **any** - the **any** storage but not the *_value* or *_type*, for **out** , the introduced pointer
  - **out** array - the top-level array storage and the introduced pointer
  - return array - the introduced pointer

  It remains the responsibility of the client to free all parameter storage returned.

For details on inout memory, see **Reusing Memory for inout Parameters** on page 255; for details on deallocating parameter and return result memory, see **Memory-Management Functions** on page 256.

Although the client or DSOM run-time in the server process is responsible for allocating the above portions of **out** parameters and return results, the client or DSOM run-time is not responsible for initializing the allocated memory. The target object should assume that all **out** parameter and return-result memory provided by the client or by DSOM are uninitialized when received. The target object is responsible for insuring that all **out** parameter and return-result memory contains valid values before returning.

Introduced Pointers : When passed as parameters, some data types in some directions have a required pointer introduced by the mapping from IDL to C or C++. The following table shows when these pointers are introduced:

	In	**Out**	**Inout**	**Return Value**
scalar types	**no**	**yes**	**yes**	**no**
pointer types	**no**	**yes**	**yes**	**no**
struct types	**yes**	**yes**	**yes**	**no**
arrays	**yes**	**yes**	**yes**	**yes**

Table 3. Introduced pointers by data types

The categories of data types shown in the table are:

- **Scalar types**: **short**, **ushort**, **long**, **ulong**, **enum**, **float**, **double**, **octet**, **boolean**, **char**
- **Pointer types**: **string**, object reference, **TypeCode** and pointer
- **Struct types: struct**, **union**, **sequence**, **any**, **Environment**

Note: An array is the contiguous block of memory holding the data, not including the pointer to the first element.

Reusing Memory for inout Parameters: For **inout** parameters, where the value provided by the client can differ from the value returned by the target object, the question arises as to when storage is reused or freed and reallocated. This section describes DSOM's default conventions for reusing **inout** parameter storage. For local/remote transparent applications, object implementations should adhere to the same conventions.

For **inout** arguments, DSOM reuses the provided storage, when possible, to hold the returned value in the **out** direction. Specifically, DSOM reuses storage for **inout** scalars, pointers, bounded strings, the *_buffer* member of bounded sequences, arrays, and the top-level storage for **inout struct**, **union**, **sequence** and **any**. Embedded storage is treated according to the rules for its own type.

For **inout** unbounded strings, DSOM reuses the provided storage if the returned string is not longer than the original string. For **inout** unbounded **sequence**, DSOM reuses the provided *_buffer* storage if the *_length* of the returned sequence is not greater than the *_maximum* of the original sequence. For either **inout string** or **inout sequence**, if DSOM cannot reuse the provided storage, it deallocates, using **SOMFree**, the original storage and reallocates, using **SOMMalloc**, new storage of the right size. See **suppress_inout_free** IDL modifier for additional information.

DSOM supports the transmission of a NULL pointer or **string**. If an **inout** pointer or **string** is NULL on return, DSOM deallocates the original value before making the value NULL, to avoid memory leakage. Similarly, for **inout sequence**, either bounded or unbounded, if the *_length* of the returned **sequence** is less than the *_length* of the orginal, and the **sequence** element type contains storage, then DSOM deallocates the **sequence** elements between the new *_length* and the old *_length* to avoid memory leakage of these orphaned elements.

For **inout** parameters of all other types (**any**, object reference, **TypeCode**, **SOMFOREIGN**), the original storage is deallocated and new storage allocated to hold the returned value. Any new storage allocated by the DSOM run-time becomes the responsibility of the caller. This **inout** storage is allocated with **SOMMalloc**, so the caller can uniformly use **SOMFree** to deallocate it; **ORBfree** never applies to **inout** storage. Client programs should allocate memory for **inout** parameters, using **SOMMalloc**, rather than using static storage or storage on the stack, for parameter and return result memory DSOM may free.

Memory-Management Functions

DSOM programs manage four kinds of memory resources: blocks of memory, Environment structures, objects and object references. Each resource has different allocation and release functions.

Blocks of Memory: SOM provides the **SOMMalloc** and **SOMFree** functions for allocating and releasing blocks of memory. Memory allocated by DSOM for **inout** parameters and some return values is allocated using **SOMMalloc** and should be freed with **SOMFree**.

For **out** parameters and certain types of return values, CORBA specifies the use of **ORBfree** to free DSOM-allocated storage, for DSOM may use special memory-management techniques to allocate memory. Storage so allocated must be treated specially by the user; specifically, pointers within it may not be modified, nor can they be freed using **SOMFree** and must be freed using **ORBfree**.

On remote method calls, all **out** parameters, except object references and TypeCodes, and the return values for strings, pointers, arrays and sequences are subject to special allocation and must be freed by the client with **ORBfree**. All other result types and all **in** and **inout** parameters except object references and TypeCodes are allocated by DSOM using **SOMMalloc** and should be freed with **SOMFree**.

The major distinction between **SOMFree** and **ORBfree** is that **ORBfree** applies to a whole parameter and recursively frees all embedded memory within a data structure allocated by DSOM. Therefore, the argument to **ORBfree** is the top-level pointer used to return the parameter as required by CORBA 1.1, section 5.16.

For all **out** parameters, the argument to **ORBfree** is the pointer used to return the **out** parameter. Although the caller allocated the top-level storage for the parameter, this is true. Nevertheless, **ORBfree** frees only the portions of the data structure allocated by DSOM. For return value sequences, except **sequence**, the argument to **ORBfree** is the returned pointer or object reference. For return value **sequence**, the argument to **ORBfree** should be *_buffer* of the **sequence**.

If you want to use a single function to free blocks of memory whether allocated by the application or by DSOM, you can call **SOMD_NoORBfree** just after calling **SOMD_Init** in the client program. **SOMD_NoORBfree** requires no arguments and returns no value. **SOMD_NoORBfree** disables the special allocation for **out** parameters as well as the result types listed above and specifies that the client program will free all memory blocks using **SOMFree**, rather than **ORBfree**. In response to this call, DSOM does 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, calling **SOMFree** for each block of memory within. The **SOMD_FreeType** utility function may be useful for this task, if a **TypeCode** for the data structure is available or can be obtained from the Interface Repository.

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. Exceptions returned from remote calls are similar to method results or **out** parameters and have similar memory-management issues. The caller provides the **Environment** structure, and the target object allocates the exception name and exception parameters when an exception is raised.

By default, exceptions resulting from a remote call are subject to special allocation, and must be freed by calling **exception_free** or **somdExceptionFree** on the **Environment** structure where the exception was returned. Both functions are equivalent, but **exception_free** is a CORBA mandate. **somdExceptionFree** performs a deep free of exceptions that resulted from remote calls, but only performs a shallow free on local calls. If the exception parameters contain nested blocks of memory, these must be explicitly freed by the application (if using **somExceptionFree**). DSOM programmers must be aware if the exceptions were received from local or remote calls to know how to free them.

Programmers may want to use a single technique for freeing exceptions regardless of call location. Similar to **ORBfree**, the programmer can disable the special allocation of exceptions from remote calls, and the deep-freeing behavior of **somdExceptionFree**, by using **SOMD_NoORBfree**. While **SOMD_NoORBfree** is in effect, remote exceptions are allocated with **SOMMalloc**, and **somdExceptionFree** behaves the same on all exceptions. By using **SOMD_NoORBfree**, you can use a single mechanism to free all exceptions. For complex data structures, you must walk the structure, explicitly freeing each memory block.

Exceptions raised by the Interface Repository contain memory taken from a single allocated block of memory rather than having exception parameters individually allocated. You should not walk these exceptions nor free them with **somdExceptionFree** when returned from remote method calls. Rather you should free them using **somExceptionFree**.

Objects and Object References: Creating and destroying remote objects was discussed in **Creating Remote Objects** on page 245 and **Destroying Remote Objects** on page 265; local objects in **SOM Classes: The Basics** on page 71. Object references or proxies are typically created by DSOM for the client program. They are released in the client program by using the **release** method or as a side-effect of destroying a remote object using **somFree** or **somDestruct**.

Object references embedded within a larger data structure freed using **ORBfree** should not be released by the application; **ORBfree** will release the embedded object references, rendering them invalid.

TypeCode pseudo-objects should be freed using **TypeCode_free**.

Advanced Memory-Management Options

The default memory-ownership convention provided by DSOM corresponds to the CORBA standard that all parameters and return results are caller owned. This policy is sufficient for most applications. To give programmers more flexibility, DSOM provides three options that allow them to specify when the caller or the object owns parameter storage, and when DSOM will free storage. These options do not affect the allocation of parameter and return result memory, only the freeing of it.

The three types of options are *object-owned*, *dual-owned* and **suppress_inout_free**. Each option is specified using SOM IDL modifiers on a method-by-method basis. The memory-management policy for a particular parameter applies to the parameter and its embedded memory. For example, if a **struct** is owned by the caller, then so are all its members. For more information on the SOM IDL modifiers, see **Implementation Statements** on page 132.

The SOM Compiler does not perform error checking for memory-management modifiers. To insure correct behavior in your DSOM application, you should insure that these modifiers are used correctly. Be sure the modifier names are spelled correctly and annotate the correct method and parameter name.

Object-Owned Policy: Object-owned policy is the opposite of caller-owned policy. The target object takes responsibility for the parameter storage, and the caller cannot free or modify it.

Object-owned **in** parameters are logically transferred from the caller to the target object. During remote invocation, DSOM acts for the target object in the client's address space, and frees, with **SOMFree**, the storage for **in** parameters as part of marshaling. These parameters must be allocated by the client using **SOMMalloc**. Object-owned **inout** parameters are reused, or freed and reallocated, as in the caller-owned case. Object-owned **out** parameters, **inout** parameters or results are not owned by the client. The client must not free their storage after the method completes. This storage remains logically owned by the target object. If the object is remote, this storage is managed by the proxy and is released when the proxy is released. See **The somdReleaseResources method and object-owned parameters** on page 259 for additional information.

On the server side of a remote call, DSOM acts for the caller and allocates and initializes object-owned parameters as in caller-owned parameters. After the method completes, the DSOM run-time within the server process does not free the storage for any object-owned parameters. Just as for local calls, the target object's implementation determines when the memory associated with object-owned parameters and results are freed.

The sequence of events during a remote invocation of an object-owned parameter is:

1. The client application allocates and initializes all **in** and **inout** parameter and return result memory and initializes the pointers used to return **out** parameters by providing the top-level storage. The client then makes the remote method invocation.
2. After sending the method request to the server, the DSOM run time in the client process releases the storage associated with object-owned **in** parameters and for **inout** parameter storage that will not be reused.
3. When the method request is received by the server, the DSOM runtime allocates and initializes memory in the server process corresponding to the memory allocated and initialized in the client process by the client.
4. The target object in the server allocates memory for **out** parameters and return results. Unlike memory for caller-owned parameters, the memory for object-owned parameters can be static memory rather than memory allocated from the heap, for the target object assumes responsibility for deallocating this storage and for releasing the storage associated with object-owned **in** and **inout** parameters.
5. When the client receives the response, the DSOM runtime allocates memory in the client process corresponding to the memory allocated in the server process by the target object. It returns this storage to the client application. The responsibility for deallocating it rests with the proxy object on which the remote method was invoked. The memory is released when the proxy is destroyed. The proxy assumes ownership of the storage associated with object-owned **inout** parameters.

For parameters whose IDL-to-C or -C++ mapping introduces a pointer, object-ownership sometimes applies to the introduced pointer and the data item. For scalar types and types whose representation is a pointer, the introduced pointer for an **out** or **inout** is not subject to object-ownership. This pointer should be freed by the client, and it will be freed by the DSOM run-time in the server process after the method completes. For all other types, the introduced pointer for an **in**, **out** or **inout** is subject to storage ownership, and remains the responsibility of the target object and of the proxy object, in the client's address space.

If the client calls a method having an object-owned **out struct** and an object-owned **out string** as parameters, the client must pass a pointer to heap-allocated storage for the **struct**. It may pass a pointer to static storage for the pointer implicit in the **string** type, but not for the characters within the string. The **struct** pointer becomes object-owned, but the **string** pointer does not.

To designate that the result of a particular method is object-owned, use the SOM IDL modifier **object_owns_result**. To designate particular parameters of a method as object-owned, use the **object_owns_parameters** modifier. To designate the result and parameters `abc` and `def` of method `newmethod` as object-owned in the implementation section of SOM IDL, use the following statements:

```
newmethod : object_owns_result;

newmethod : object_owns_parameters = "abc, def";
```

To designate an attribute's **set** and **get** methods as object-owned, annotate the methods as:

```
_get_myattrib : object_owns_result;parameter memory

_set_myattrib : object_owns_parameters = "myattrib";
```

The somdReleaseResources method and object-owned parameters: when a DSOM client program makes a remote method invocation, via a proxy, on a method having an object-owned parameter or return result, the client-side memory associated with the parameter or result is owned by the caller's proxy; the server-side, the remote object. The memory owned by the caller's proxy is freed when the client program releases the proxy.

A DSOM client can instruct a proxy object to free all its memory for the client without releasing the proxy, by invoking the **somdReleaseResources** method on the proxy object. Calling **somdReleaseResources** prevents unused memory from accumulating in a proxy.

Consider a client program repeatedly invoking a remote method that returns a string designated as object-owned. The proxy stores the memory associated with all returned strings, even non-unique strings , until the proxy is released. If the client program only uses the last result returned, then unused memory accumulates in the proxy. The client program can prevent memory accumulation by invoking **somdReleaseResources** on the proxy object periodically.

Dual-owned policy: The dual-owned policy is a combination of the default, caller-owned, policy and object-owned policy, but is only meaningful for remote calls. In the dual-owned policy, the caller and the object are each responsible for the release of its own copy of the parameter. In a remote call, at least two copies of a parameter always exist, since there must be a copy in the local and remote address spaces. It is reasonable to think of each side owning its copy. In a local call there is usually only one copy of the parameter, so there is no need for dual ownership.

For a dual-owned parameter, DSOM frees certain storage for **inout** parameters but does not free any other storage in the caller. The caller is responsible for all **out**, **inout** and result storage. To the caller, dual-owned looks no different than the default behavior. On the server side, dual-owned looks like object-owned; DSOM allocates and initializes the parameters, but frees nothing after method completion, except certain introduced pointers.

The sequence of events during a remote invocation of a dual-owned parameter is:

1. The client application allocates and initializes all **in** and **inout** parameter and return result memory and initializes the pointers used to return **out** parameters. The client then makes the remote method invocation.
2. After sending the method request to the server, the DSOM run-time in the client process releases any **inout** parameter storage that will not be reused.
3. When the server receives the method request, the DSOM run-time allocates and initializes memory in the server process corresponding to the memory allocated and initialized in the client process by the client.

4. The target object in the server allocates memory for **out** parameters and return results and frees any **inout** parameter and return result memory not reused.
5. As part of sending the response back to the client, the DSOM run-time in the server process deallocates only those introduced pointers that it allocated that are not subject to object ownership. All other parameter or result storage is not deallocated. The target object retains ownership of this storage.
6. When the client receives the response, the DSOM run-time allocates memory in the client process corresponding to the memory allocated in the server process by the target object. It returns this storage to the client application.
7. The client application then has the responsibility to deallocate all parameter and return result memory, including the introduced pointers for **inout** and **out** parameters.

To designate that the result of a particular method is dual-owned, use the SOM IDL modifier **dual_owned_result**. To designate particular parameters of a method as dual-owned, use the **dual_owned_parameters** modifier. To designate the result and parameters `abc` and `def` of method `newmethod` as dual-owned in the implementation section of SOM IDL, use the following statements:

```
newmethod : dual_owned_result;
newmethod : dual_owned_parameters = "abc, def";
```

To designate an attribute's **set** and **get** methods as dual-owned, annotate the methods as

```
_get_myattrib : dual_owned_result;
_set_myattrib : dual_owned_parameters = "myattrib";
```

suppress_inout_free: The **suppress_inout_free** SOM IDL modifier suppresses the freeing by DSOM of any part of an **inout** parameter in the caller's address space. The caller assumes responsibility for the storage that DSOM would have freed by default. This modifier is useful if the original storage for the **inout** was static storage and should not be freed by DSOM. When using **suppress_inout_free**, avoid orphaning the original storage and creating a memory leak.

To designate that DSOM should not free any part of a particular **inout** parameter of a method, use the **suppress_inout_free** SOM IDL modifier on the method, giving the parameter name as the modifier value. For parameter `abc` of method `newmethod` in the implementation section of SOM IDL, use the following statement:

```
newmethod : suppress_inout_free = abc;
```

If multiple parameters require annotation, all the parameter names should be listed:

```
newmethod : suppress_inout_free = "abc, def, xyz";
```

The DSOM 2.x Memory-Management Policy: DSOM's current default memory-management policy is caller-owned. In contrast, the default policy in DSOM 2.x was a combination of caller-owned for **in** and **inout** parameters, dual-owned for **out** parameters and return results, and **suppress_inout_free** for **inout** parameters.

DSOM 2.x applications that depend on the previous default should have the appropriate modifiers added to their IDL when the application is recompiled for use with the current release of DSOM. For methods having an **inout** parameter of type **any**, **TypeCode** or an object type, the implementation section of the IDL should be annotated with the modifier:

```
<method-name> : suppress_inout_free = <parameter-name>;
```

For methods having a return value that is a non-scalar type, such as **string**, and the application depends on the DSOM 2.x behavior, the implementation section of the IDL should be annotated with the modifier:

```
<method-name> : dual_owned_result;
```

An attribute's **get** method annotation is:

```
_get_<attribute-name> : dual_owned_result;
```

For any methods returning an **out** parameter of a non-scalar type, to preserve the DSOM 2.x runtime behavior, use the following IDL modifier:

```
<method-name> : dual_owned_parameters = <parameter-name>;
```

Note: The new default memory-management policy does not take effect for a class library until it is rebuilt using the latest **ih** or **xih** emitter. Until it is rebuilt, the DSOM 2.x policies still apply, and the above IDL modifiers need not be used.

Passing Objects by Copying

In most cases, objects passed as method parameters in remote calls are passed by reference. This is inappropriate for some objects, such as when the object is not a server or window object but rather encapsulates a data structure. DSOM provides a means of having these objects passed by copy. A copy of the object appears at the remote site: a by-copy **in** parameter is copied onto the server; a by-copy return-result; onto the client.

To pass an object parameter by-copy in a remote call, the implementor must ensure:

- The object's class is derived from **CosStream::Streamable** and overrides the **internalize_from_stream** and **externalize_to_stream** methods. The object implements methods to externalize it and to initialize it by reading from a stream.
- The client and server both load the actual DLL that contains the class implementation and not a stub DLL.

The first condition is checked by DSOM at run-time; the second, assumed but not checked. DSOM requires that the object to be passed by-copy be a local object; otherwise, a run-time exception occurs.

Consider that a class `C` is to be passed by-copy to a method `foo` and then returned from a method `bar`. The methods are defined as usual:

```
void foo(in C x);
C bar();
```

The following modifiers must be added to the implementation section of the SOM IDL:

```
foo : pass_by_copy_parameters = x;
bar : pass_by_copy_result;
```

A programmer may want by-copy argument-passing, but is uncertain that the objects are descended from **CosStream::Streamable** and are local objects. In this case, use the **maybe_by_value_parameters** and **maybe_by_value_result** SOM IDL modifiers in place of the **pass_by_copy_parameters** and **pass_by_copy_result** modifiers:

```
foo : maybe_by_value_parameters = x;
bar : maybe_by_value_result;
```

If the object is not a **Streamable** object, or the object is a proxy, an object-reference will be sent instead.

The **pass_by_copy_parameters** and **maybe_by_value_parameters** modifiers take a value that is a comma-separated list of parameter names. The **pass_by_copy_result** and **maybe_by_value_result** modifiers take no value. For more information on using SOM IDL modifiers, see **Implementation Statements** on page 132**.**

The default memory-management policy for by-value object parameters is similar to the policy for object references, except that **somFree** is used instead of **release** to release storage for a by-value object parameter. For an **in** parameter passed by-value, DSOM creates the object in the server's address space and then frees it in the server, using **somFree**, after the call has completed. The client retains ownership of the input object in the client address space. Similarly, an **out** or returned by-value parameter is freed by DSOM in the server as part of sending the response to the client. To allow the object in the server to retain ownership of a by-value parameter or return result, use the IDL modifiers **object_owns_result** or **object_owns_parameters**.

When an **inout** parameter is **passed maybe_by_value_parameters** modifier, in the client's address space DSOM uses **somFree** to free the input object and then allocates a new output object. In the server's address space, DSOM creates the input object and frees the output object, assuming that the target object has freed the input object. For example, DSOM allocates obj1, then the target object allocates obj2, frees obj1, and returns obj2, then DSOM frees obj2. For **out** parameters and objects returned by-value, DSOM frees the object in the server's address space. The client retains ownership of the object in the client address space.

Note: You cannot use pass by copy when the client program uses dynamic invocation, using **SOMObject::somDispatch** or the Dynamic Invocation Interface; static bindings are required to assist in copying the object parameter.

Passing Foreign Data Types

DSOM supports the marshaling for foreign data types, SOMFOREIGN types, but the object implementor must provide support. DSOM provides the following support techniques:

- opaque marshaler that marshals the binary representation of the foreign data
- dynamic foreign marshaling methods
- static foreign marshaling functions

In each case, the implementor of a module or interface must declare a type as foreign

```
typedef SOMFOREIGN a_foreign;
```

Consider a method `foo` that takes this foreign type as an argument, and a method `bar` that returns this foreign data type as result:

```
void foo(in a_foreign x);
a_foreign bar();
```

For the marshaler to manipulate a foreign data type, the following details about the type must be specified via SOM IDL modifiers:

- The **length IDL modifier**, the size in bytes, of the top-level, contiguous storage of the type. The default is 4 bytes, and the value must be non-zero.
- The storage class of the type, either **pointer** or **struct**, that indicates when pointers are introduced for parameters of that data type. The default is **pointer**. See **Introduced Pointers** on page 255 for additional information.
- The function or method DSOM can call to marshal, demarshal or free parameters of the data type. Use the **impctx** SOM IDL modifier to specifier either:
  - a static, C-callable marshaling function that DSOM calls for marshaling
  - the name of a class descended from **SOMDForeignMarshaler**, defined in **formarsh.idl**, that overrides the **marshal** method that DSOM invokes for marshaling.

Example: The provider of the `a_foreign` type above would provide, in the IDL, the following modifiers. Assume the storage class for the type is **struct** and the **length** of the top-level, contiguous storage is 4 bytes:

```
a_foreign : length = 4
a_foreign : struct;

/* and either */
a_foreign : impctx = "C,struct,opaque";
/* or */
a_foreign : impctx = "C,struct,dynamic(A_Foreign_Marshaler,foobar)";
/* or */
a_foreign : impctx = "C,struct,static(static_foreign_marshaler,0)";
```

The **struct** modifier indicates that the data type has the storage behavior of a **struct**. When a parameter of `a_foreign` type is passed, the caller passes a pointer to the **data** rather than the **data** itself.

The “C” in the **impctx** value refers to the language of the foreign data type.

The `struct` part of the **impctx** modifier value must be present if and only if the **struct** modifier was used, so the storage class of the foreign data type will be reflected by the TypeCode generated by the SOM compiler.

The `opaque`, `dynamic` or `static` part of the **impctx** modifier value tells DSOM how to marshal the foreign type. See each marshaling support technique, in the following paragraphs, for information on coding that specific marshaling technique.

Generic IDL for impctx: The **impctx** SOM IDL modifier tells DSOM the function or method to call to marshal, demarshal or free foreign data types.

foreign-type-name : impctx = “language, optional-sclass, marshaler-spec”

language
: the language of the foreign data type. For C or C++, the value “C” suffices.

optional-sclass
: the storage class of the foreign data type. The represented value can be either **struct** or **pointer**. **pointer** is the default value.

marshaler-spec
: the specification of the marshaling support technique. This value can be the opaque marshaler, dynamic marshaling methods or static marshaling functions. The generic syntax for these support techniques is:

```
opaque
dynamic(class-name,latent-param-name)
static(function-name,C-initializer-statement)
```

The description for the dynamic parameters *class-name* and *latent-param-name* and for the static parameters *function-name* and *C-initializer-statement* is with dynamic and static support methods respectively.

impctx Modifier with Opaque: The opaque part of the **impctx** modifier value specifies that DSOM should use a generic opaque-octet marshaler. DSOM copies the number of bytes specified by the **length** IDL modifier in messages between client and server.

a_foreign : impctx = “C,struct,opaque”

a_foreign : impctx = “C,opaque”

impctx Modifier with Dynamic: The dynamic part of the **impctx** modifier value tells DSOM to support marshaling with a dynamic foreign marshaling method.

a_foreign : impctx = “C,struct,dynamic(class-name,latent-param-name)”

a_foreign : impctx = "C,dynamic(class-name,latent-param-name)"

class-name
the name of a class descended from **SOMDForeignMarshaler**, defined in **formarsh.idl**, that overrides the **marshal** method. In the example, `A_Foreign_Marshaler`

latent-param-name
the value to be passed as the **latent_param** parameter value when DSOM invokes the **marshal** method. This value allows you to implement foreign marshaling for a variety of foreign data types using the same class. In the example, the string `foobar`.

impctx Modifier with Static: The static part of the **impctx** modifier value tells DSOM to support marshaling with static foreign marshaling functions.

a_foreign: impctx = "C,struct,static(function-name,C-initializer-statement)"

a_foreign: impctx = "C,static(function-name,C-initializer-statement)"

function-name
the name of the function that marshals, demarshals or frees the foreign data types. See **function-name Parameters** for the signature and descriptions of parameters.

C-initializer-statement
A C-expression that evaluates to a word-sized value, such as scalar or pointer. This value is passed as the first argument to the marshaling function designated by *function-name*, the *latent-param* parameter.

function-name Parameters: This function, `static_foreign_marshaler` in the **Example** on page 262, marshals, demarshals or frees foreign data types, similar to the **SOMDForeignMarshaler::marshal** method. This function has the signature:

```
void SOMLINK static_foreign_marshaler
  (
   void * latent_param,
   char * foreign_addr,
   som_marshaling_direction_t direction,
   som_marshaling_op_t function,
   CosStream_StreamIO * stream,
   Environment *ev
  );
```

latent_param
is the value specified by *C-initializer-statement* in **impctx** for the foreign data to be marshaled. This value allows you to implement foreign marshaling for a variety of foreign data types using the same function.

foreign_addr
is the address of the data to be marshaled.

direction
tell the marshaler the direction of the marshaling. It has IDL type:

```
enum som_marshaling_direction_t {SOMD_DirCall, SOMD_DirReply};
```

function
tells the marshaler whether to marshal, demarshal or free the data's non-contiguous storage. It has IDL type:

```
enum som_marshaling_op_t
```

```
{SOMD_OpMarshal, SOMD_OpDemarshal, SOMD_OpFreeStorage};
```

When called with **SOMD_OpFreeStorage**, the marshaling function should not free the top-level storage.

stream
the stream the marshaler uses to read or write the wire representation of the data

Using #pragma with Foreign Data Types: It is essential that the modifiers describing the SOMFOREIGN data types be declared before using the data type in a method declaration. This may require using a **#pragma** modifier rather than a modifier statement in the SOM IDL implementation section.

For example, the following IDL would not result in the correct C or C++ bindings for the `foo` method because the **struct** modifier of `a_foreign` follows the declaration of `foo`:

```
typedef SOMFOREIGN a_foreign;
void foo (in a_foreign x);
implementation {
     a_foreign : struct;
     ...
};
```

The correct bindings will be generated if this IDL is used:

```
typedef SOMFOREIGN a_foreign;
#pragma modifier a_foreign : struct;
void foo (in a_foreign x);
```

The **#pragma** modifier is useful for associating modifiers with foreign types declared outside the scope of any interface. IDL modifiers are within the interface statement; if there is no interface statement, you must use a **#pragma** modifier.

Destroying Remote Objects

When **somFree** is invoked on a proxy object, the proxy forwards the method to the target object and then destroys itself. **somFree** destroys the target and proxy object. When the C++ **delete** operator or the **somDestruct** method is used on a proxy, the operation is forwarded to the remote object and then executed on itself.

To state if the proxy or remote object is being deleted, the methods **somdTargetFree** and **release** should be used.

```
_somdTargetFree(car, &ev);
```

frees the remote `Car`, but not the proxy.

```
_release(car, &ev);
```

frees the proxy, but not the remote `Car`.

The **release** method implementation allows invocation not only on proxy objects but on any SOM object. This implementation gives applications local/remote transparency in destroying or releasing references to their objects. When **release** is invoked on a local object pointer, instead of a proxy object, the local object is unaffected.

Note: In DSOM 2.x, invoking **somFree** on a proxy object caused the remote object to be destroyed, but not the proxy. Application binaries compiled with DSOM 2.x will operate as before, but applications recompiled with DSOM 3.0 will destroy the proxy and the target object.

Some applications may require a different semantics for object destruction, in which some operation frees the object if it is local, but releases the proxy object only if the object is remote. The application designer has the responsibility of implementing such a destruction operation if one is required.

Inquiring about a Remote Object Interface or Implementation

A client may wish to inquire about the server implementation of a remote object. All objects in a server share the same implementation definition, described by an object of type **ImplementationDef**. When a proxy is obtained by a client, the client can inquire about the server implementation by obtaining its corresponding **ImplementationDef**. To get the implementation definition associated with a remote object, invoke the **get_implementation** method. If a program has a proxy for a remote `Car` object, it can get a proxy to the **ImplementationDef** object for the server with the method call:

```
ImplementationDef implDef;
Car car;
...
implDef = _get_implementation(car, &ev);
```

Once the **ImplementationDef** is obtained, the application can access its attributes using the corresponding **get** methods. Additional information about the **ImplementationDef** class is in **Implementation Definitions** on page 31.

When invoked on an object that does not reside in a server, an object local to a pure-client process, **get_implementation** returns NULL.

An application can query an object for its interface. **get_interface** invocation on a proxy returns a proxy to a remote **InterfaceDef** object that describes the interface supported by that object. If **SOMIR** is set, a **get_Interface** invocation on a local object returns a local **InterfaceDef** object; otherwise, NULL is returned. The **InterfaceDef** class is discussed in **Programming with the Interface Repository Objects** on page 340.

Working with Object References

There are three methods that can be used to work with object references (for example, proxy objects). Although these methods are defined in SOMDObject, they have been implemented so that they can be invoked on any SOMObject.

- The **duplicate** method is used to duplicate an object reference. If **duplicate** is invoked on a SOMDObject, then a new object is returned. The new object refers to the same remote object as the original. If **duplicate** is invoked on a local object, the same object is returned. The result of **duplicate** should be destroyed using the **release** method. When **release** is invoked on a SOMDObject, the SOMDObject, but not the object it refers to, is destroyed. When **release** is invoked on a local object, no action is taken (since **duplicate** invoked on a local object does not return a true copy).
- The **is_proxy** method can be used to determine whether an object is an instance of **SOMDClientProxy** or some subclass of **SOMDClientProxy**. If **is_proxy** is invoked on a local object, FALSE is returned.
- The **is_nil** method can be used to distinguish a NIL object reference from a valid object reference. A NIL object reference is one that does not refer to any local or remote object. Since **is_nil** is defined as a procedure method, it can be invoked on any object or on a NULL pointer. The constant OBJECT_NIL represents a NIL object reference.

Saving and Restoring References to Objects

Both proxy objects and pointers to local objects are a kind of “object reference”. An object reference contains information that is used to identify a target object. To illustrate, a pointer to a local object contains the actual physical address of the object. Similarly, a proxy object contains information needed to locate the target server and then the target object within that server.

In many applications, it is useful to convert object references to a string form (for example, to save references in a file system or to exchange object references with other application processes). DSOM defines a method for converting object references (both local object pointers and proxy objects) to an external form. This external form is a string that can be used by any process to identify the target object. DSOM also supports the translation of these strings back into the original local objects or equivalent proxies.

The **ORB** class defines two methods for converting between object references and their string representations. The IDL prototypes are as follows:

```
string object_to_string (in SOMObject obj);
SOMObject string_to_object (in string str);
```

The next example assumes that the target object is remote (*objref* is a proxy). The client program creates a `Car` object, generates a string corresponding to the proxy, and saves the string to a file for later use.

```
#include <stdio.h>
#include <somd.h>
#include <Car.h>
main( )
{
    Environment ev;
    Car car;
    string objref;
    FILE* file;
    SOM_InitEnvironment(&ev);
    SOMD_Init(&ev);

    /* create a Car object */
    car = somdCreate(&ev, "Car", TRUE);

    /* save the reference to the object */
    objref = _object_to_string(SOMD_ORBObject, &ev, car);
    file = fopen("/u/joe/mycar", "w");
    fprintf(file, "%s", objref);
...
```

Next is an example client program that retrieves the string and regenerates a valid proxy for the original, remote `Car` object (assuming the original, remote `Car` object still exists in the server or is a persistent object that can be reactivated).

```
...
    Environment ev;
```

```
    Car car;
    char buffer[256];
    string objref;
    FILE* file;

...
    /* restore proxy from its string form */
    file = fopen("/u/joe/mycar", "r");
    objref = (string) buffer;
    fscanf(file, "%s", objref);
    car = _string_to_object(SOMD_ORBObject, &ev, objref);
...
```

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.

When a string refers to a remote object, the **string_to_object** method always returns a new proxy object for that string. The returned proxy is not the same as the proxy passed to **object_to_string**, and repeated invocations of **string_to_object** each return different proxy objects. These duplicate proxies can be destroyed using the **release** method, described earlier.

When a string refers to a local object, the **string_to_object** method returns a pointer to the local object. If the object no longer exists, and is not a persistent object that can be re-activated, an exception is returned. Note that, if **object_to_string** is invoked on a local object within a client-only process (a process that is not a server), the resulting string has validity only as long as the process is active, and only within that process. When **object_to_string** is invoked on a local object within a server process, however, the resulting string can be distributed to other DSOM processes, which can then call **string_to_object** with the string to generate a proxy to the original object.

Note: In DSOM 2.x, the **string_to_object** method, when invoked within the server process in which the object resides, returned a SOMDObject (an object reference for the object) rather than the object itself. Similarly, **object_to_string** in DSOM 2.x required an input SOMDObject rather than a SOMObject. In the current release, for increased local/remote transparency for DSOM applications, the **string_to_object** and **object_to_string** methods map between a SOMObject (either a local object or a proxy object) and the string form of a reference to that object. (The intermediate transformation from a local **SOMObject** to/from a SOMDObject is done by DSOM.) Hence, server code need no longer invoke **SOMDServer::somdRefFromSOMObj** before invoking **object_to_string** and need no longer invoke **SOMDServer::somdSOMObjFromRef** after invoking **string_to_object**. Although these invocations are no longer necessary, however, they are harmless.

As with all object references, the lifetime of a string reference to an object in a server depends on the implementation of the server; if the server supports persistent objects, then the reference is valid even after the server process terminates. (The next time the reference is used by a client, the DSOM daemon will restart it, and the server can re-activate the referenced object.) The default server program, using the default server object, supports only transient objects (so that references are only valid for the lifetime of the server process). The main section "Basic Server Programming" describes how to implement servers that support persistent objects.

Note that the string form of an object reference (the result of calling **object_to_string**) should be considered opaque to application programmers. The only assumption that can be made about such a string is that it can be passed to **string_to_object** to locate the original object. It is possible for two different strings to refer to the same object; therefore it is not, in general, safe for an application to use the strings as unique object identifiers.

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. **SOMD_Uninit** frees the **ORB** object stored in **SOMD_ORBObject**. **SOMD_Uninit** should be called even if the client program terminates unsuccessfully; otherwise, system resources will not be released.

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.

Maintaining Thread Safety

The **SOMDClientProxy** class, which is used to generate client proxies, is thread-safe. Client applications can simultaneously invoke methods on the same proxy object from multiple threads. However, one thread should not change the state of the proxy (for example, by calling **somdReleaseResources** or **somdProxyFree**) while other threads are still using the proxy.

Although DSOM allows multiple invocations on the same proxy, the implementation of the target class must also be thread-safe. Class implementors are responsible for implementing thread-safe classes. If a target class is not thread-safe, then clients should treat proxies to that class as if they are not thread-safe.

In addition, although DSOM-generated proxy classes are thread-safe, user-written proxy classes aren't necessarily thread safe. It is up to the provider of an application-specific proxy class to provide thread safety for any data that is introduced by that proxy class.

Invoking **SOMD_Init** and **SOMD_Uninit** simultaneously from different threads is not supported. It is recommended that these functions be called from the main thread.

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 **Advanced Topics** on page 310.

Writing Distributed Workplace Shell Applications

The Workplace Shell provided with the current release of the OS/2 Toolkit is enabled as a DSOM server. This means that Workplace Shell applications can be written as DSOM client programs that manipulate the OS/2 desktop from a separate process.

The Workplace Shell provides a utility, **wpdsinit**, for starting and stopping the DSOM daemon used by the Workplace Shell and the Workplace Shell's server thread. (The server thread is initially disabled, to improve performance and footprint of the Workplace Shell when the server thread is not in use.) The Workplace Shell server thread uses a specialized **SOMDServer** subclass, called **wpdServer**. The Workplace Shell also provides its own Interface Repository and Implementation Repository, with the Workplace Shell classes and its server thread registered therein. The Workplace Shell Interface Repository is typically found in `\TOOLKIT\SOM\COMMON\ETC\SOM.IR`, and the Workplace Shell Implementation Repository files are typically found in `\OS2\ETC\DSOM`.

The Workplace Shell Implementation Repository has the DSOM 2.x format and must be converted to the DSOM 3.0 format with the **migimpl3** tool before you can use it with DSOM 3.0. This should be done after Naming/Security Service configuration (using the **som_cfg** tool). See Migrating DSOM 2.x Implementation Repositories to the current DSOM format for more information on the **migimpl3** utility. Be sure to use the **-l** option to **migimpl3** when converting the Workplace Shell Implementation Repository.

Workplace Shell DSOM client programs require special initialization. In particular, the application must merge the SOM Class Manager with the Workplace Shell Class Manager and initialize all application classes used by the program.

The Workplace Shell Programming Guide provided by the OS/2 Toolkit includes a sample application. It also includes more information on customizing the Workplace Shell server thread and on starting the Workplace Shell's server thread either automatically at boot time or programmatically.

The Workplace Shell uses the environment settings in effect at boot time. Therefore, if you are using the Workplace Shell as a DSOM server, you must set all DSOM-related environment variables in the **config.sys** file. The variables include **SOMENV**, **PATH**, **LIBPATH** and **DPATH**. In the configuration file, set the **SOMDDIR** setting that the Workplace Shell uses; the setting is indicated by the **config.sys** setting of **SOMENV**. You can set the SOMIR setting in either the **config.sys** file or in the configuration file indicated by the **config.sys** setting of **SOMENV**. (If set in both, the **config.sys SOMIR** setting takes precedence.)

All SOMobjects 3.0 libraries (DLLs) that DSOM requires must appear in a directory indicated by the boot-time setting of **LIBPATH**. The directory that contains the SOMobjects 3.0 libraries must appear (in LIBPATH) prior to any directory containing SOM/DSOM 2.1 or Warp's SOM/DSOM libraries. Similarly, all executables (including **somdd**) that DSOM 3.0 requires must appear in a directory indicated by the boot-time setting of **PATH**, which must precede any directory that contains SOM/DSOM 2.1 (or Warp's SOM/DSOM) executables.

Compiling and Linking Clients

All client programs must include the main DSOM header file (**somd.h** for C or **somd.xh** for C++) in addition to any header files they require from application classes. This DSOM header file will #include all other DSOM header files that are needed. Before compiling the DSOM application, be sure to run the appropriate header-generation script (for example, **somcorba, somstars,** or **somxh**).

All DSOM client programs must link to the SOMobjects Toolkit library: **libsomtk.a** on AIX and **somtk.lib** on OS/2 and Windows NT. For more information, see **Compiling and Linking** on page 95.

Designing Local/Remote Transparent Programs

You want to write a program that runs successfully whether it is working with local or with remote objects. This section gives guidelines to follow that will help you design programs with local/remote transparency. The goal of local/remote transparency is to let most of a program's code and design be independent of the location of objects. You should have local/remote transparency in mind when you design the program. The following are the overall principles for local/remote transparency; the rest of this section describes more specific ways to achieve transparency.

- SOM and DSOM also provide non-transparent interfaces for efficiency or convenience.
- To be transparent, programs must adhere to good object-oriented programming practices, in particular encapsulation and polymorphism.
- Any SOM/DSOM interfaces that reveal implementation information will not be transparent (for example, **somGetInstanceSize**). Exceptions to the above rule are made only to preserve binary compatibility with existing interfaces.

The following sections of this chapter describe various issues of local/remote transparency and provide guidelines for writing transparent code.

Class Objects

Class objects are one of the primary areas where SOM does not support local/remote transparency. A class object can be accessed remotely but it will not be local/remote transparent. Given a reference to a remote class, you can invoke the `somNew` method and create an instance object of the remote class in the remote location of the class, much as you can do with a local class. However, the object creation methods that require a pointer to the memory to be used for a new object do not work because there is no meaningful way to pass a memory pointer to a remote object. Although directly interacting with the class object this way is not transparent, you can wrap a class object in a local factory object and have the factory offer a set of creation methods that are transparent. See factory discussion below.

Class objects anchor the implementation of an object. For example:

- They must respond to questions about the object's memory requirements.
- They must participate in class hierarchies to achieve an object's implementation. Many of the methods that go between class objects in a class hierarchy cannot be distributed because they involve parameters that are difficult to marshal, such as procedure pointers.
- They must be different for two objects that have different implementations, even when the objects are largely or completely type compatible.

Guideline

Because it is neither possible nor desirable to make class objects completely transparent, class objects should be used rarely and with great care in code that you want to be transparent. To access the class of an object, use the **SOM_GetClass** macro instead of the **somGetClass** method because, **somGetClass** is forwarded to the remote object for backward compatibility reasons and it provides a handle to the remote class rather than the local class of the proxy object.

Object Creation

Although class objects are not fully local/remote transparent, it is reasonable to use a reference to a remote class object to create instances of the class at the remote site.

Remote class objects cannot be declared statically like a local class object; you must use explicit calls to create the instances. For example:

```
Foo* makeFooInstance (SOMClass *aFooClass, long arg)
{
Foo *obj = (Foo *) _somNewNoInit(aFooClass);
aFooInitMethod(obj, NULL, arg);
return obj;
```

The above procedure assumes that Foo has an initializer method, `aFooInitMethod`, that needs to be called on all new instances, and that it takes one argument, a long, that is passed in on the call to `makeFooInstance`. This procedure works equally well for creating a local or a remote object; it depends on which kind of class instance is passed in.

The class of the object created is not the same as the remote class object. It is the proxy class of the proxy object that represents the remote instance object. Use a method like **somdGetTargetClass** to enquire about the class of a remote object.

Using Factories to Create Objects

Use factory objects to create objects that are local/remote transparent. Factory objects are not class objects but are objects that have methods that create object instances. It is easy to design code that uses factory objects to create transparent object instances.

The following example, corresponding to the class-based example, shows the use of a factory object:

```
void aProcedure (environment *ev,
FooFactory *aFooFactory)
{
Foo *obj = _newFoo1(aFooFactory, ev, 45);
if (ev->_major != NO_EXCEPTION) {
/* handle error */
}
/* do something with obj */
_somDestruct(obj);
}
```

The procedure above is not concerned with where the object instance is created. In fact, the same procedure could be called with many different instances of FooFactory each of which creates instances at different locations, including one that creates local instances.

It is easy to implement a factory object that just wraps a particular class object. SOMObjects Release 3.0 has two types of factories; a user-defined factory, and a class object that masquerades as a factory. When a client tries to find a factory that creates a given type of object, for example, `Foo`, either an instance of a user-defined factory specified in the *idl*, for example, `MyFooFactory`, is returned, or the `Foo` class object is returned. To create an instance, use a method like **somNew** when the factory is a class object. Use a factory-specific creation method when the factory is user-defined. The client is aware which type of factory is returned because of the IDL interface. Typically, the factory classes are not related; a factory for a class will have no class hierarchy relationship to a factory for one of its subclasses.

Using Factories While Controlling Memory Allocation

Factory objects can also be used even when the client code needs to control memory allocation. However, this requires the factory to provide transparent versions for all creation methods such as **somNew**, **somRenew** or **somRenewNoInit** The following pseudocode illustrates how to encapsulate a class object in a factory object and how to create instances even while controlling memory allocation.

```
interface SOMFactory : SOMObject {
```

SOMFactory introduces factory methods for creating instances of a single class. You can provide storage for these instances. SOMFactories will only create objects of a single class. Once such a factory is created, the class of objects that it creates cannot be changed. SOMFactory is an abstract class.

```
long somFactoryGetInstanceSize();
long somFactoryGetAlignment();
SOMObject somFactoryRenewNoInit(in void* memory);
SOMObject somFactoryRenew(in void* memory);
SOMObject somFactoryNewNoInit();
SOMObject somFactoryNew();
};

interface SOMFactoryFromClass : SOMFactory {
void somFactoryInitializeWithClass(
inout somInitCtrl ctrl,in SOMClass clsObj);
```

The example above is an initializer method that takes a class object or a proxy to a class object as an input parameter. Note: usually class objects and proxies for class objects cannot be interchanged, but this factory takes the necessary steps to allow this. Objects created by `SOMFactoryFromClass` instances are guaranteed to respond to all the methods that `clsObj` instances respond to.

```
};
```

Assuming that `SOMFactoryFromClass` has three instance variables:

long instanceSize;

boolean isLocal;

SOMClass *clsObj;

The implementation of some of the methods may look like this:

```
void somFactoryInitializeWithClass(SOMFactoryFromClass *somSelf,
      Environment *ev,
      somInitCtrl* ctrl,
      SOMClass* classObj)
{
  SOMFactoryFromClassData *somThis; /* set in BeginInitializer */
  somInitCtrl globalCtrl;
  somBooleanVector myMask;
```

```
  SOMFactoryFromClass_BeginInitializer_
    somFactoryInitializeWithClass;

  SOMFactoryFromClass_Init_SOMFactory_somDefaultInit(somSelf, ctrl);

  _clsObj = classObj;
  if (!classObj)  {
    _instanceSize = 0; _isLocal = FALSE;
    return;
  }
  if (_somIsA(classObj, _SOMDClientProxy)) {
  /* proxy to remote class object */

    SOMClass *proxyClass;
    string classname;

    classname = _somGetName(classObj);
    proxyClass = somdCreateDynProxyClass(ev, classname, 0, 0);
    ORBfree(classname);
    if (proxyClass) {
      _instanceSize = _somGetInstanceSize(proxyClass);
      _isLocal = FALSE;
    }
    else {  /* could not create proxy class */
      _clsObj = (SOMClass *)NULL;
      _instanceSize = 0;
      _isLocal = FALSE;
    }
  } else {  /* not a proxy */       _isLocal = TRUE;
    _instanceSize = _somGetInstanceSize(_clsObj);
  }
}

/* The following creates an initialized object using the memory */
/*provided. */

SOMObject*  somFactoryRenew(SOMFactoryFromClass *somSelf,
            Environment *ev,
            void* memory)
{
    SOMFactoryFromClassData *somThis=
              SOMFactoryFromClassGetData(somSelf);
    if (!_clsObj || !memory)
```

```
        return ((SOMObject *)NULL);
      if (_isLocal) {
        return _somRenew(_clsObj, memory);
      } else {
        /* create a remote, initialized object and copy the proxy */
        SOMObject *newObj = _somNew(_clsObj);
        memcpy(memory, newObj, _instanceSize);
        SOMFree(newObj);  /* don't _release or _somFree! */
        return memory;
      }
  }
```

Guideline

In summary, there are two ways to create objects: Class-based and Factory-based. Of these, only the latter is suitable for local/remote transparent programming.

Gaining Access to Existing Objects

Existing objects are accessed through various object services. Generally these services are local/remote transparent if the client does not depend on the implementation class of the objects it accesses. Typically, a client searches the name space (talks to a name server) to find an object that has all the desired properties.

Guideline

The client code should make no assumptions about an object's implementation beyond the properties it specifies as parameters to the name service's search.

Proxy versus Object Destruction

When a destruction method is called on a proxy object, does it destroy the proxy, the target or both? SOM has methods for each: **release**, **somdTargetFree** and **somDestruct** respectively. **somFree**, although ambiguous, can be resolved by the object; the object knows what to do. Call **somDestruct** when you want to be sure that both the proxy and the remote object are destroyed for sure and you have no other intentions. In a client/server environment, the object implementor may choose to assign different meanings to **somFree** (for example to implement reference count-based freeing). While you could do this by overriding either **somDestruct** or **somFree**, it is better to use **somFree** and keep the sure fire semantics of **somDestruct** intact. Also note that while **somDestruct** is a true deinitializer (it can walk up the multiple inheritance hierarchy correctly), **somFree** is really only a deallocator.

The CORBA **release** method is available on local objects and object references. (The method is introduced by **SOMDObject**, but has been implemented in such a way that it can be invoked on any object.) **release** performs no operations on a local object, as there are no resources associated with local object references. To be transparent, **release** should be called on each object reference.

somdTargetFree forwards the **somFree** method call to the remote object, but the proxy object is not destroyed.

somDestruct on a proxy gets forwarded, and it destroys both the remote object and the local proxy if its parameter requires.

somFree has an ambiguity that is resolved by the object. **somFree** releases any resources associated with the object reference, and is forwarded to the remote object when a proxy object is involved. Factories create objects that assign the same meaning to **somFree**. However, object and factory designers can use other means of determining the meaning of **somFree**. Class factories produced by the SOMClass metaclass assign **somFree** the **somDestruct** meaning. **somFree** applies uniformly to class and non-class objects.

Guideline

Use **somFree** uniformly. You can override **somFree** and implement some kind of reference-counting before destroying the object. Use **somFree** on local and remote objects and classes. Security and integrity considerations might require disallowing **somFree** on remote classes.

Memory Management of Parameters

SOMObjects 2.1 introduced new modifiers in IDL to make memory management of parameters explicit. They default to the SOMobjects 2.0 policies, for backward compatibility. There is a **memory_management=corba** modifier to specify CORBA 1.1 parameter and return result memory management. In SOMObjects 3.0 the default memory management policy is corba. All parameter and return result memory is uniformly caller-owned, and the caller is eventually responsible for freeing it.

Parameters for which the *object owns* memory are inherently problematic for local-remote transparency. In the local case, the clients typically get pointers into the data structures owned by the object. This breaks encapsulation in addition to being an integrity concern.

Guideline

The following are the guidelines for local/remote transparent programming with respect to parameter and return result memory management:

- Don't use object-owned parameters. If you must, limit yourself to object-owned IN parameters.
- Don't retain a reference or alias to memory whose ownership you have given away.
- Don't give away ownership of memory unless it is allocated with **SOMMalloc** so the receiver can assume the use of **SOMFree** to free it.
  - This simplifies life for the ownership receiver, and also the ownership can be transferred several times.
  - This applies to all object-owned IN parameters and all the caller-owned parameters that the ORB is supposed to free.
- Use **SOMFree** rather than **ORBfree** for parameter cleanup, after initializing your program with **SOMD_NoORBfree**.
- Use **somExcetpionFree** to free exception structures from both local and remote calls, after initializing your program with **SOMD_NoORBfree**.

A corollary of these guidelines is that all object-owned IN parameters must be created and freed using **SOMMalloc** and **SOMFree** respectively. Object-owned is a SOM feature, not a CORBA feature. Another corollary is that because **SOMFree** and **somExceptionFree** do a shallow free, you must walk the structures and free any embedded objects. The typecode-based free function, **SOMD_FreeType**, eases this task.

Conclusion: The simplest rule for transparent programming is: Stick to CORBA rules and use **SOMMalloc** and **SOMFree** to manage your parameter and return result memory.

Distribution Related Errors

Working with a proxy object introduces error possibilities that don't exist when working with the object directly, especially if the proxy is for an object running in another process or on another machine. For example, the real object may go away (because of an error or because it was explicitly destroyed) while the client still has a proxy for it. Or, the real object may become unreachable because of transport failures.

Check for error conditions by examining the Environment (ev) parameter. Distribution-related errors are reflected through the same parameter.

Guideline

To be local/remote transparent, the client code should handle whatever errors it can cope with and handle the rest (distribution-related errors) gracefully using a default error handler.

Generating and Resolving Object References

All CORBA objects are accessed through object references. The ORB fabricates externalizable object references and resolves them. In DSOM, the ORB object uses the SOMOA (SOM Object Adapter) and **SOMDServer** to generate and resolve object references.

Passing an object as a parameter to a remote object automatically causes a reference to be created for the object by the object adapter. This works as long as the program passing the object reference has an object adapter (has server capability). A process that does not have an object adapter cannot pass a reference for one of its local objects to a remote object. Such processes are pure clients.

Guideline

From a pure-client process, don't write out stringified object references. They are good only as long as the ORB object that created them lives.

Support for CORBA Specified Interfaces to Local Objects

All CORBA specified interfaces that apply to objects work on local SOM objects. These are the methods defined on Object in the CORBA documentation. For local objects these methods are defined as follows:

- **get_implementation** returns NULL, if the object is in a pure client process. If the object is in a server process, it returns the **ImplementationDef** object for the server.
- **get_interface** works as defined if object is registered in the IR; otherwise, returns NULL.
- **is_nil** is a procedure method. It return true if the pointer is NULL, and false, otherwise.
- **duplicate** copies the pointer to the object.
- **release** does nothing.
- **create_request** works as defined; the DII interface works with local objects.

Guideline

OBJECT_NIL is defined by CORBA as an object reference that does not refer to any object. To maintain local/remote transparency, do not assume that OBJECT_NIL is mapped to a null pointer.The code `if (!objref) ...` is not transparent. Instead, use the form `if _is_nil(objref) ...`. Also, in general it is not a good idea to do checks like `if (objref == OBJECT_NIL) ...`, because the inequality among two object references is not a guarantee that they are referring to two different objects (according to CORBA). Of course, if they are equal then they do refer to the same object. You should test for null object reference by calling the **_is_nil** function. Do not call **release** on a null object reference. For example, you can guard the release call as in

```
if (!_is_nil(objref)) _release(objref); .
```

Data Types not Supported In Distributed Interfaces

There are types not fully defined in IDL or that cannot be marshaled by DSOM. Therefore, you can provide custom marshalling support for any such types. These types must be declared as SOMFOREIGN in the IDL. Also, there must be IDL modifiers indicating the length in bytes of the type, its storage class (mainly to know if there is an introduced pointer), and the name of the custom marshaling function. The custom marshaling can be done either through a static function, a dynamic method, or a DSOM default. The DSOM default marshaling strategy for SOMFOREIGN types is to simply marshal the binary representation of the type, using the specified length or a default size.

When SOMFOREIGN types are passed by copy, you control the implementation of externalization methods and copy constructors. Therefore, you must ensure that either method produces semantically equivalent object copies.

SOM Objects That Do not Have IDL Interfaces

SOMobjects does not support distribution of these objects unless their interfaces are either in the Interface Repository or the marshaling plan is compiled into the **.dll**.

Procedure Methods

SOMobjects does not support remote access to procedure methods.

Global Variables

SOMobjects does not support remote access to global variables, or any way of linking global variables across address spaces.

Class Data

SOMobjects does not do anything special for class data to support local-remote transparency.

Class Methods

The class of a proxy object is a local proxy-class object. Any class methods will be executed locally and, therefore, will not generally be transparent. Also, to reach the class object use the **SOM_GetClass** macro instead of `somGetClass` method.

Direct Instance Data Access

SOM does not support remote access to data, only to remote method calls.

Passing Objects by Value

SOMObjects supports IDL interface declarations that specify that an object is to be passed by value. This must work in the local case as well as the remote case. With SOMObjects 2.1, SOM supports pass by value to local targets as part of the DTS extensions based on copy initializers (which may be defaulted). SOMObjects 3.0 extends this capability to calls to remote targets through the externalization framework. Different results may be produced in the local versus the remote case because of the difference between a copy initializer and the externalization framework. This is under the control of the object designer. For transparency you must ensure that they produce semantically equivalent copies.

Objects passed by value in C++ have somewhat strange semantics. For example, if a formal parameter expects an object of type `Animal` and the actual argument happens to be of type `Dog`, then C++ makes a copy of `Dog` and truncates the copy after the `Animal` portion of data. Technically speaking, it has all the state data for an Animal and thus it should behave like an animal. However this breaks polymorphism because the overriding behavior of `Dog` is not seen in the callee. It is seen when passed by reference.

In general, when a class is subclassed, the subclass programmer might implement a superclass's abstractions in terms of its own data and may not keep the superclass's data up-to-date. Truncating an object (to match a formal parameter that is of the superclass type) in such cases breaks the object. Respecting polymorphic behavior means ensuring that polymorphic methods operate on the right data. Any direct access to data members in C++ does not respect encapsulation and it also does not respect polymorphic behavior: that is breaking encapsulation implies breaking polymorphic behavior. Of course, the converse is not true: breaking polymorphic behavior does not imply breaking encapsulation. For example, not keeping the parent class's data up-to-date and trying to truncate the object to make it an instance of the parent class does not break encapsulation, but it does break polymorphic behavior.

Guideline

The following are the guidelines for passing objects by value with local/remote transparency.

- Use **pass_by_copy_parameters** modifier in the IDL, if you want C++ By-Value semantics.
- Use **maybe_by_value_parameters** modifier, if you primarily want efficiency (both local and remote).
- An object can be passed by value only if the object supports COSS Streamable interface. Make sure that externalize and internalize produce a semantically equivalent copy to a copy constructor on the object.
- Make sure that the code does not break polymorphism (implies encapsulation as well).

Object Invocation: Synchronous, Oneway, Deferred Synchronous and Asynchronous

The method invocation syntax for synchronous, deferred synchronous (through CORBA DII), and one way calls are exactly the same for both local and remote objects, and therefore SOM is local/remote transparent in these cases.

Guideline

For using deferred synchronous invocations, use CORBA request objects. These work for local and remote objects. Use the returned values only after calling `get_response` method.

Summary of Local/Remote Guidelines

The following is a quick summary of the guidelines for designing transparent local/remote programs:

General

- Methods with externally visible side effects, for example, calling **printf** from inside a method, can't be transparent.
- For transparency with respect to parameters passed by value, all ways of creating copies, including copy constructors, externalization, and custom marshaling must produce semantically equivalent data as in the local case.
- In general, any communication between an object and outside (other objects included) that is not through methods is not guaranteed to be transparent. Some examples are:
  - Aliasing: Caller and Callee sharing pointers.
  - Global variables access
  - Instances communicating via common class data members.
  - Direct instance data access from outside the object
- Procedure methods are not forwarded to remote objects.
- Methods added dynamically to a SOM class object are not forwarded.
- Non-polymorphic code is not guaranteed to be local/remote transparent. (for example, copy constructors, assignment operators and externalization methods should all respect polymorphism.)

Specific

- Classes are implementation means for objects. In local/remote transparent programs, use them rarely and with great care, and use only those operations on them that are transparent.
- Use factories to create objects.
- An existing object's access through various object services is transparent. No other assumptions (besides supporting the object services) about their implementation should be made by clients.
- To destroy objects use `somFree` uniformly on both instances and classes. Use `release` uniformly on local and remote object references.
- To check if an object reference is null, use **_is_nil** function. Do not make assumptions on the representation for OBJECT_NIL.
- For memory management of parameters, stick to CORBA rules (that is, Caller-Owned) and use `SOMFree` uniformly. Free exception structures using `somExceptionFree` uniformly.
- Anticipate distribution-related error returns and handle them gracefully.
- If in a pure client process, don't externalize stringified object references and don't pass object references in remote calls. Passing objects by value is allowed.
- To pass an object by value or copy, make sure that its class supports the COSS Streamable interface and it produces semantically equivalent copy through the copy constructors and externalization.
- For using deferred synchronous invocations, use CORBA request objects. The returned values should be used only after calling the **get_response** method.
- For writing transparent assignment operator methods make sure that they do not break the polymorphism of the from object. That is, make all accesses to the from object's state either method calls or attribute access calls.

Method Classification for Local/Remote Transparency

This section lists and classifies the SOM kernel and DSOM methods and tells how they relate to local/remote transparency. This section defines the terms used to classify methods, tells how the terms can be combined, then lists the methods. Methods that are local only or are deprecated are not listed.

Terms Used in Method Classification

The following terms are used to classify or explain methods:

Forward
: When a method is applied to a local proxy object, it is said to be **forwarded** if it is sent on to the server object that the proxy object is representing. If the method is not special it causes the same method to be invoked on the server object. If it is special, then one or more different methods may be invoked on the server object. If a method is not forwarded, then it is handled locally and no message is sent to the server object

Transparent
: A method is transparent if the caller does not need to know that the method's target object is a proxy. When a method is transparent, it should be practical to invoke it on a collection of objects, some of which are local and some of which are remote (represented by proxies) without concern for local versus remote.

Special
: A method is special if it must be implemented by the proxy and possibly converted into other method calls.

A method can be classified into one or more of the above groups as follows:

Forwarded, Transparent: T his is the default for all developer-defined methods. Almost all methods are just forwarded to the remote object, and neither the client nor the server programmer needs to make any special provision for these methods. However, many of the SOMObject base class methods do not fall into this category because of their special nature of being part of the runtime.

Not Forwarded, Transparent: A small number of methods are not forwarded to remote objects because their definition makes more sense in the local context. The best examples of this category are certain debugging methods (such as `somPrintSelf`) and methods that inquire about an object's class or class properties. Class inquiries must be handled by the proxy object's class because they are supposed to reveal information about the implementation or type of the actual object they are applied to (abstraction cannot be used for these methods). For example, a client might ask for an object's class and then ask the class object about its instances' size. In this case the client must be given the size of the proxy object, as anything else could lead to memory errors. Even though these methods are not forwarded, they are still transparent (except for some of the debugging methods for which transparency is poorly defined). In fact, to forward them would make them non-transparent.

Forwarded, Non-Transparent: Some methods are forwarded to remote objects and they work equally for local and proxy objects. However there may be observable side effects (needing special care to free parameter and return result memory) depending on whether the target is local or remote. These cases typically arise with older SOMObjects methods that are preserved for backward binary compatibility.

Forwarded, Transparent, Special: A few methods must be processed locally and then sent on, possibly as different methods.

Not Forwarded, Transparent, Special: An advanced developer may place some methods in this category in the definition of an caching proxy class. That is, the developer might actually execute some method locally while preserving the exact semantics of remote execution.

Variable, Transparent, Special: A few methods have a variable definition: sometimes they are forwarded and sometimes they are not. This may depend on the proxy object or it may depend on the arguments passed to the method.

The following tables list the functions and their classifications.

Table 4. SOM Kernel Functions Classification

Method Name	Classification
somApply	Variable, Transparent, Special
somClassResolve	Not Forwarded, Transparent
somIsObj	Not Forwarded, Transparent
is_nil	Not Forwarded, Transparent
somParentNumResolve	Not Forwarded, Transparent
somParentResolve	Not Forwarded, Transparent
somResolve	Not Forwarded, Transparent
somResolveByName	Not Forwarded, Transparent

Table 5. SOMObject Methods Classification

Method Name	Classification
create_request	Not Forwarded, Transparent
create_request_args	Not Forwarded, Transparent
duplicate	Not Forwarded, Transparent
get_implementation	Forwarded, Transparent
get_interface	Forwarded, Transparent
is_proxy	Not Forwarded, Transparent
release	Not Forwarded, Transparent
somCastObj	Forwarded, Transparent
somDefaultAssign	Forwarded, Transparent
somDefaultConstAssig	Forwarded, Transparent
somDefaultConstCopyInit	Forwarded, Transparent
somDefaultConstVAssig	Forwarded, Transparent
somDefaultConstVCopyInit	Forwarded, Transparent

Table 5. SOMObject Methods Classification

Method Name	Classification
somDefaultInit	Forwarded, Transparent, Special
somDefaultCopyInit	Forwarded, Transparent
somDefaultVAssign	Forwarded, Transparent
somDefaultVCopyInit	Forwarded, Transparent
somDestruct	Forwarded, Transparent, Special
somDispatch	Variable, Transparent, Special
somClassDispatch	Not Forwarded, Transparent
somDumpSelf	Not Forwarded, Transparent
somDumpSelfInt	Not Forwarded, Transparent
somFree	Variable, Transparent, Special
somGetClass	Forwarded, Transparent
somGetClassName	Forwarded, Non-Transparent
somGetSize	Not Forwarded, Transparent
somInit	Forwarded, Transparent
somIsA	Not Forwarded, Transparent
somIsInstanceOf	Not Forwarded, Transparent
somPrintSelf	Not Forwarded, Transparent
somResetObj	Forwarded, Transparent
somRespondsTo	Not Forwarded, Transparent
somUninit	Forwarded, Transparent

Table 6. SOMClass Methods

Method	Classification
somCheckVersion	Forwarded, Transparent
somDescendedFrom	Forwarded, Transparent
somGetInstanceSize	Forwarded, Transparent
somGetName	Forwarded, Non-Transparent
somGetNumMethods	Forwarded, Transparent
somGetNumStaticMethods	Forwarded, Transparent
somGetParents	Forwarded, Transparent

Table 6. SOMClass Methods

Method	Classification
somGetVersionNumbers	Forwarded, Transparent
somNew	Forwarded, Transparent
somNewNoInit	Forwarded, Transparent

Table 7. SomClassMgr Methods

Method Name	Classification
somClassFromId	Forwarded, Transparent
somFindClass	Forwarded, Transparent
somFindClsInFile	Forwarded, Transparent
somGetInitFunction	Forwarded, Non-Transparent
_get_somInterfaceRepository	Forwarded, Transparent
_set_somInterfaceRepository	Forwarded, Transparent
_get_somRegisteredClasses	Forwarded, Transparent
somLocateClassFile	Forwarded, Non-Transparent
somRegisterClass	Forwarded, Transparent
somSubstituteClass	Forwarded, Transparent
somUnloadClassFile	Forwarded, Transparent
somUnregisterClass	Forwarded, Transparent
somClassFromId	Forwarded, Transparent
somFindClass	Forwarded, Transparent
somFindClsInFile	Forwarded, Transparent

Table 8. DSOM Functions

Method Name	Classification
send_multiple_requests	Forwarded, Transparent, Special
somdCreate	Not-Forwarded, Transparent

Table 9. ImplementationDef Methods

Method Name	Classification
_get_config_file	Forwarded, Transparent

Table 9. ImplementationDef Methods

Method Name	Classification
_get_impl_id	Forwarded, Transparent
_get_impl_alias	Forwarded, Transparent
_get_impl_def_class	Forwarded, Transparent
_get_impl_def_struct	Forwarded, Transparent
_get_impl_program	Forwarded, Transparent
_get_impl_flags	Forwarded, Transparent
_get_impl_server_class	Forwarded, Transparent
_get_protocols	Forwarded, Transparent
_get_svr_objref	Forwarded, Transparent

Table 10. ImplRepository Methods

Method Name	Classification
add_class_to_impldef	Forwarded, Transparent
add_impldef	Forwarded, Transparent
delete_impldef	Forwarded, Transparent
find_all_aliases	Forwarded, Transparent
find_all_impldefs	Forwarded, Transparent
find_classes_by_impldef	Forwarded, Transparent
find_impldef	Forwarded, Transparent
find_impldef_by_alias	Forwarded, Transparent
find_impldef_by_class	Forwarded, Transparent
remove_class_from_all	Forwarded, Transparent
remove_class_from_impldef	Forwarded, Transparent
update_impldef	Forwarded, Transparent

Table 11. ORB Methods Classification

Method Name	Classification
object_to_string	Not-Forwarded, Transparent
string_to_object	Not-Forwarded, Transparent

Table 12. Principal Methods Classification

Method Name	Classification
userName	Forwarded, Transparent
hostName	Forwarded, Transparent

Table 13. Contained Methods Classification

Method Name	Classification
describe	Forwarded, Non-Transparent
within	Forwarded, Non-Transparent

Table 14. Container Methods Classification

Method Name	Classification
contents	Forwarded, Transparent
describe_contents	Forwarded, Non-Transparent

Table 15. InterfaceDef Method Classification

Method Name	Classification
describe_interface	Forwarded, Non-Transparent

Table 16. Repository Methods Classification

Method Name	Classification
lookup_id	Forwarded, Transparent
lookup_modifier	Forwarded, Transparent
release_cache	Forwarded, Transparent

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
- Demarshaling client requests into method invocations on their local objects
- Marshaling method invocation results into responses to clients
- 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 libraries (DLLs) that the implementor wants to distribute. Optionally, the programmer can customize the behavior of the default server program by supplying an application-specific *server class*, derived from SOMDServer, to alter the behavior of the server’s *server object*. (By default, the class of the server object is SOMDServer.) 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 and Windwos NT.

Note: When using iostream in servers, the stream library is just written and linked in C. It does not perform the iostream initialization usually done behind the scenes in a C++ main () program. To avoid this, write your own server program in C++.

A second server program, **somossvr**, is also provided, to be used with the SOMobjects object services. See **Chapter 5, Object Services Server** on page 35 of *Programmer’s Guide for Object Services* for more information on the somossvr server program.

Some applications may require additional flexibility or functionality than what is provided by the SOMobjects server programs. 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), derived from the abstract BOA interface
- The application-specific server object (an instance of either SOMDServer or a class derived from SOMDServer)

Server Implementation Definition

When a server is registered with DSOM (for example, via the **regimpl** utility), the administrator specifies various run-time characteristics of the server. These characteristics make up the server’s implementation definition. An implementation definition is represented by an object of class **ImplementationDef**, whose attributes describe a server’s ID, user-assigned alias, program pathname, the class of its server object, whether or not it is multi-threaded, and so forth. Implementation definitions are stored in the Implementation Repository, a database that is represented by an object of class **ImplRepository**.

Implementation IDs uniquely identify servers (implementation definitions) within the network, and are used as keys into the Implementation Repository when retrieving the **ImplementationDef** for a particular server. The server's user-assigned *alias* is unique within a particular Implementation Repository, and can also be used as a key for retrieving a particular server's implementation definition. **ImplementationDef** objects can be retrieved from the Implementation Repository by invoking methods on the **ImplRepository** object.

An implementation ID identifies a *logical server*, and the associated **ImplementationDef** object describes the current implementation of that logical server. It is possible to change the implementation characteristics of a (logical) server, even to the point of using a completely different server program, by simply changing the attributes of the server's **ImplementationDef** object in the Implementation Repository. The location of the logical server can be changed by moving the server's **ImplementationDef** object from one Implementation Repository to another (on a different host).

When a server is initialized, it must retrieve a copy of its **ImplementationDef** object from the Implementation Repository (by invoking a method on the global **ImplRepository** object, **SOMD_ImplRepObject**), and keep it in a global variable, **SOMD_ImplDefObject**. (See the example server program later in this section.) Client-only programs can leave the **SOMD_ImplDefObject** variable set to NULL. This variable is used by the DSOM run time within a server process (for example, to know whether the server should be multi-threaded). Because the sever's **ImplementationDef** object represents the server's identity to DSOM, and because this object is stored in a variable global to the a process, it is currently not possible for a single DSOM process to have multiple server identities.

By referring to the **ImplementationDef** object, DSOM allows users to customize many aspects of the behavior of a server without writing any server code.

See **Registering Servers and Classes** on page 31 for details on server registration. Two registration methods are described: *manual* (via the **regimpl** command line utility), and *programmatic*, via **ImplRepository** methods.

SOM Object Adapter

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 (described below) to create and resolve DSOM references to local objects, and to dispatch methods on objects.

There is one SOMOA object per server process. (The SOMOA class is implemented as a single instance class.) This object must be explicitly created by the server program and stored in the global variable **SOMD_SOMOAObject**.

Server object (SOMDServer)

Each server process contains a single server object. By default, the server object is an instance of class SOMDServer. The purpose of the server object is to allow applications to customize the way the default server program generates/resolves object references, creates factories, and dispatches methods, with a minimal amount of new code.

The server object has the following responsibilities for managing objects in the server:

- Provides an interface to the DSOM run time for dynamic factory creation. Factories are necessary for basic object creation.
- Provides an interface to the DSOM run time for creating and resolving object references (which are used to identify an object in the server).
- Provides an interface to the DSOM run time for dispatching requests.

The class of the server object to be used with a server is contained in the server's **ImplementationDef**. The SOMDServer class 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 in order to tailor the server object functionality to a particular application. Also see **Chapter 5, Object Services Server** on page 35 of *Programmer's Guide for Object Services* for a discussion of a subclass of SOMDServer, **somOS::Server**, provided by SOMobjects for use with the object services.

The server object is created by the SOMOA object in response to the **impl_is_ready** method invocation, which must be made explicitly by the server program. The SOMOA object examines the **ImplementationDef** object (which must have been previously stored in the **SOMD_ImplDefObject** global variable by the server program) to determine what type of server object to create. (The default is **SOMDServer**.) The SOMOA object then stores the server object in the global variable **SOMD_ServerObject**.

Server Activation

Most server programs can be activated either

- Automatically by the DSOM daemon, **somdd**
- 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. 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.

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
```

The example server shown in **Example Server Program** on page 290 illustrates how the server program can use the implementation ID 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.

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.

Rather than being registered (for example, via **regimpl**) before being run, 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 as described in **DSOM Configuration** on page 18.

Example Server Program

Shown below are the key statements of a simple DSOM server program. Actual server programs would contain additional code for error handling and application-specific processing. (See **Chapter 5, Object Services Server** on page 35 of *Programmer's Guide for Object Services* for an example server program designed to work with the SOMobjects object services.) Each portion of the sample program is discussed in the topics following the example code.

```
#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]);

   /* Create the server's Object Adapter: */
   SOMD_SOMOAObject = SOMOANew();

   /* Notify the DSOM daemon that the server is ready
    * to process requests from clients: */
   _impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
   /* Go into an infinite loop of processing requests: */
   _execute_request_loop(SOMD_SOMOAObject, &ev, SOMD_WAIT);

   /* Server cleanup code: */
   /* tell DSOM (via SOMOA) that server is now terminating */
```

```
    _deactivate_impl(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);

   SOMD_Uninit(&ev);
   SOM_UninitEnvironment(&ev);
}
```

Initializing a Server Program

The following topics discuss the initialization steps for a DSOM server program that are illustrated in the preceding example.

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 own implementation with the Implementation Repository dynamically (as shown in **DSOM Configuration** on page 18).

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]);
...
}
```

A server can also use its alias to get its **ImplementationDef** from the Implementation Repository, using the **find_impldef_by_alias** method. See the DSOM `animal` sample program for a sample server program that can be started either by the DSOM daemon or from the command line (with a **-a** option to designate the server's alias).

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 then sets up the necessary communications resources for receiving incoming messages, which it registers with the DSOM daemon. Once the DSOM daemon has been notified of the server's ports, it assists client applications in *binding* (that is, establishing a connection) to that server.

The **impl_is_ready** method also causes the server object to be created. The server object's class (for example, **SOMDServer**) is specified by the **impl_server_class** attribute of the server's **ImplementationDef**. The server object can be referenced through the global variable **SOMD_ServerObject**.

When the server invokes **SOMOA::impl_is_ready**, if the server's **ImplementationDef::config_file** attribute differs from the current **SOMENV** setting, the contents of the configuration file named by **ImplementationDef::config_file** will be read, any DSOM run-time initialization performed during **SOMD_Init** will be refreshed, and for the duration of the server process the setting of **ImplementationDef::config_file** will be prepended to the current **SOMENV** setting.

As part of **SOMOA::impl_is_ready**, if the server has been registered as a secure server, the server will contact the security server to initialize the security service run-time within the server to enable the server to reject requests from unauthenticated clients. For additional information.

Note: A server program should not attempt to export object references or use any other Object Adapter services until it has invoked **impl_is_ready**, as some crucial server initialization steps are performed at that time. The only exception is the **activate_impl_failed** method.

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:

```
_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 or when the server is terminated using the **SOMDServerMgr::somdShutdownServer** method. If SOMD_NO_WAIT is passed, it will return when there are no more outstanding messages to be processed.

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 to your application servers that call the **deactivate_impl** method upon normal or 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 the **SOM_UninitEnvironment** call, which frees any memory associated with the specified Environment structure.

Managing Objects in the Server

The following topics discuss the appropriate classes and their methods that can be used to manage various kinds of objects in the server.

Object References (SOMDObjects)

Within the **execute_next_request** method (and hence, within **execute_request_loop**), the DSOM SOMOA object receives client requests, transforms them into method calls on local objects, and then returns the results to the client. When a dispatched method returns an *object* as a result or an **out** parameter, the server must return to the client process a *proxy* to the actual object, not merely a pointer to the object in the server (because the pointer will be meaningless in the client's address space).

Recall 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 the responsibility of the DSOM run-time environment to ensure that proxies, rather than simply object pointers, are returned to clients.

The SOMOA object (**SOMD_SOMOAObject**) and the server object (**SOMD_ServerObject,** an instance of SOMDServer or a subclass) work together to perform this service. Whenever a result from a remote method call includes a SOMObject, the SOMOA object invokes a method (**somdRefFromSOMObj**) on the **SOMD_ServerObject**, asking it to create an object reference (SOMDObject) from the SOMObject. Similarly, when an object reference is detected within an incoming request from a remote client, the SOMOA is responsible for converting it back into a pointer to the local object to which it refers. It does this with the help of the **SOMD_ServerObject**, by invoking a method (**somdSOMObjFromRef**) that performs the inverse of the **somdRefFromSOMObj** method.

An object reference is an exportable handle to an object. DSOM implements object references as separate objects, of class SOMDObject. Proxy objects (objects of class **SOMDClientProxy**) are also examples of object references. Hence, **SOMDClientProxy** is a subclass of SOMDObject. The difference between **SOMDClientProxy** and SOMDObject is as follows:

- An instance of **SOMDClientProxy** resides in a client process, representing a remote object, and the client can invoke methods on the remote object by invoking them on the proxy.
- An instance of SOMDObject, by contrast (referring to the same target object), resides only in the server process (where the target object resides). Unlike a proxy object, it does not support the same interface as the target object. It is simply an object in the server that the DSOM run time uses to create the corresponding proxy object in the client process.

The way in which objects are mapped to object references, and vice versa, can be customized by an application by subclassing SOMDServer and providing new implementations of the **somdRefFromSOMObj** and **somdSOMObjFromRef** methods. For example, to support persistent objects (objects whose state persists between activations of the server process), the generic server program could use a subclass of **SOMDServer** written to map from persistent objects to references (SOMDObjects) that contain some kind of persistent object identifier, and vice versa.

Shown below are the IDL declarations of the SOMDServer methods that map between local SOMObjects and object references (SOMDObjects):

```
SOMDObject somdRefFromSOMObj(in SOMObject somobj);
SOMObject somdSOMObjFromRef(in SOMDObject objref);
```

The SOMOA object invokes **somdRefFromSOMObj** on the **SOMD_ServerObject** each time a local object is to be returned as the result (or out parameter) of a remote method call that the server has dispatched. The SOMOA object invokes **somdSOMObjFromRef** on the **SOMD_ServerObject** for each object reference found as a parameter in an incoming request. (If the input object to **somdRefFromSOMObj** is already an object reference, either a SOMDObject or a **SOMDClientProxy**, the default implementation does no conversion. If the input object to **somdSOMObjFromRef** is a **SOMDClientProxy** rather than a SOMDObject, signifying that the target object is not local to the server, then no transformation is done.)

SOMobjects provides a special subclass of SOMDServer, called somOS::Server, to be used with all the SOMobjects object services (Naming, Security, Persistent Object Service, LifeCycle and Transactions). The **somOS::Server** provides persistent object references and other services.

ReferenceData

The data contained in an object reference (whether a SOMDObject or **SOMDClientProxy**) is used in two ways:

- to assist a client process in locating the server process where the target object resides
- to assist the server process in locating or activating the target object

The information used for the first task is generated by DSOM when any object reference is created. The information used for the second task is called ReferenceData:

- ReferenceData is the information a server uses to identify the target of a remote call.

ReferenceData is represented in an IDL sequence of up to 1024 bytes of information about the object. This sequence may contain the object's location, state, or any other information that the application server needs to locate or activate a target object in the server.

When a subclass of SOMDServer is written to override **somdRefFromSOMObj** and **somdSOMObjFromRef**, the new implementation can change the way that ReferenceData is generated during the construction of new object references (and, inversely, the way in which the ReferenceData in existing object references is interpreted).

- An application-specific implementation of the **somdRefFromSOMObj** method will typically
  - generate application-specific ReferenceData, containing information necessary to identify the local object (such as a persistent object identifier)
  - invoke the **create** method on SOMOA, passing the constructed ReferenceData, to complete the task of constructing a new object reference (SOMDObject).
- Similarly, an application-specific implementation of the **somdSOMObjFromRef** method typically
  - invokes the **get_id** method on SOMOA to get the ReferenceData associated with an object reference (SOMDObject)
  - map the ReferenceData to a local SOMObject, in an application-specific way.

The SOMOA interface supports two operations for creating object references: **create** and **create_SOM_ref**. (DSOM 2.x also provided a third method, **SOMOA::create_constant**. This method is deprecated in the current release.) An application-specific implementation of the method **SOMDServer::somdRefFromSOMObj** would use one or more of these methods to create a new object reference (SOMDObject).

Creating Simple SOM Object References

The default implementation (SOMDServer's implementation) for **somdRefFromSOMObj** uses the **SOMOA::create_SOM_ref** method to return a simple reference for the SOMObject. The **create_SOM_ref** method creates a simple DSOM reference (SOMDObject) for a local SOM object. The reference is "simple" in that, unlike a reference created by the **create** method, 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 **SOMOA::get_SOM_object** method.

The **SOMOA::is_SOM_ref** method can be used to tell if the object reference passed to **somdSOMObjFromRef** was created (in **somdRefFromSOMObj**) using **create_SOM_ref** or not (and hence, whether **get_SOM_object** can be used to retrieve the original SOMObject).

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();
```

Creating Application-Specific Object References

Application-specific server objects (instances of a subclass of **SOMDServer**), within the implementation of **somdRefFromSOMObj**, may elect to use **create**, rather than **create_SOM_ref**, to construct an object reference if the application requires ReferenceData to be stored in the object reference.

For example, a server object for a server of persistent objects might elect to store persistent object identifiers in the **ReferenceData** of a reference to a persistent object, so that when the object reference is converted back to a local object pointer (by **somdSOMObjFromRef**), the persistent identifier can be extracted from the **ReferenceData** of the object reference and used to reactivate the persistent object.

The **ReferenceData** associated with a SOMDObject created using **create** can be retrieved by invoking the **get_id** method on the SOMOA object.

The IDL SOMOA interface declarations of **create** and **get_id** are presented below.

```
/* From the SOMOA interface */


   sequence <octet,1024> Referencedata;
   SOMDObject create(in ReferenceData id,
                        in InterfaceDef intf,
                        in ImplementationDef impl);


   ReferenceData get_id(in SOMDObject objref);
```

Note: DSOM 2.x provided another method, **SOMOA::create_constant**, for creating object references. This method is deprecated in the current release. In DSOM 2.x, the **create** and **create_constant** methods served different purposes. In the current release, however, the methods are equivalent, except for servers whose **ImplementationDef** object (from the Implementation Repository) specifies an *object reference table filename*. (To retain the DSOM 2.x semantics for the **create** method, simply specify a reference table filename for the server using **regimpl**'s **-f** *filename* argument. Otherwise, **SOMOA**'s implementation of the create method will simply invoke **SOMOA::create_constant**.) In addition, the DSOM 2.x methods **SOMOA::change_id** and **SOMOA::create_constant**, and the use of the Object Reference Table, have been deprecated. (They are still supported in this release, but may be eliminated in a future release.) Eventually, the non-CORBA-compliant **SOMOA::create_constant** will be eliminated in favor of the CORBA-compliant **create** method. Servers that need persistent storage of object-reference data, such as that previously provided by the Object Reference Table, should implement this functionality in an application-specific subclass of **SOMDServer**, or use the **somOS::Server** class for this purpose.

DSOM 2.x semantics for the create and create_constant methods: Servers whose **ImplementationDef** objects specify an object reference table filename will continue to receive the DSOM 2.x semantics for the **create** method. To specify an object reference table filename for a server, use the **regimpl** option **-f***filename* when registering or updating the server. Note that the filename specified is otherwise ignored; object reference table data will be stored in the directory designated by SOMDDIR. The filename specified to **regimpl** is simply used as a flag to indicate that the server needs to use the DSOM 2.x semantics for the **create** method.

When using DSOM 2.x semantics, 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 **ReferenceData** associated with an object reference created by **create** can be changed (via the **change_id** method). This is because the **create_constant** method stores the **ReferenceData** directly in the object reference (**SOMDObject**), while the **create** method stores the **ReferenceData** in a *ReferenceData table* associated with the server (and stores a key into the **ReferenceData** table in the object reference).

When using the DSOM 2.x semantics, references created with **create_constant** can be distinguished from references created with **create** by using the **SOMDObject::is_constant** method. Hence, a server can use the **is_constant** method to determine whether the ReferenceData associated with a given object reference can be changed using **change_id**.

Because object references constructed with **create** (when using DSOM 2.x semantics) support changeable ReferenceData**,** but object references constructed with **create_constant** do not, one might wonder when it would advantageous to use **create_constant** versus **create** with the DSOM 2.x semantics. Recall that, to allow a server to change the **ReferenceData** associated with an object reference after the reference has been distributed to clients, invocations of **create** add entries to a table called the *ReferenceData Table*. (The ReferenceData goes into the table, while a key into the table is stored in the object reference.) 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 different SOMDObjects are created. Each of these must have its own entry in the **ReferenceData** table, so that their **ReferenceData** can be changed independently. If a server using **create** wishes to avoid cluttering up the **ReferenceData** table with multiple **ReferenceData** entries for the same object, it must maintain a table of its own to keep track of the SOMDObjects it has already created, to avoid calling **create** twice with the same arguments. In addition, the application must devise some mechanism for removing obsolete entries from the table.

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 is willing to give up changeable ReferenceData so that it does not have to maintain its own table of SOMDObjects, nor pay the penalty of cluttering up the ReferenceData table with multiple entries for the same object. A server using **create_constant** also avoids the overhead of updating a persistent table (which requires a disk access) each time an object reference is created.

To summarize the different uses for **create_SOM_ref**, **create**, and **create_constant**:

- use **create_SOM_ref** to create transient object references that are valid only for the lifetime of the in-memory object.
- use **create** or **create_constant** to create persistent object references to either transient or persistent objects. (Remember that **create_constant** is deprecated.)
- use **create** in a server whose ImplementationDef specifies an object reference table filename to create persistent object references to either transient or persistent objects, with the ability to change the ReferenceData associated with the object without invalidating outstanding object references

Example: Writing a Persistent Object Server

This section shows an example of how to provide a server class implementation for persistent SOM objects. The example shows how a server class might use and manage ReferenceData in object references to find and activate persistent objects. 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 `SOMPServer` that is derived from SOMDServer**.** The `SOMPServer` class overrides the two SOMDServer methods **somdRefFromSOMObj** and **somdSOMObjFromRef**.

The IDL specification for `SOMPServer` follows:

```
interface SOMPServer : SOMDServer {
  #ifdef __SOMIDL__
    implementation {
        somdRefFromSOMObj : override;
        somdSOMObjFromRef : override:
    };
  #endif
};
```

The following two procedures override SOMDServer's implementations of the methods **somdRefFromSOMObj** and **somdSOMObjFromRef**:

```
SOM_Scope SOMDObject SOMLINK
              somdRefFromSOMObj(SOMPServer somSelf,
                                Environment *ev,
                                SOMObject obj)
{
  SOMDObject objref;
  Repository repo;
/* is obj persistent */
  if (object_is_persistent(obj, ev)) {
    /* Create an object reference based on persistent ID. */
    ReferenceData rd = create_refdata_from_object(ev, obj);

repo = SOM_InterfaceRepository;

    InterfaceDef intf =
       _lookup_id(repo,ev,
                  somGetClassName(obj));
    objref = _create (SOMD_SOMOAObject, ev, &rd,
                              intf, SOMD_ImplDefObject);
    _somFree(intf);
    _somFree(repo);
    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. This implementation invokes **SOMOA::create** to create a SOMDObject. The prerequisites for asking SOMOA to create a SOMDObject are:

- Some ReferenceData to be associated with the SOMDObject. The application must define a function (such as the function `create_refdata_from_object` above) to retrieve the persistent object identifier (PID) from the object and coerce it into the datatype ReferenceData.

- An InterfaceDef that describes the interface of the object. The InterfaceDef is retrieved from the SOM Interface Repository using the object's class name as key.
- An ImplementationDef that describes the server containing the object. The ImplementationDef is held in the variable **SOMD_ImplDefObject** that is set when the server process is initialized.

With these three arguments, SOMOA's **create** is called to create the SOMDObject.

The preceding example assumes that there is some method or function available to the server, such as `object_is_persistent` above, to determine whether an object is persistent and hence has a persistent object identifier.

The next example overrides **SOMDServer**'s implementation of **somdSOMObjFromRef**:

```
SOM_Scope SOMObject SOMLINK
              somdSOMObjFromRef(SOMPServer somSelf,
                                 Environment *ev,
                                 SOMDObject objref)
{  SOMObject obj;
      if (_is_nil(objref, ev))
      return (SOMObject *) NULL;
   /* Make sure this isn't a local object or proxy: */
   if (!_somIsA(objref, _SOMDObject) || _is_proxy(objref, ev))
      return objref;

   /* test if objref is mine */
   if (!_is_SOM_ref(objref, ev)) {
      /* objref was mine, activate persistent object myself */
      ReferenceData rd = _get_id(SOMD_SOMOAObject, ev, objref);
      obj = get_object_from_refdata(ev, &rd);
      SOMFree(rd._buffer);
   } else
     /* it's not one of mine, let parent activate object */
     obj = parent_somdSOMObjFromRef(somSelf, ev, objref);
   return obj;
}
```

`SOMPServer`'s implementation of **somdSOMObjFromRef** must determine whether the SOMDObject (*objref*) is one that it created (that is, one that represents a persistent object), or is one that was created by the SOMDServer code (its parent), via the parent-method call in **somdRefFromSOMObj**, above. This determination is done via the **is_SOM_ref** method. If the **is_SOM_ref** method fails, then the `SOMPServer` can 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 an application-provided function such as `get_object_from_refdata`. The implementation of a `get_object_from_refdata` function might convert the **ReferenceData** to a persistent object identifier (PID) and use the PID to locate (and activate, if necessary) the persistent object.

Observe that, if a server class is not directly subclassed from **SOMDServer** (but from some other subclass of **SOMDServer**), then **is_SOM_ref** may not be sufficient for determining whether to make a parent-method call within **somdSOMObjFromRef**. In general, some application-specific technique may be required for distinguishing between the object references created by different subclasses of **SOMDServer** (such as using special “flags” embedded within the **ReferenceData**).

To summarize, the following guidelines apply when implementing overrides of **SOMDServer**::**somdSOMObjFromRef**:

- Insure that local objects, proxy objects, NULL pointers, and OBJECT_NIL are all handled appropriately. (NULL pointers and OBJECT_NIL can be detected using the **is_nil** method.)
- Do not attempt to invoke **SOMOA::get_id** on a local object, proxy object, NULL pointer, or OBJECT_NIL.
- Do not attempt to interpret the **ReferenceData** of a SOMDObject that was not created by your corresponding override of **SOMDServer::somdRefFromSOMObj** (that is, one that was created by a parent implementation via a parent-method call).

Validity Checking in somdSOMObjFromRef

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);
        }
    }
```

Customizing Factory Creation

The SOMDServer class defines a method for the creation of SOM object factories in a server. The **somdCreateFactory** method is invoked by the DSOM run time when a client requests that a factory be dynamically created in the server:

```
SOMObject somdCreateFactory(in string className, in
                            ExtendedNaming::PropertyList props);
```

The purpose of this method is to allow applications to subclass SOMDServer to customize the way in which factories are associated with classes or the way factories are created. See **Finding a SOM Object Factory** on page 245 for more information about factories.

The **somdCreateFactory** method creates a factory object that can create objects of the specified class. Two kinds of factories can be created and returned by the default implementation of **somdCreateFactory**: an application-specific factory or a SOM class object. The default implementation of **somdCreateFactory** uses the IDL modifier **factory** to map from the given class name to an application-specific factory class name. For example, if instances of class `Car` are created by class `CarFactory`, the IDL for the interface `Car` could include the modifier:

```
factory=CarFactory;
```

When the **factory** modifier is specified in the IDL for the specified class, **somdCreateFactory** will create an instance of the factory class using **somNew**. No initializers will be called by **SOMDServer**, although the client is free to invoke any method on the returned factory object (including an initializer). Because the default implementation of **somdCreateFactory** invokes **somNew** on each invocation, there is potential for factory objects to accumulate in the server. To avoid this accumulation, the factory class can be given the **SOMMSingleInstance** metaclass.

If the **factory** modifier is not specified in the IDL for the specified class, the default implementation of **somdCreateFactory** returns the SOM class object (such as, the `Car` class object) as the default factory.

Applications may choose to override **somdCreateFactory** to take advantage of the *props* parameter. This parameter is a complete list of the properties associated with the factory entry in the Naming Service. The **somdCreateFactory** method may also be overridden to support factory classes that require application-specific initializers.

Customizing Method Dispatching

After SOMOA (with the help of the local server object) has resolved all the SOMDObjects present in a request received from a client, it is ready to invoke the specified method on the target. Rather than invoking **somDispatch** directly on the target, it calls the method **somdDispatchMethod** on the server object. The parameters to **somdDispatchMethod** are the same as the parameters for **SOMObject::somDispatch**. (See *Programmer's Reference for SOM and DSOM* for a complete description.)

```
void somdDispatchMethod(in SOMObject somobj,
                        out somToken retValue,
                        in somId methodId,
                        in va_list ap);
```

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, so that the server object can perform application-specific computations before or after the method is dispatched.

For example, the following override of somdDispatchMethod simply displays a message just before and just after dispatching each application method in the server:

```
SOM_Scope void SOMLINK somdDispatchMethod (MySOMDServer somSelf,
                             Environment *ev, SOMObject somobj,
                        somToken *retValue, somId methodId, va_list
ap)
{
     somPrintf("About to invoke method %s\n",
               somStringFromId(methodId));
     MySOMDServer_parent_SOMDServer_somdDispatchMethod(somSelf,
               ev, somobj, retValue, methodId, ap);
     somPrintf("Method dispatch complete.\n");
}
```

Identifying the Source of a Request

CORBA 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, 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.

The value of the **userName** attribute is the user name with which the client logged in on the client's machine. If the user has not logged in (or if LOGIN_INFO_SOURCE is set to a null (blank) in the SOMobjects configuration file), the user is treated as an unauthenticated user and the **userName** attribute will be an empty string (“”).The **hostName** attribute is obtained (by DSOM) from the **HOSTNAME** environment variable (or the **HOSTNAME** setting in the [somd] stanza of the configuration file), if the user is authenticated. If the user is not authenticated, then the **hostName** attribute will be an empty string (“”).

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 is typically made either by the target object or by the server object, when a method call is received.

Note: CORBA defines a **TypeCode** of tk_Principal, 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.

A more extensive client-authentication service is provided by the SOMobjects Security Service. The Security Service allows a server to be registered (via **regimpl**) as a secure server. A secure server will automatically reject all requests from users that are not authenticated.

Compiling and Linking Servers

The server program must include the **somd.h** header file (for C) or **somd.xh** (for C++). Server programs must link to the SOMobjects Toolkit library: **libsomtk.a** on AIX, and **somtk.lib** on OS/2 and Windows NT.

For more information, see **Compiling and Linking** on page 195.

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 easy to use SOM classes in DSOM-based applications. In fact, many existing SOM class libraries may be used in DSOM applications without any special coding or recoding for distribution. DSOM uses a generic server program (**somdsvr**) that uses SOM and SOMOA to load SOM class libraries when an object of a particular class is created or activated.

Registering Servers and Classes on page 31 discusses how to register a server implementation comprised of a DSOM generic server process and at least one SOM class libraries.

Role of somdsvr

somdsvr provides basic server functionality. This program constantly receives and executes requests, via an invocation of the **SOMOA::execute_request_loop** method, until the server is stopped. Some requests result in the creation of SOM objects. **somdsvr** finds and loads the DLL for the object’s class, if it is not loaded.

When **somdsvr** functionality is insufficient for a particular application, application-specific server programs can be developed. For example, one application may want to interact with a user or I/O device between requests.

Role of SOMOA

SOMOA 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 simple 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 factory objects via **somdCreateFactory**. This method is called by the DSOM run time when a client requests a SOM object factory that must be created dynamically.
- Mapping between SOMDObjects and SOMObjects via **somdRefFromSOMObj** and **somdSOMObjFromRef.** These methods are invoked on the server object by the SOMOA when:
  - Objects are to be returned to clients
  - Incoming requests contain object references
- Dispatching remote requests to server process objects via **somdDispatchMethod** when the method is ready to be dispatched, respectively

By partitioning out these functions into the server object, the application can customize them without building object adapter subclasses. SOMDServer can be subclassed by applications that want to manage object location, object activation and method dispatching.

These features of SOMOA and SOMDServer enable existing SOM classes, that were written for a single-address space environment, to be used unchanged in a DSOM application.

Implementation Constraints

somdsvr, SOMOA and SOMDServer make it easy to use SOM classes with DSOM. If any part of the class implementation was 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:

- **Printing to Standard Output.** Any text printed by a method will appear at the server, not the client. The server may not be attached to a text display device or window, so the text may be lost. Any textual output generated by a method should be returned as an output string.
- **Creating and Deleting Objects**. Methods that create or delete objects may have to be modified if the created objects are intended to be remote.
- **Using procedure Methods.** Methods having the **procedure** SOM IDL modifier cannot be invoked remotely using DSOM, for these methods are called directly rather than via SOM's normal method resolution mechanisms.

In addition to the coding practices that do not port to a distributed environment, there are other restrictions DSOM imposes:

- **Using void* Types.** DSOM can make remote invocations only on methods whose parameter types are completely defined SOM IDL types. A type is completely defined if it contains no **void*** or **somToken** types.
- **Packing of Structures used as Method Arguments.** When building a SOM class library to be distributed using DSOM, avoid using compiler options that pack or optimize **struct**s, including reordering of **struct** members, or **union**s. For data structures that require nonstandard alignment, it is preferable to declare the types as SOMFOREIGN and to provide custom marshaling support for those types.

Some applications may need to associate specific identification information with an object, to support application-specific object location or activation. In this case, an application server should create object references explicitly by using the **SOMOA::create** method. These calls should be placed in a subclass of SOMDServer.

Using Other Object Implementations

As an ORB, DSOM must support a wide range of object implementations, including non-SOM implementations, for example, a print spooler application where the implementation may be provided by the operating system. The methods on the print queue may be executable programs or system commands. In this example, the application may need to participate in object identification, activation or request dispatching.The server can supply a customized server object that works with SOMOA for this purpose.

Wrapping a Printer API

Below is an example showing how an API could be wrapped as SOM objects. Although this API is simple, readers should understand this process to create more sophisticated applications.

The API wrapped consists of two system calls; the first asks for a file to be printed on a specific printer, the second, to cancel the file currently being printed on device `printername`.

```
print /D:printerName filename
print /D:printerName /C
```

An IDL interface `Printer` is declared in the module `PrinterModule`. The `Printer` interface wraps the two system calls.

```
module PrinterModule {
  interface Printer : SOMObject {
    attribute string printerName;
    void print(in string fname);
    void cancel();
    #ifdef __SOMIDL__
    implementation {
      printerName: noset;  // memory to be allocated
      releaseorder : _get_printerName,_set_printerName,print,cancel;
};
    #endif
```

```
  };
};
```

The Printer interface defines an attribute, printerName, that identifies the printer and is set when it is created. The operations, print and cancel, correspond to the system commands the interface encapsulates.

The next three method procedures show how the interface is implemented for _set_printerName, print and cancel. _set_printerName is implemented to make a copy of the input string passed by the client.

```
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 this example must be provided by the application, but it is not defined in the example.

Building and Registering Class Libraries

The generic server uses SOM's run-time facilities to load class libraries dynamically. DLLs should be created for the classes. During development of the DLL, remember the following:

- Export a routine called **SOMInitModule** in the DLL, which will be called by SOM to initialize all the class objects implemented in that library. For more information, see **Specifying the Initialization and Termination Function** on page 215. There is a special emitter to generate the **SOMInitModule** function.
- 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. The **dllname** modifier is used by the SOM run time for dynamically finding and loading the library containing the implementation of a SOM class.
- 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
```

Note: If the classes are not compiled into the Interface Repository, DSOM may still be able to invoke methods of the class remotely. However, the SOM class manager will be unable to dynamically load a class's library unless the DLL name is the same as the class name, because SOM will be unable to dynamically discover the value of the **dllname** modifier of the class to be loaded. For other situations in which DSOM requires a class interface to be compiled into the Interface Repository, see **Registering Class Interfaces** on page 30.

- Put the DLL in one of the directories listed in **LIBPATH** for AIX or OS/2 or PATH for Windows NT.

Running DSOM Applications

Before any DSOM processes are started, the DSOM environment should be configured appropriately, as discussed in **Step 6. Configuring User Applications** on page 29. Of particular importance are the **SOMENV** and **SOMIR** environment variables. For workgroup (cross-machine) applications, **SOMDPROTOCOLS** must be set appropriately in the configuration file, and the **HOSTNAME** settings must also be set appropriately, for each protocol specified by **SOMDPROTOCOLS**. The Naming Service and Security Service must have been configured, using the **som_cfg** tool. All server programs to be used by the application must be registered in the Implementation Repository, using the **regimpl** tool.

Running the DSOM Daemon

To run a DSOM application, the DSOM daemon, **somdd,** must be started on each server machine. Client machines do not require an active DSOM daemon.

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 and Windows NT. The **somdd** command has the following syntax:

somdd [**-q**]

where the optional **-q** flag signifies *quiet* mode. By default, **somdd** produces a `ready` message when the DSOM daemon is ready to process requests,. In quiet mode the `ready` message does not appear.

The **somdd** daemon is responsible for *binding* a client process to a server process and will activate the desired server if necessary. The DSOM run time on behalf of client programs contacts the DSOM daemon on the server's machine to retrieve the server's communications address (a port). The daemon activates servers dynamically as separate processes.

On AIX, when running **somdd** under a different identity than client programs, insure that the files created in the /tmp directory are writable by client programs, by setting the umask appropriately before starting **somdd**.

The DSOM daemon should not be terminated unless all other DSOM processes are stopped. The daemon can be restarted without reconfiguration of the Naming Service.

DSOM provides the **somdDaemonReady** function for determining programmatically whether the DSOM daemon is running and ready to process requests.

Running DSOM Servers

Once the **somdd** daemon is running, application programs can be started. If the application uses the generic SOM server, **somdsvr**, the server can be started either from the command line or the application can allow the server to be started automatically (by **somdd**) on 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`.

SOMobjects Developer Toolkit provides another server program, **somossvr**, which is required by all SOM object services. This server program can also be started from the command line (using the same command-line arguments as **somdsvr**) or automatically by **somdd**. The only exception is that the first time **somossvr** is run using a particular server ID/alias, it must be started from the command line and passed the **-i** argument. The **-i** argument instructs the server to initialize its persistent storage. If **somossvr** is not given the **-i** argument the first time it is run, it will terminate with an error. See **Chapter 5, Object Services Server** on page 35 of *Programmer's Guide for Object Services* for more information on the **somossvr** server program.

The naming server and security server (registered by the **som_cfg** tool) require the **somossvr** server program. However, these servers need not be initialized manually; the **som_cfg** utility initializes them.

After a server program has been terminated, it can be restarted without restarting the DSOM daemon, provided that the server unregistered itself with the daemon during termination. This unregistration occurs in the default server program (**somdsvr**) and in the object services server (**somossvr**), unless the server terminated due to a system trap or (on AIX) a "kill -9" command. Application-specific server programs should insure that unregistration occurs by invoking the **deactivate_impl** method on the **SOMOA** object as part of an exit handler.

Running the Client Program

Once the DSOM daemon is running on the server machines, the client program can be started. If the server to be used by the client is not already running, it will be started automatically by the DSOM daemon when the client attempts to use it. After the client program ends, the server and daemon will continue to run, accepting connections from new clients, until they are explicitly terminated.

Running Workgroup Applications

Before running a workgroup (cross-machine) application, insure that the following configuration steps have been taken:

- The **SOMDPROTOCOLS** setting of both client and server contain a common entry, and that common protocol supports cross-machine communication (the **SOMD_IPC** protocol does not).
- The server has been registered (using **regimpl**) on the machine on which it will run. A server cannot be registered from a different machine. The **HOSTNAME** and **SOMDPORT** settings in effect at the time the server was registered must be the same as those in effect when the server (and its associated daemon) are executed.

Freeing Interprocess Communication Resources on AIX

DSOM allocates interprocess communication (IPC) resources during execution. Normally, these resources are freed and returned to the system when **SOMD_Uninit** is called. On AIX, however, if a DSOM process terminates without calling **SOMD_Uninit**, the resources often remain allocated.

The **cleanipc** script can be used to free the IPC resources allocated to a particular AIX user, as follows:

cleanipc [*userId*] - Frees resources for the specified user

If the *userId* parameter is not specified, **cleanipc** by default uses the environment variable USER.

The cleanipc script should be run only after all DSOM processes have ended. A limitation of cleanipc is that it is not able to distinguish between resources that were created by DSOM and resources created by other products. As a result, **cleanipc** may affect other applications.

Advanced Topics

Applications that have unusual requirements may benefit from some of the following advanced capabilities.

Peer versus 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, this is best accomplished by dedicating a separate process thread that handles DSOM communications and dispatching.

Multi-Threaded DSOM Programs

In a system that supports multi-threading, 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 **Basic Server Programming** on page 286.

```
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);

/* tell DSOM (via SOMOA) that server is now terminating */
    _deactivate_impl(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
}
```

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.

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. DSOM supports dynamic request invocation on both local objects and objects outside the address space of the request initiator, via proxies. The (non-CORBA) **somDispatch** method can also be used to invoke methods dynamically on either local or remote objects.

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 shown here:

```
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
an **Identifier** string as defined in the CORBA specification

argument
an **any** structure with the following declaration:

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

_type
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 9, The Interface Repository Framework** on page 337 for a complete explanation of **TypeCode**s.

_value
a pointer to the value of the **any** structure. 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
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	sizeof (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 number of elements

Table 17. Data Types for C Bindings

arg_modes
a bitmask (unsigned long) field 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 *Programmer's Reference for SOM and DSOM* 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 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 );
```

Although the **create_request_args** method is introduced by SOMDObject, it can be invoked on any local or remote SOMObject.

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.

The **create_request_args** method is not defined in CORBA. Hence, the method **create_operation_list**, defined on the **ORB** class, should be used instead when writing CORBA-compliant programs.

Building a Request

There are two ways to build a **Request** object. Both begin by calling the **create_request** method on the object on which the method is to be invoked. 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 prototype for **invoke** is as follows:

```
ORBStatus invoke(
    in  Flags  invoke_flags );
```

There are currently no flags defined for the **invoke** method. When the target object of the dynamic method is local, **invoke** simply dispatches the local method. When the target object is remote, **invoke** calls the ORB, which handles the method invocation and returns the result. This method will block while awaiting return of the result.

The IDL prototype for **send** is as follows:

```
ORBStatus send(
    in  Flags  invoke_flags );
```

The following flag is defined for **send**:

INV_NO_RESPONSE
Means that the caller does not want to wait for a response.

When the target object is local, the **send** method has slightly different semantics depending on whether the INV_NO_RESPONSE flag is set. If this flag is set, **send** dispatches the local method and any output arguments will be updated.

When the object is local but this flag is not set, **send** has no effect and the client must call **get_response** to dispatch the method. When called with a remote target object, 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.

If **send** is called with INV_NO_RESPONSE for a local target object, **get_response** has no effect, since the method has already been dispatched. Otherwise, **get_response** called for a local object dispatches the method and any output arguments will be updated.

For a remote target object, **get_response** determines whether a request has completed. 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

Given 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;
   Repository repo = SOM_InterfaceRepository;
   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(repo, *ev,
                  "testObject::testMethod");
   _somFree(repo);
   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);
}
```

Building a Client-Only stub DLL

When developing a DSOM client program that invokes methods on a remote object without having a local copy of the DLL for the object's class, the developer must create a local "stub" DLL for the remote object. This DLL is needed because it contains the class data structure for the object's class, and that data structure is needed in order to create the local proxy object for the remote object, and to use the static language bindings.

Instead of complete method functions, stub DLLs contain only stub method functions. Stub DLLs, unlike the full-implementation DLLs, can be generated automatically by a developer having only the IDL specification of a class. Only the server of the remote object needs to have the object's full implementation.

Client-side stub DLLs can be constructed by performing the following steps:

- Run the SOM Compiler on the (public) IDL class interface specification, using the **h** emitter, the **ih** emitter, the **c** emitter and the **imod** emitter. (Alternatively, the **xh**, **xih** and **xc** emitters can be used.) For more information, see **Chapter 6, The SOM Compiler** on page 155 or see the DSOM sample programs.
- Compile these files together to yield a client-side stub DLL, in the same way that regular class DLLs are compiled.

A stub DLL cannot be used to invoke methods on a local object. It is sufficient, however, for the creation of a local proxy for a remote object, and provides the necessary support to allow methods to be invoked on the remote object via the proxy.

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 circumstances 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 DSOM. DSOM constructs a proxy class by creating a class that inherits the interface and implementation of **SOMDClientProxy**, and inherits the interface (but not the implementation) of the target class (the class of which the remote object is an instance).

SOMDClientProxy inherits from **SOMMProxyForObject**, which is a base class for creating proxies. Every proxy contains the **sommProxyDispatch** method, inherited from **SOMMProxyForObject**. This method is used to dynamically dispatch a method on an object, and it can also be overridden with application-specific dispatching mechanisms. In **SOMDClientProxy**, the **sommProxyDispatch** method is overridden to forward method calls to the corresponding remote target object. For more information, refer to the **SOMMProxyForObject** class in the Metaclass Framework section of the *Programmer's Reference for SOM and DSOM* or see **Chapter 10, The Metaclass Framework** on page 357.

Almost all methods invoked on a default proxy are simply forwarded and invoked on the remote object. This is true for all methods introduced by the target class. However, some methods introduced by **SOMDClientProxy** (or an ancestor class) have special behavior and actually forward other methods to the remote object. A number of methods are not forwarded to the remote object because their definition makes more sense in the local context. For a list of methods in each category, see **Making Remote Method Calls** on page 250.

Shown next 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`.

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 the **SOMDClientProxy** and the target class (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";
     _get_attribute_long: override;
     _set_attribute_long: override;
  };
};
```

Normally one would use the **abstractparents** IDL modifier to indicate that abstract inheritance should be used for the target class of the user-defined proxy class, for example, `Foo`. In this example, however, abstract inheritance is not desired, because the proxy class should inherit the instance data of the `Foo` class to permit the caching of instance data.

When you build a user-supplied proxy, you only need to override methods introduced by the target interface which you do *not* want forwarded or that require special processing. In the implementation section of the `fooproxy.idl` file, methods `_set_attribute_long` and `_get_attribute_long` are overridden to prevent the methods from being forwarded.

```
/* fooproxy.c */
#include <fooproxy.ih>

SOM_Scope long SOMLINK _get_attribute_long(Foo__Proxy somSelf,
                                           Environment *ev)
{
    Foo_ProxyData *somThis = Foo__ProxyGetData(somSelf);
    Foo__ProxyMethodDebug("Foo__Proxy","_get_attribute_long");

    return Foo__Proxy_parent_Foo__get_attribute_long(somSelf, ev);
}
SOM_Scope void SOMLINK _set_attribute_long(Foo__Proxy somSelf,
```

```
                                         Environment *ev,
                                         long attribute_long)
{
    Foo_ProxyData *somThis = Foo__ProxyGetData(somSelf);
    Foo__ProxyMethodDebug("Foo__Proxy","_set_attribute_long");
    Foo__Proxy_parent_Foo__set_attribute_long(somSelf, ev,
                                              attribute_long);
}
```

If you need to override a method to perform special, local processing, but still want to invoke a method on the remote object, you will need to explicitly call **sommProxyDispatch**.

```
SOM_Scope string SOMLINK method1(Foo__Proxy somSelf,
                                 Environment *ev,
                                 string* a,
                                 long* b,
                                 ReferenceData* c)
{
    string methodName = "method1";
    somId temp_id = &methodName;
    SOMMProxyForObject_sommProxyDispatchInfo dispatchInfo;
    string ret_str;
    Foo_ProxyData *somThis = Foo__ProxyGetData(somSelf);
    Foo__ProxyMethodDebug("Foo__Proxy","method1");

    /* perform special processing here */
    /* redispatch method, remotely */
    _somGetMethodData(_Foo, temp_id, &dispatchInfo.md);
    _sommProxyDispatch(somSelf, (void **)&ret_str,
                       &dispatchInfo,
                       somSelf, ev, a, b, c);
    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.

  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
  ```

  In the implementation section of the **.idl** file, override all methods that you do not want forwarded and methods that require special processing. Be sure to include a **dllname** modifier and an **abstractparents** modifier in the **implementation** section if abstract inheritance is desired for the target class.

- Compile the **.idl** file. Be sure the Interface Repository gets updated with the .idl file. Complete any overridden methods. 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 PATH for Windows NT. Before creating the default proxy, the DSOM run time checks for a DLL containing the class named “*targetClassName*__**Proxy**”, as indicated by the SOM IDL modifier **dllname**. 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 **sommProxyDispatch** 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 **sommProxyDispatch** 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 preceding example, subclasses of `XYZ` do not inherit the value of the **baseproxyclass** modifier. If needed, the SOM IDL modifier **baseproxyclass** must be specified explicitly in each class.
- The IDL files containing the **baseproxyclass** modifier must be compiled into the Interface Repository so that the modifier will be accessible to DSOM at run time.

Error Reporting and Troubleshooting Hints

This section describes the format of DSOM error messages and provides numerous troubleshooting hints that can help you verify the DSOM setup and analyze problem conditions.

Error Reporting

When the DSOM run-time environment encounters an error during execution of a method or procedure, a SYSTEM_EXCEPTION is raised. The standard system exceptions are discussed in **Exceptions and Error Handling** on page 100. The minor field of the returned exception value contains a DSOM error code.

Although a returned exception value can indicate that a DSOM run-time error occurred, it may be difficult to determine what caused the error. Consequently, DSOM has been enabled to report run-time error information.

DSOM Error Codes

The error codes that may be encountered when using DSOM are listed in **Error Codes** on page 397 which contains the codes for the entire SOMobjects Developer Toolkit.

Fatal Errors

In DSOM 2.x, certain categories of errors were deemed to be *fatal*. If a fatal error occurred, the DSOM process would be immediately terminated by the DSOM run-time. DSOM 3.0 no longer raises fatal error; instead, all error conditions are reported to the caller and the process is never terminated by DSOM. As a result of this change, DSOM applications must ensure that they properly check for exceptions in the **Environment** structure following every DSOM operation to prevent subsequent problems. DSOM applications should no longer rely on DSOM to terminate the process when an error occurs.

Note: When the DSOM daemon, **somdd**, is terminated, all servers are also terminated.

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 ensure that the DSOM environment is set up correctly.

- If you are using DSOM for cross-machine communication, ensure that your networking hardware and software are properly installed.The hosts file must contain an entry for both the client and server machines. The **ETC** environment variable must be set on OS/2 for all supported protocols.

Verify the networking setup for your particular networking product as follows:

- For AnyNet:
  On OS/2, There should not be an `%ETC%\resolv` file, unless TCP/IP is running and it is configured to resolve host names with a domain name server. Host name resolution can take several seconds when there is a resolve file present and the domain name server it references is not running or is not accessible. Make sure there are no errors in running **sxstart.cmd** by opening the task window for **sx.exe**. Make sure the basic networking is operating properly by using AnyNet's **ping** command to ping the other machine (client or server).
- For TCP/IP:
  Ensure that TCP/IP is loaded and operational.

For more information on your network's hardware/software setup, see

- The documentation provided with your networking software

The **SOMDPROTOCOLS** setting of both client and server contain a common entry, and that common protocol supports cross-machine communication (the **SOMD_IPC** protocol does not).

- Use the **somdchk** tool to verify DSOM environment configuration (particularly the **SOMIR**, **SOMENV** and **SOMDDIR** settings). If **SOMIR** 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.) All files indicated by **SOMIR** should exist and be readable. The directory indicated by **SOMDDIR** should exist and be readable and writable.
- For all application class libraries to be loaded dynamically, IDL containing dllname modifiers must be compiled into the Interface Repository designated by **SOMIR**. (This includes any application-specific subclasses of **SOMDServer** and **SOMDClientProxy**.) You can verify that a class exists in the Interface Repository by executing **irdump** *className*. See **Registering Class Interfaces** on page 30 for more information.
- An implementation must be registered with DSOM by running the **regimpl** utility. See **Registering Servers and Classes** on page 31 for more information. The server must be registered on the machine on which it will run. (A server cannot be registered from a different machine.) The **SOMDDIR**, **SOMDPROTOCOLS**, **HOSTNAME** and **SOMDPORT** settings in effect at the time the server was registered must be the same as those in effect when the server (and its associated daemon) are executed. Before a server can be registered the Naming Service and Security Service must be configured, using the **som_cfg** tool, and the daemon must be running. For workgroup configurations, **somdd** must be running on the machine on which the global root of the Naming Service resides and the machine on which the Security Service resides.
- Verify that all class libraries and networking libraries are in directories specified in LIBPATH. Both the client and server machines need a DLL for each class. For machines that will run only client programs (and not servers), a *stub* DLL may be used instead. See **Building a Client-Only stub DLL** on page 318 for more information.
- Ensure that memory-management IDL modifiers correctly reflect the memory-management policy executed by the application, and that they have been correctly expressed in IDL. (The SOM Compiler currently does not check the correctness of these modifiers.) See **Memory Allocation and Ownership** for more information.)
- Ensure that the contents of the **SOMDDIR** directory and the **SOMNMOBJREF** setting are consistent with one another (that is, they were all produced from the same run of the **som_cfg** tool). Further, the **SOMDPROTOCOLS**, **HOSTNAME** and **SOMDPORT**

settings of the name server and its associated daemon should match those in effect when **som_cfg** was run.

Analyzing Problem Symptoms

Check any DSOM error codes returned. If none of the error codes apply to your situation, check the following suggestions for possible solutions.

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. Use the **imod** emitter to build the DLL; it takes care of the details.

A SOMDERROR_ClassNotFound error is returned by a server when creating a remote object. 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 if the DLL for the class does not provide a **SOMInitModule** that initializes all the classes it contains.

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 in the library's **SOMInitModule** function.

A DLL built with the **imod** emitter provides a proper **SOMInitModule**.

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.

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 should not assume ownership of memory passed to it in a method parameter unless the IDL description of the method includes the SOM IDL modifier **object_owns_parameters**. Otherwise, if a parameter value is meant to persist between method invocations, then the object is responsible for making a copy of the parameter value.

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 pointers (pointer types are a SOM extension), by dereferencing the pointer in the call, and copying the base value. The value and the related pointer are 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]`).

A segmentation violation occurs when passing an any argument to a method call, where the any value is a string, array, or object reference. The NamedValues 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. To have DSOM transmit an any whose *_value* field is NULL, set the *_type* field of the any to tk_null.)

When a server program or a server object makes a call to get_id or to get_SOM_object on a SOMDObject, a BAD_OPERATION 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.

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, such as **SOMD_ImplRepObject** or **SOMD_ORBObject**.

Unexplained Program Crashes

Verify that the DSOM environment is configured properly, as described in **DSOM Configuration** on page 18. Verify that all class libraries are in directories specified in LIBPATH for AIX and OS/2. 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 on all participating server machines.

A trap is occurring when a method is invoked on a proxy, and the SMNOTC environment variable is set. The **SMNOTC** environment variable should never be set when using DSOM. Installation of the OS/2 Developer Toolkit may update your **config.sys** to set `SMNOTC=1`. This setting should be removed from config.sys. If the SMNOTC environment variable was set when the class's **.ih** or .**xih** file was generated, you will need to regenerate that file.

Problems occur after changing the setting of SOMDPROTOCOLS, HOSTNAME, or SOMDPORT. After a change to SOMDPROTOCOLS, or to the HOSTNAME or SOMDPORT setting within a protocol stanza, the Naming Service must be reconfigured (using **som_cfg**), and all application servers must be re-registered using **regimpl**.

The somossvr server program terminates immediately after starting. This will occur if the **somossvr** server program has not been initialized the first time it is run. Unlike the generic **somdsvr** server program, the **somossvr** server program must be run manually at least once before it can be started on demand by the DSOM daemon. For this initial run, it must be given the **-i** command-line argument.

A server program using a SOMobjects object services terminates. This will occur if the server program was not registered (using **regimp**) to use both the **somossvr** executable and the **somOS::Server** server class. This is required by all the SOMobjects object services.

A method can be invoked successfully on a server the first time, but subsequent invocations return garbage data to the caller. This occurs if the target object's implementation does not adhere to the caller-owned policy for parameter memory management, but the object's IDL does not contain the proper memory-management modifiers (for example, **object_owns_result**, **object_owns_parameters**). As a result, the

DSOM runtime in the server process is freeing memory that the target object still holds a pointer to. See **Memory Allocation and Ownership** on page 252.

The server traps just after an application method (invoked remotely) returns. This can occur if the application method being invoked in the server does not properly initialize the inout/out parameters and the return value data structures according their declared IDL types. As part of marshalling results back to the caller, DSOM traverses all returned data structures according to their declared IDL types, copying the data therein to an interprocess message. If any values in those data structures are invalid (for example, if one contains an invalid pointer), this will cause DSOM to trap.

A client application traps just after invoking a method remotely, and the remote method is never executed in the server. This can occur if the client does not properly initialize the in and inout parameters according to their declared IDL traverses all in/inout parameters according to their declared IDL types, copying the data therein to an interprocess message. If any values in those data structures are invalid (for example, if one contains an invalid pointer), this will cause DSOM to trap.

A client application traps while making a remote method call, just after the remote method has been executed in the server, but before control returns to the client application. This can occur if the client does not provide storage for out parameters and return values according to their declared IDL types. The client is responsible for allocating (but not necessarily initializing) the top-level storage for all out and return values. If this storage is not allocated, then DSOM may trap when attempting to store the out/return values from the remote call in the storage it thinks has been provided. See **Memory Allocation and Ownership** on page 252.

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 1.1).

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, that include the:

- Interface Definition Language (IDL) for defining classes (discussed in **Chapter 5, SOM Interface Definition Language** on page 115)
- C usage bindings (procedure-call formats) for invoking methods on remote objects
- Dynamic Invocation Interface and an Interface Repository, that support the construction of requests at run time
- Object Request Broker run-time programming interfaces

SOM and DSOM were developed to comply with these specifications along 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 does not need to be aware of SOM as the implementation technology for the ORB.

This section assumes familiarity with CORBA 1.1. The specification is published jointly by the OMG and X/Open'. The mapping of some CORBA terms and concepts to DSOM terms and concepts is described in the remainder of this section.

Object Request Broker Run-Time Interfaces

CORBA 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 and are implemented in DSOM. They are introduced briefly here, and discussed in more detail throughout this chapter. See *Programmer's Reference for SOM and DSOM* for the complete interface definitions.

Object
: The Object interface defines operations on an object reference: the information needed to specify an object within the ORB.

  In DSOM, the **SOMDObject** class 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
: 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 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 in **Chapter 9, The Interface Repository Framework** on page 337.

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 and a list of input and output arguments. A Request can be invoked synchronously, asynchronously or as a oneway call. See **Dynamic Invocation Interface** on page 311 for more information on Request objects.

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 and free the value's memory and so on.

Context

A Context object contains a list of *properties* that represent information about an application process environment. Each property consists of a *name,string_value* pair, and is used by application programs or methods much like the environment variables found in operating systems like AIX and OS/2. 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 user, the *principal*, on whose behalf a request is being performed. CORBA 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

An Object Adapter (OA) provides the primary interface between an implementation and the ORB core. An ORB may have a number of OAs, with interfaces that are appropriate for specific kinds of objects. The Basic Object Adapter (BOA) is intended to be a general-purpose OA available on all CORBA-compliant ORBs. The BOA interface provides support for the 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 SOM Object Adapter (SOMOA) class, derived from BOA, is DSOM's primary OA implementation. The SOMOA interface extends the BOA interface with several methods not defined by CORBA.

Object References and Proxy Objects

CORBA defines the notion of object references. An object reference is the information that specifies an object in the ORB. An object is defined by the **ImplementationDef** of its server, its **InterfaceDef** and application-specific **ReferenceData** 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 ReferenceData to the OA. The OA returns the reference. In DSOM, the creation of object references is typically done within the **SOMDServer::somdRefFromSOMObj Method.**

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 be used to identify the object on that server. The DSOM class that implements simple object references is called *SOMDObject*. However, in a client's address space, DSOM represents the remote object with a proxy object. When an object reference is passed from server to client, DSOM dynamically 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. To create a proxy object, DSOM builds a proxy class using the SOM facilities for building classes at run-time. The proxy class is constructed using multiple inheritance. The functionality of the proxy object is inherited from SOMDClientProxy, and only the interface of the target class is inherited. See **Figure 16**.

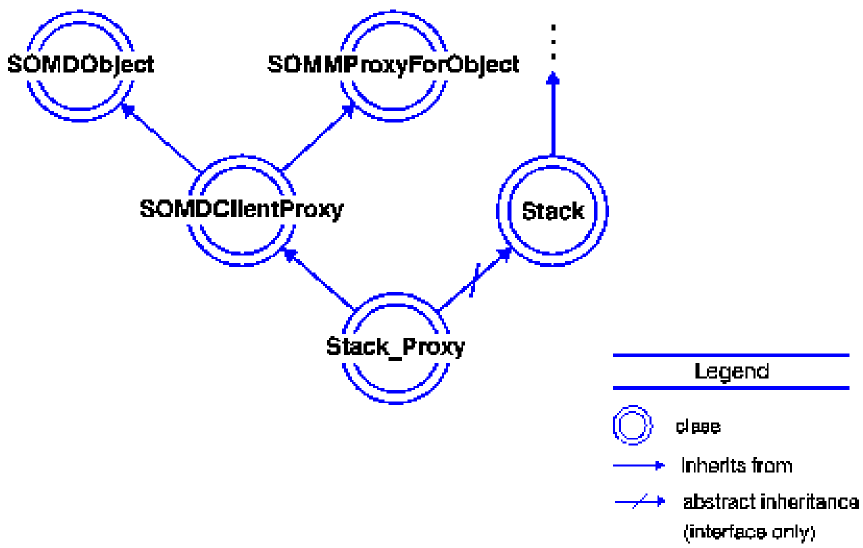

Figure 16. Construction of a Proxy Class

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 on the remote object using the same language bindings.

Recall the `Stack` class used in the **DSOM Tutorial** example. When a server returns to the client a reference to a remote `Stack`, DSOM builds a `Stack__Proxy` class (note the 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 themselves rather than on the remote objects. These operations are defined by the SOMOA, SOMDObject and ORB classes. Examples of these operations are listed below, expressed in terms of their IDL definitions.

SOMOA methods

Create and return an object reference.

```
sequence <octet,1024> ReferenceData;
SOMDObject  create (in ReferenceData id, in InterfaceDef intf,
                    in ImplementationDef impl);
```

SOMDObject methods

Create and return a duplicate object reference.

```
SOMObject  duplicate ( );
```

Destroy an object reference.

```
void  release ( );
```

Test to see if the object reference is NULL.

```
boolean  is_nil ( );
```

ORB methods

Convert an object reference to a (storable) string form.

```
string  object_to_string ( in SOMObject obj );
```

Convert a string form back to the original object reference.

```
SOMObject  string_to_object ( in string str );
```

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, Converting OIDL Files to IDL** on page 417.)

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 **Exceptions and Error Handling** on page 100 for a description of how to **get** and **set** error information in the **Environment** structure.

Dynamic Invocation Interface

The CORBA specification defines a Dynamic Invocation Interface (DII) that can be used to dynamically build requests on remote objects. In DSOM, method invocations on proxy objects are forwarded to the remote target object. SOMobjects applications can use the SOM method **somDispatch** for dynamic method calls on local or remote objects. The DSOM implementation of the DII is described in **Dynamic Invocation Interface** on page 311.

Implementations

The CORBA specification defines *implementation* as the code that implements an object. The implementation usually consists of a program and class libraries.

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, program name, location and so forth. The objects are stored in an Implementation Repository. The Implementation Repository is represented by an **ImplRepository** object.

A CORBA-compliant ORB must provide the mechanisms for a server program to register itself with the ORB. Self registration with an ORB tells enough information about the server process so the ORB will be able to locate, activate, deactivate and dispatch methods to the server process. DSOM supports these mechanisms, so server programs written in arbitrary languages can be used with DSOM. See **Object Adapters** on page 332 for additional information.

Besides the generic registration mechanisms provided by all CORBA-compliant ORBs, DSOM provides support for SOM-class libraries. DSOM provides a generic server program that registers itself with DSOM, loads SOM-class libraries on demand, and dispatches incoming requests on SOM objects. By using the generic server program, a user may be able to avoid writing any server program code.

Servers

Servers are processes that execute object implementations. CORBA defines four activation policies for server implementations as follows.

- A shared server implements multiple objects at the same time and allows multiple methods to be invoked simultaneously.
- An unshared server 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.
- A persistent server is a shared server that is activated “by hand” instead of being activated automatically when the first method is dispatched to it.

The term “persistent server” refers to the relative lifetime of the server. CORBA implies that persistent servers are started at ORB boot time. However, it should not be assumed that a persistent server implements persistent objects that persist between ORB reboots.

Note: The current release of DSOM supports a simple server activation policy 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.

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 required. For example:

- An object that requires a server-per-method implementation could 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. This may be done dynamically, see **Programmatic Interface to the Implementation Repository** on page 38.

Object Adapters

An Object Adapter (OA) provides the mechanisms a server process uses to interact with DSOM. An Object Adapter is responsible for:

- server activation and deactivation
- dispatching methods

- activation and deactivation of individual objects
- providing the interface for authentication of the principal making a call

DSOM defines a BOA interface as an abstract class. The BOA interface represents generic OA 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 SOMOA that uses the SOM compiler and runtime support to accomplish dispatching of methods. 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 customize them without having to build object adapter subclasses.

SOMOA introduces two methods to 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 where if no messages are pending, the method call will return instead of wait. The generic server program provided by DSOM uses **execute_request_loop** to receive and execute requests on SOM objects.

If the server implementation has been registered as *multi-threaded* by using the IMPLDEF_MULTI_THREAD flag in the **ImplementationDef**, SOMOA automatically runs 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.

ORB-to-ORB Interoperability

Interoperability between CORBA implementations is first addressed by OMG in the CORBA 2.0 specification. CORBA 2.0 interoperability is based on ORB-to-ORB communication. The objective is to enable objects in one ORB to invoke methods on objects in a different ORB implementation transparently. That is the client-object need not do any thing special to invoke a method on the object in the different ORB. The ORB implementations are responsible for forwarding method requests to the other ORB if the object is resident in the other ORB. This kind of support has to be provided independent of, and inspite of, differences such as platform, protocol, format, and so forth that might exist between the two ORBs. This is possible if the two ORBs have a standardized way of invoking method and receiving responses, that enables them to translate the messages to data that can be interpreted by each ORB.

CORBA 2.0 interoperability specifies the standardized message protocol and formats for ORB-to-ORB communication. There are two broad specifications of interoperability in CORBA 2.0. The general interoperability protocol is called General Inter-ORB Protocol GIOP). This specifies the Common Data Representation and seven GIOP message formats. The GIOP protocol can be implemented on any reliable transport. The mandatory transport is TCP/IP. The GIOP implementation using TCP/IP is called Internet Inter-ORB Protocol or IIOP. It is mandatory for all interoperating ORBs to support IIOP.

SOMobjects 3.0 supports IIOP. That is, a SOMobjects client can invoke methods on objects in any ORB that support IIOP, and also a client in any ORB can invoke methods on objects in a SOMobjects server. SOMobjects does not require any special compiler options for IIOP.

The other part of the interoperability specification is the Environment-Specific Inter-ORB Protocol or ESIOP. Support for ESIOP is not mandatory. OMG has defined one ESIOP protocol based on DCE. This is officially designated by OMG as DCE-Common Inter-ORB Protocol (DCE-CIOP). SOMobjects 3.0 does not provide support for DCE-CIOP.

Besides the Inter-ORB protocols, the object reference has to be standardized so that it can be interpreted by any ORB. The standardized object reference format is called the Interoperable Object Reference (IOR). The IOR specification adopted by OMG in the CORBA 2.0 standard has a type ID, which is the same as in the Interface Repository for that interface. Additionally the IOR has one or more tagged profiles, one each for a protocol that the object (or more correctly the server containing the object) supports. Only part of the profile information such as the protocol and the hostname and (listening) port can be deciphered from the profile. The rest of the profile information (the object key) has some server-specific information and can only be interpreted by the server.

Note: While SOMobjects 3.0 supports IIOP 1.0, which is defined in the CORBA 2.0 specification, SOMobjects 3.0 does not claim full support for CORBA 2.0. New features defined in CORBA 1.2 and 2.0 are not generally supported by SOMobjects 3.0.

DSOM Limitations

DSOM implementation has the following limitations in implementing CORBA specification:

- 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 **ImplRepository.** 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.
- DSOM supports a simple server activation policy, 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 Extensions

DSOM implementation extends its implementation of the CORBA specification in the following ways:

- 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 passing objects by copy (C++ semantics) or by value.
- DSOM allows non-standard types to be expressed in IDL and marshalled using DSOM. For example, pointers and SOMFOREIGN types are supported. (SOMFOREIGN types require a user-supplied marshalling function or method.)
- DSOM allows different dispositions for parameter memory in addition to the standard caller owned.

Deprecated DSOM Objects and Methods

Although the current release of DSOM generally provides backward compatibility for all objects and methods supported in DSOM 2.x or before, the programming model has evolved to incorporate new standards and provide greater flexibility and extensibility.

- The **ObjectMgr** and **SOMDObjectMgr** interfaces are deprecated. In DSOM 2.x, methods **somdNewObject**, **somdFindServer**, **somdFindServerByName**, **somdFindServersByClass** and **somdFindAnyServerByClass** were used as part of object creation. For information on creating objects in the current release of DSOM, see **Finding a SOM Object Factory** on page 245 and **Creating an Object from a Factory** on page 247.

  Method **somdReleaseObject** was used to destroy a proxy object (but not the target object). Method **SOMDObject::release** or **SOMDClientProxy::somdProxyFree** can be used for this purpose.

  Method **somdDestroyObject** was used to destroy both the proxy object and the target object. Method **SOMObject::somFree** can be used for this purpose. Note that the default behavior of **somFree**, when invoked on a proxy object, is different in the current release of DSOM. In 2.x, **SOMDObjectMgr** attribute **somd21somFree** needed to be explicitly set for **somFree** to destroy both the proxy object and target object. Destroying both objects is the default behavior in the current release of DSOM.

  Methods **somdGetIdFromObject** and **somdGetObjectFromId** were used to convert between a proxy object and its externalized, string form. **ORB** methods **object_to_string** and **string_to_object** can be used for this purpose.
- The **SOMDObject** methods **is_SOM_ref** and **is_constant** should only be called from a server. If these methods are invoked on a client proxy object, an exception will be raised.
- The **SOMDServer** methods **somdCreateObj**, **somdDeleteObj** and **somdGetClassObj** are deprecated.

  In DSOM 2.x, methods **somdCreateObj** and **somdGetClassObj** were used as part of object creation. For information on creating objects in the current release of DSOM, see **Finding a SOM Object Factory** on page 245 and **Creating an Object from a Factory** on page 247.

  Method **somdDeleteObj** was used to delete a remote, target object. To accomplish this same purpose now, you should use the **SOMDClientProxy::somdTargetFree** method.
- The **SOMDServerMgr** methods **somdDisableServer**, **somdEnableMethod** and **somdIsServerEnabled** methods are deprecated. These methods can still be called, but the **somdDisableServer** and **somdEnableServer** methods will have no effect, and the **somdIsServerEnabled** method will always return TRUE.

- The **SOMOA::change_id** and the **SOMDObject::is_constant** methods are deprecated, as is the use of the Object Reference Table file for persistent storage of ReferenceData associated with object references exported by a server. Unless a server's **ImplementationDef** object (from the Implementation Repository) specifies an object reference table filename, SOMOA's implementation of the **create** method simply invokes the **SOMOA::create_constant** method. Eventually, the non-CORBA-compliant **SOMOA::create_constant** will be eliminated in favor of the CORBA-compliant **create** method. Servers that need persistent storage of object-reference data, such as that previously provided by the Object Reference Table, should implement this functionality in an application-specific subclass of **SOMDServer**, or use the **somOS::Server** class for this purpose.

Chapter 9. The Interface Repository Framework

The SOM Interface Repository (IR) is a database 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 Framework is a set of classes 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. 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 **SOM Interface Definition Language** on page 116. 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 using the methods provided by the Interface Repository Framework.

This chapter describes how to build and manage interface repositories and the programming interfaces embodied in the SOM Interface Repository Framework.

Using the SOM Compiler to Build an Interface Repository

The SOMobjects Toolkit includes an Interface Repository emitter, the **ir** emitter, invoked whenever the SOM Compiler is run with the **-u** option to update the interface repository. The IR emitter can 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 and Windows NT:

```
set SOMIR=c:\myfiles\newicls.ir

sc –u newcls
```

For AIX:

```
export SOMIR=~/newcls.ir
sc –u newcls
```

Note: Ensure that no spaces separate the environment variable **SOMIR**, the equals sign (=) and the value being set.

If the **SOMIR** environment variable is not set, the Interface Repository emitter creates a file named **som.ir** in the current directory.

The SOM Compiler command runs the Interface Repository emitter plus any other emitters indicated by the environment variable **SMEMIT**. To run the Interface Repository emitter by itself, run the SOM Compiler with the **-s** option (which overrides **SMEMIT**) set to `ir`. For example:

```
sc –u –sir newcls
```

or equivalently,

```
sc –usir newcls
```

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 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.

For additional information on the **SMEMIT** and **SOMIR** environment variables, see **Environment Variables Affecting the SOM Compiler** on page 159. For additional information on the **-u** and **-s** compiler options, see **Running the SOM Compiler** on page 161.

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 Developer 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 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 IR 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. A single IR 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 With the SOMIR Environment Variable

The SOMobjects Developer 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 IR 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) IR 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 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 Developer Toolkit.

The IR Framework only permits updates in the rightmost file 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 IR files, or else petition the owner of that interface to make the change.

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 Developer 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`. (Some of the following code lines wrap.)

For OS/2 and Windows NT:

```
set BASE_IRLIST=%SOMBASE%\IR\SOM.IR;C:\IR\COMPANY.IR;C:\IR\DEPT10.I
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
```

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
```

This command will place all of the information in the `myclass1.idl` file, including the private portions, in the Interface Repository. For additional information on the **-p** and **-u** compiler options, see **Running the SOM Compiler** on page 161.

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.

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 IR 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 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. **ModuleDefs** are **Containers**, and they can hold **ConstantDefs**, **TypeDefs**, **ExceptionDefs**, **InterfaceDefs** and other **ModuleDefs**.

InterfaceDef
An instance of this class exists for each interface named in an IDL source file. (One **InterfaceDef** corresponds to one SOM class.) **InterfaceDefs** are **Containers**, and they can hold **ConstantDefs**, **TypeDefs**, **ExceptionDefs**, **AttributeDefs** and **OperationDefs**.

AttributeDef
An instance of this class exists for each attribute defined in an IDL source file. **AttributeDefs** are found only inside of (contained by) **InterfaceDefs**.

OperationDef
An instance of this class exists for each operation (method, **_set** method, and **_get** method) in an IDL source file. **OperationDefs** are **Containers** that can hold **ParameterDefs**. **OperationDefs** are found only inside of (contained by) **InterfaceDefs**.

ParameterDef
An instance of this class exists for each parameter of each operation (method) defined in an IDL source file. **ParameterDefs** are found only inside of (contained by) **OperationDefs**.

TypeDef
An instance of this class exists for each typedef, struct, union, or enum defined in an IDL source file. **TypeDefs** 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. **ConstantDefs** 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. **ExceptionDefs** 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.

For additional information on each Interface Repository Class, see **Chapter 3, Interface Repository Framework Classes** on page 311 of *Programmer's Reference for SOM and DSOM*

Methods Introduced by Interface Repository Classes

The Interface Repository classes introduce several new methods that 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 **Chapter 3, Interface Repository Framework Classes** on page 311 of *Programmer's Reference for SOM and DSOM.*

Contained Class

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 and a *value* field holding an *any* structure that points to another structure containing the IDL information for that particular element.

within
Returns a sequence designating the objects 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 objects except an **OperationDef** object.

Container Class

Some IR objects contain other objects and exhibit this behavior.

contents
Returns a sequence of pointers to the objects 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.

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

ModuleDef
Override **describe** and **within.**

InterfaceDef Class

describe_interface
Returns a description of all methods and attributes of a given interface definition object that are held in the Interface Repository. Overrides **describe** and **within**.

AttributeDef Class

AttributeDef
Overrides **describe**.

OperationDef Class

OperationDef
Overrides **describe**.

ParameterDef Class

ParameterDef
Overrides **describe**.

TypeDef Class

TypeDef
Overrides **describe**.

ConstantDef Class

ConstantDef
Overrides **describe**.

ExceptionDef Class

ExceptionDef
Overrides **describe**.

Repository Class

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

One instance of the Repository class exists for the entire SOM Interface Repository. This object does not 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**, **TypeDef**, **ExceptionDef**, **InterfaceDef** and **ModuleDef** class objects that are global in scope (that is, not contained inside of any other **Containers**).

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** or **InterfaceDef** objects). 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 **_get_somInterfaceRepository** 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 under **Invoking Methods on Objects** on page 76 for non-C/C++ programmers.
2. Use the **somResolve** or **somResolveByName Function** to obtain a pointer to the **_get_somInterfaceRepository** 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 Class** 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 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 uses the **contents** method, but also 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 *Programmer's Reference for SOM and DSOM.*

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 attributes (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 memory will be released automatically when the object that contains it is freed.

There are five methods that give out object references: **contents Method, describe_contents Method**, **lookup_id Method, lookup_name Method** and **within Method**. When finished with the object, you are expected to release the object reference by invoking the **somFree Method**. Do not release the object reference until you have either copied or finished using all of the information obtained from the object.

The **describe Method, describe_contents Method** and **describe_interface Method** return structures and sequences 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 free some of the information contained in the returned structures when you are finished.

During execution of the **describe** and **lookup** methods, sometimes intermediate objects 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, invoking the **release_cache Method** 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.

For all SOM Interface Repository framework methods that raise user exceptions, the exception parameters for those exceptions are allocated from a single block of memory and are not allocated piecemeal. Therefore, when freeing an exception such as **irOpenError**, do not treat the exception parameter data structure as an opaque entity. If the exception is received from a remote call, do not use the **somdExceptionFree** function to free it. Instead, use either the **somExceptionFree Function** or the **SOMFree Function**.

Using TypeCode Pseudo-Objects

Much of the detailed information contained in Interface Repository objects is represented in the form of TypeCodes. TypeCodes are complex data structures whose actual representation is hidden. A TypeCode is an architected way of describing in complete detail everything that is known about a particular data type 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 that classifies it, and one or more parameters that carry descriptive information appropriate for that particular category of TypeCode. 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 TypeCodes 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.)

A **tk_objref** TypeCode represents an object reference; its parameter is a repository ID that identifies its interface.

See **Table 2** on page 350 of *Programmer's Reference for SOM and DSOM* for a complete table showing the parameters of all possible TypeCodes.

TypeCodes are not actually objects in the formal sense. TypeCodes are referred to in the CORBA standard as pseudo-objects and described as opaque. This means that, in reality, TypeCodes 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 TypeCodes support the architected behavior and have additional capability as well.

Although TypeCodes 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 TypeCodes are named **TypeCode_***function-name*. Because of this 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 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.
};
```

Providing alignment Information

In addition to the parameters in the TypeCodes 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 TypeCodes 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` 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 or as unaligned, by using a compile-time option or **#pragma**.

The more important consideration is to make certain that the TypeCodes going into the Interface Repository actually reflect the alignment that your compiler provides. This way, when programs 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 the `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 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:

- The name of the type
- An implementation context string
- 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:

- Define the foreign type as a **typedef SOMFOREIGN** in the IDL source file.
- 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. For additional information on the **impctx** modifier, see **Modifier Statements** on page 133.
- Compile the IDL file using the **-u** option to place the information in the Interface Repository. For additional information on the **-u** option, see **Running the SOM Compiler** on page 161.

For example (note that the 2nd line wraps):

```
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. For additional information on **passthru** use, see **Passthru Statements** on page 147.

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 SOM Compiler command. For example, the command

```
sc -sh -mtcconsts myclass.idl
```

will cause the SOM Compiler to generate a `myclass.h` file that contains **TypeCode** constants for the types defined in `myclass.idl`. For additional information on the **tcconsts** modifier and the **-m** option, see **Running the SOM Compiler** on page 161.

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 data that was returned, while the **_type** field holds a TypeCode that describes the data.

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.

```
/* A C++ example that illustrates
 * -- use of TypeCodes
 * -- use of the "any" data type
 */
/* Here is the IDL:
#include <somobj.idl>
interface X : SOMObject {
    typedef long LongType;
    struct StructType {
        LongType a;
        };
    boolean showStructType(in StructType arg1,
                           in any arg2);
    // return TRUE if and only if arg2->_type is LongType
    // and *(arg2->_value) == arg1.a
    implementation {
        releaseorder: showStructType;
    };
};
*/
/* To build example on AIX w. somstars bindings,
 *
 *  sc -sh:ih -mtcconsts anyExample.idl
 *  cc anyExample.c -I. \
 *         -I$SOMBASE/include -L$SOMBASE/lib -lsomtk
 */

#ifndef SOM_Module_anyexample_Source
#define SOM_Module_anyexample_Source
#endif
#define X_Class_Source
#include "anyExample.ih"

/* return TRUE if and only if arg2->_type is LongType
 * and *(arg2->_value) == arg1.a
 */
SOM_Scope boolean  SOMLINK showStructType(X *somSelf,
                                   Environment *ev,
                                   X_StructType* arg1,
                                   any* arg2)
```

```
{
  /* XData *somThis = XGetData(somSelf); */
  XMethodDebug("X","showStructType");
    somPrintf("showStructType: arg1 = { a: %d }; \n", arg1->a);
  return (arg2->_type == TC_X_LongType &&
        *((X_LongType *)(arg2->_value)) == arg1->a);
}
main()
{
    X *x = XNew();
    X_LongType lt = 123;
    X_StructType st = { lt };
    any a;
    a._type = TC_X_StructType;
    a._value = &st;
    somPrintf("main: showStructType returned %d\n",
              _showStructType(x, 0, &st, &a));
    a._type = TC_X_LongType;
    a._value = &lt;
    somPrintf("main: showStructType returned %d\n",
              _showStructType(x, 0, &st, &a));
}
/* Example output:  */
showStructType: arg1 = { a: 123 };
main: showStructType returned 0
showStructType: arg1 = { a: 123 };
main: showStructType returned 1
```

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");
}
```

An **any** has no restriction on the type of data 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.

Building an Index for the Interface Repository

The **irindex** command builds indexes for one or more Interface Repository files to improve performance of the **lookup_name** method on a Repository object. An index is created in the same directory as the specified Interface Repository file. A unique index file with a **.ndx** name is created for each file specified. You can also find out the name of the index file associated with an Interface Repository file, if an index exists.

Syntax

irindex [-f] [*irFileName1*, *irFileName2* ...]

irindex creates an index for each file specified in the command line. If no Interface Repository file names are specified, indexes are built for the files listed in the SOMIR environment variable.

If you specify the **-f** option as the first option, the index names of any IR files are listed. No indexes are created. If no file names are specified with the **-f** option, the index name for each file in the SOMIR environment variable is listed. If a specified file does not have an index, **irindex** returns a message. Use the **-f** option when you are deleting or moving an Interface Repository file. You can determine the name of the index before you move or delete the file.

Successive invocation of **irindex** on an Interface Repository file does not recreate an index if it is consistent. **irindex** automatically recreates an index if it finds that a specified index is not consistent with the associated Interface Repository file.

Once you have created an index for an Interface Repository file, any updates to the Interface Repository file result in automatic updates to the index file. If you are updating an Interface Repository file often, consider not invoking **irindex** on that file until after you have completed your updates.

Return Messages

No message is returned on successful completion. If **irindex** is unsuccessful, either because of lack of access rights or because the file is in use, a message is returned indicating the file involved and the reason for the failure. The following list describes the error messages and recovery information associated with the **irindex** command:

Open failed for IR file: *filename*
Explanation: A failure occurred while opening the file either because the file does not exist or because the file is locked.
Programmer Response: Ensure that the file exists and is not locked by another process.

Read failed for IR file: *filename*
Explanation: A failure occurred while reading the file.
Programmer Response: Ensure that the file exists, that you have read permission for the file, and that the file is not exclusively locked.

Not authorized to create index for IR file: *filename*
Explanation: A failure occurred while creating an index for the file.
Programmer Response: Obtain create/write permissions for the directory containing the file and write access for updating the file:

Index creation failed for IR file: *filename*
Explanation: A failure occurred while creating the index.
Programmer Response: Delete the file listed in the associated message and try again.

Please delete file: *filename*
Explanation: This message is issued in conjunction with the preceding error code and message. A failure occurred while creating the index.
Programmer Response: Delete the partially-created index, if one exists.

Environment variable SOMIR not set.
Explanation: The **SOMIR** environment variable is not initialized.
Programmer Response: Either set the **SOMIR** environment variable or specify the file names as options on the **irindex** command line for the index you want to create.

Index is inconsistent. Please recreate index.
Explanation: The index is no longer consistent with the associated Interface Repository file.
Programmer Response: Either delete the index or recreate it by issuing **irindex** *filenam*e at the command prompt.

```
Index not present.
```

Explanation: An index for the file does not exist.
Programmer Response: To create an index for the Interface Repository file, issue **irindex** *filename* at the command prompt.

If you see the message `Index is inconsistent. Please recreate index`, you must delete the inconsistent index and rebuild it with **irindex**

Examples of IRINDEX Use

The following are examples of the **irindex** command:

1. To create an index for **som.ir** in the current directory:

```
irindex som.ir
```

2. To create indexes for files in the **SOMIR** environment variable, in the respective directories:

```
irindex
```

For example, on OS/2, if **SOMIR** is set to `d:\som\test.ir` and `e:\som.ir`, then indexes are created in d:\som for test.ir and e: for som.ir.

3. To create indexes in `d:som` for `test.ir` and in `e:` for `som.ir`:

   ```
   irindex d:\som\test.ir e:\som.ir
   ```

4. To list the name of the index for `som.ir` in the current directory, if one exists.

   ```
   irindex −f som.ir
   ```

5. To list the names of indexes for files in the **SOMIR** environment variable. A message is issued for files in the **SOMIR** environment variable that do not have indexes:

   ```
   irindex −f
   ```

6. To list the names of the index of each of the two **ir** files, if the indexes exist:

   ```
   irindex −f d:\som\test.ir e:\som.ir
   ```

Chapter 10. The Metaclass Framework

In SOM metaclasses are classes and thus are objects. **Figure 17** depicts the relationship of these sets of objects. Included are the three primitive class objects of the SOM run time: **SOMClass**, **SOMObject** and **SOMClassMgr**:

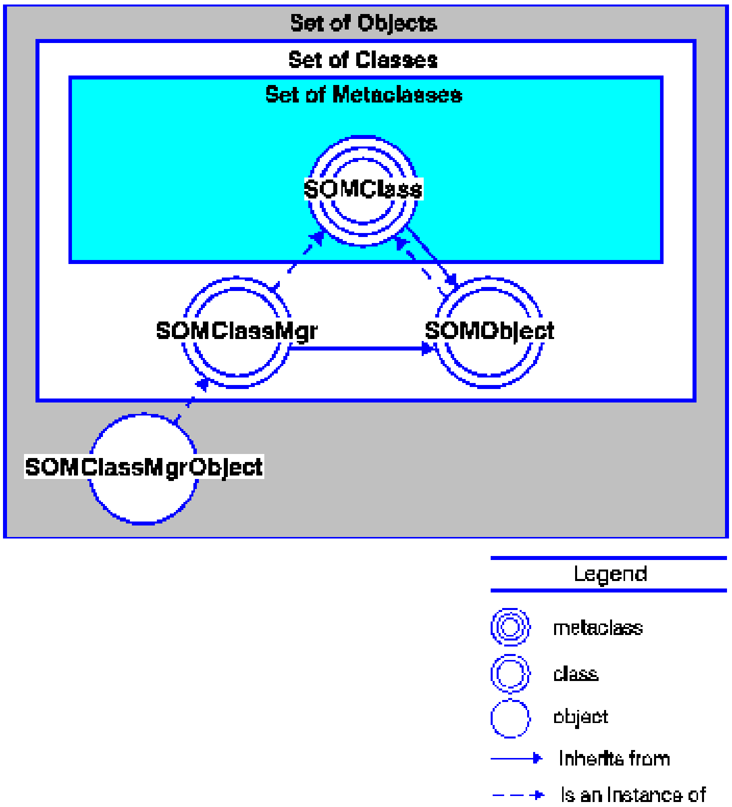

Figure 17. The primitive objects of the SOM run time.

The point to observe is any class that is a subclass of **SOMClass** is a metaclass. This chapter describes the available metaclasses in the SOMobjects Developer Toolkit. The metaclasses are:

Framework
: metaclasses for building new metaclasses

Utility
: metaclasses to help you write applications.

Framework Metaclasses

SOMMBeforeAfter Metaclass
: Used to create a metaclass that has *before* and *after* methods for *all* methods (inherited or introduced) invoked on instances of its classes.

SOMMSingleInstance Metaclass
: Used to create a class that may have at most one instance.

SOMMProxyFor Metaclass
: The metaclass for proxies.

SOMMProxyForObject Class
Serves as a helper class for making proxies. That is, you subclass this class to make a proxy (and do not use **SOMMProxyFor** directly).

Utility Metaclasses

SOMMTraced Metaclass
Provides tracing for every invocation of all methods on instances of its classes.

The diagram in **Figure 18** 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.

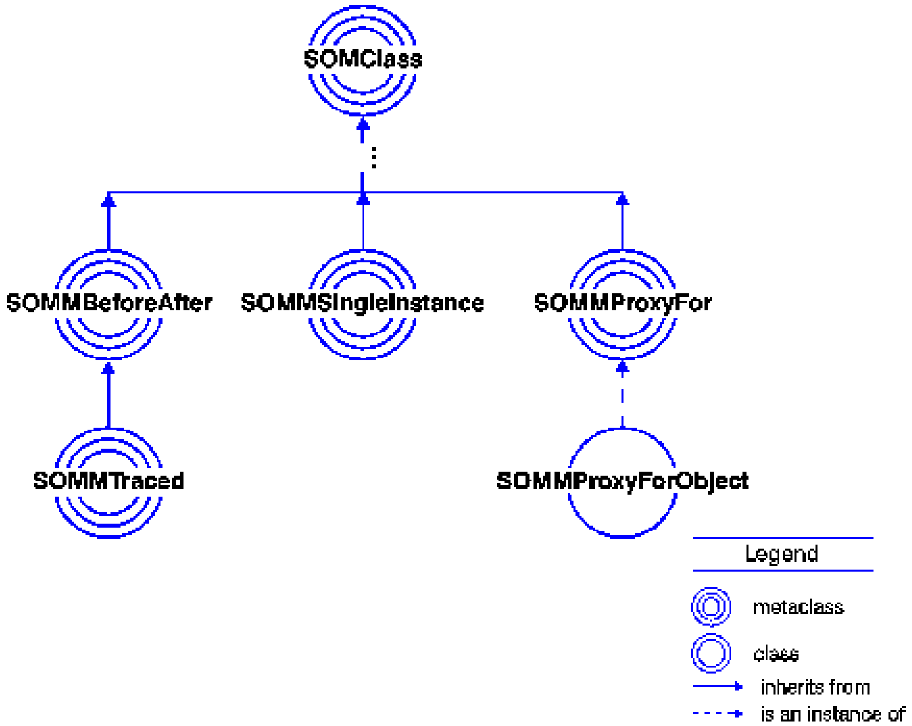

Figure 18. Class organization of the Metaclass Framework.

A Note about Metaclass Programming

SOM metaclasses are carefully constructed so that they compose. 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 Cooperation Framework used to implement the Metaclass Framework metaclasses described in this chapter.

Framework Metaclasses for before/after Behavior

The *before/after* behavior of SOM metaclasses is inherited from the **SOMMBeforeAfter Metaclass**, as described in this section.

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 the **sommBeforeMethod Method** and **sommAfterMethod Method**. 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.

In **Figure 19**, the `Barking` metaclass overrides the methods **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.

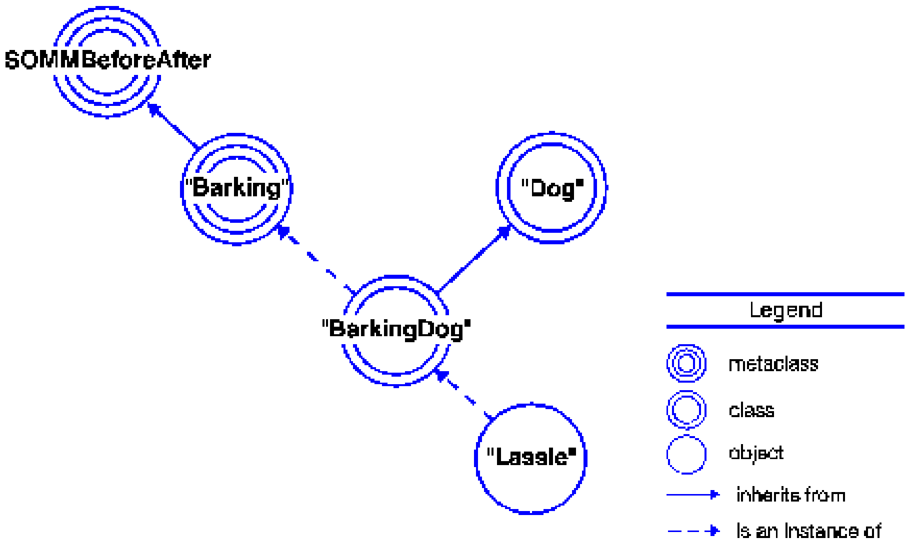

Figure 19. A hierarchy of metaclasses

The **SOMMBeforeAfter** metaclass is designed to be subclassed: a subclass (or child) of **SOMMBeforeAfter** is also a metaclass. The subclass overrides both **sommBeforeMethod** and **sommAfterMethod** or neither. These (redefined) methods are invoked before and after any method supported by instances of the subclass (these methods are called primary methods). That is, **sommBeforeMethod** and **sommAfterMethod** 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. 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. Note that CORBA specifies TRUE is 1.

Note: **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** using the object being called before **somInit** has executed, and the object is not yet fully initialized. Second, the implementation must guard against **sommAfterMethod** using the object being called after **somFree**, at which time the object no longer exists.

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" );
}
```

Since `Barking` is a subclass of **SOMMBeforeAfter**, it is not necessary to make the parent method calls in **sommBeforeMethod Method** and **sommAfterMethod Method**. Printing `Woof` does not use the object; so the guards are unnecessary. The guards are simply an example of doing so.

Composition of before/after Metaclasses

Figure 20 contains two before/after metaclasses: `Barking` and `Fierce`, which has a **sommBeforeMethod Method** and **sommAfterMethod Method** that both growl (that is, both methods 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.

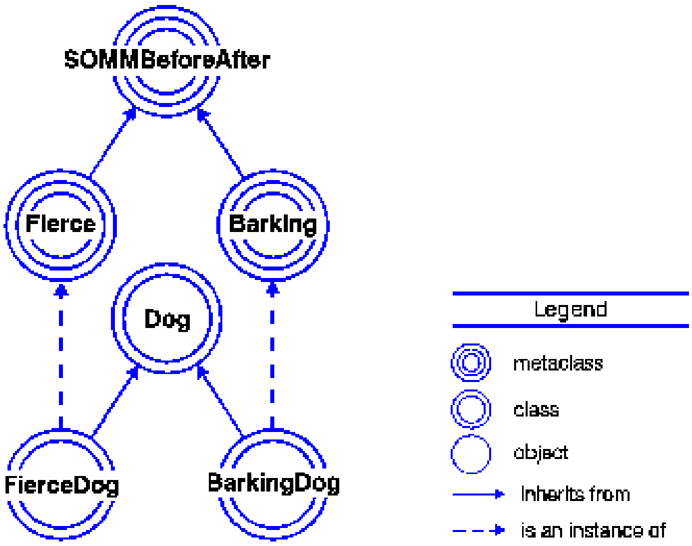

Figure 20. Example for composition of before/after metaclasses

There are several ways to express such compositions. The following list 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 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).

```
interface FB-1 : Dog
{
    ...
  implementation
   {
          metaclass =
FierceBarking;
          ...
     };
};
```

- 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`).

```
interface FB-2 : FierceDog,
             BarkingDog
{
    ...
  implementation
   {
          ...
```

```
        };
    };
```

- 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`.

```
    interface FB-3 : BarkingDog
    {
        ...
      implementation
       {
         metaclass = Fierce;
             ...
        };
    };
```

Figure 21 combines the diagrams for these techniques 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`.

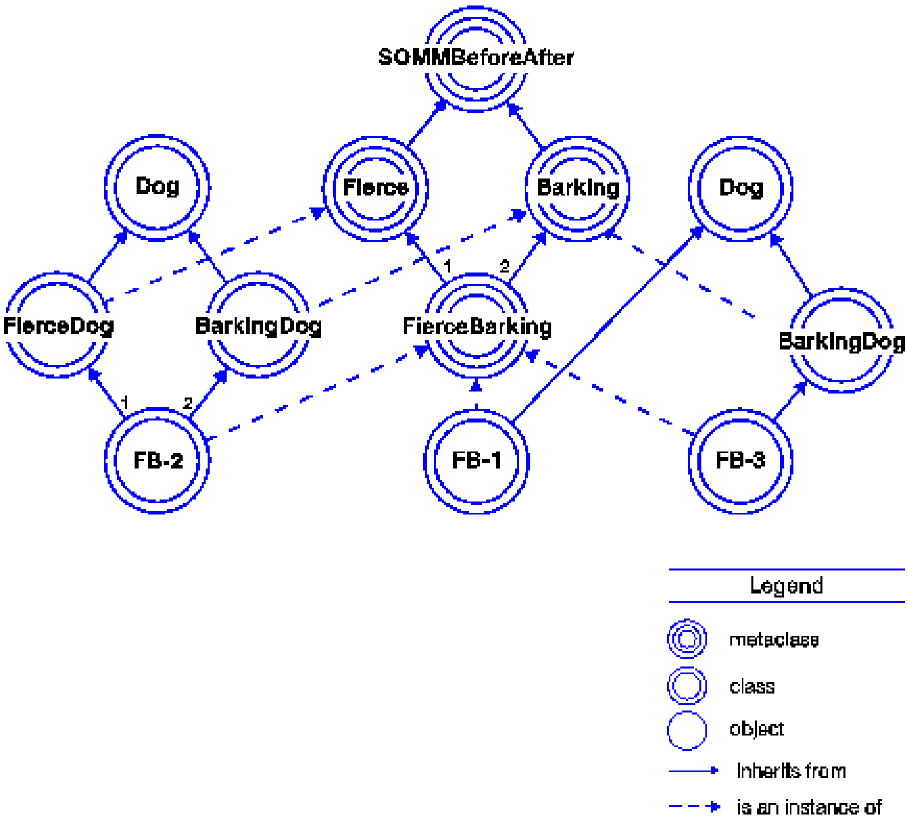

Figure 21. Relationships of the three techniques for "FierceBarkingDog"

Notes and Advantages of before/after

Notes on the dispatching of before/after methods:

- A before or after method is invoked just once per primary method invocation.
- The dispatching of before/after methods is thread-safe. That is, multiple threads can be dispatching before/after methods concurrently. (However, it is your responsibility to ensure that what those methods do 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 *n2p* 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.

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 *className***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 **sommGetSingleInstance** 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 the four methods **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 when the reference count is zero (and, 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 the **somSubstituteClass Method**. 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.)

SOMMTraced Metaclass

SOMMTraced is a metaclass that facilitates tracing of method invocations (note: select this link to jump to the reference page for the metaclass).

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 22**. 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.

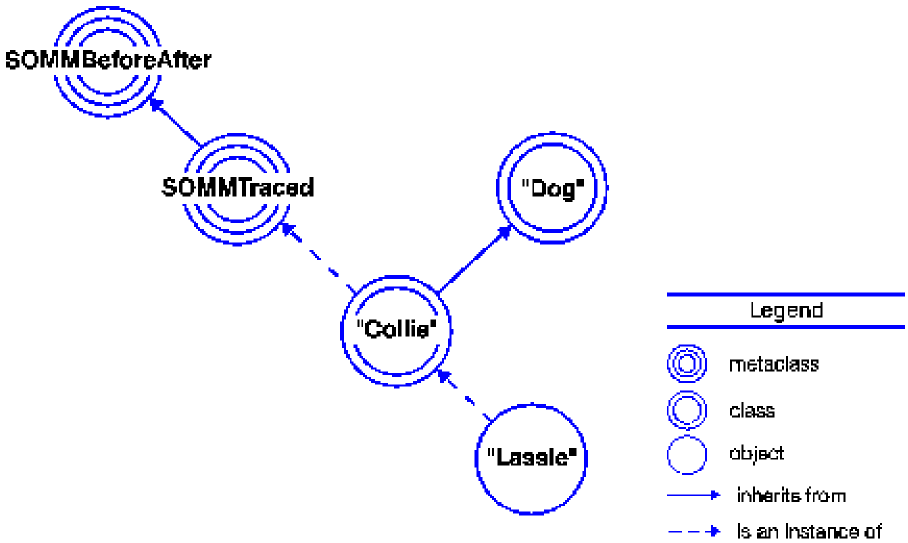

Figure 22. All methods (inherited or introduced) invoked on "Collie" are traced

It is easy to use **SOMMTraced**: Just make a class an instance of **SOMMTraced** 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. (The string that contains only two double-quotes can be used to indicate an empty string.) If

SOMM_TRACED is not the empty string, it should be set to the list of names of classes 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
```

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__ */
};
```

SOMMProxyFor Metaclass

Creating a proxy for an object is common technique in object oriented programming. The main job of a proxy is to forward method invocations to the target object. The Metaclass Framework supports this technique by allowing you to create a class for proxies. In the figure below, `ProxyForDog` is the class of any proxy for instances of `Dog`. Any method invoked on `proxyForLassie` is forwarded to `Lassie`.

Proxy classes are built appropriately by the metaclass **SOMMProxyFor**, but the metaclass **SOMMProxyFor** is never used directly. Instead, proxy classes are created by subclassing **SOMMProxyForObject**. (Note that **SOMMProxyFor** may seem like an odd name. However, in naming metaclasses, SOMobjects uses prefix phrases that impart properties to classes that impart properties to ordinary objects. Thus, the base class is named **SOMMProxyForObject**, because **SOMMProxyFor** is combined with the **SOMObject** class. This convention becomes very useful when composing proxy classes with other metaclasses.)

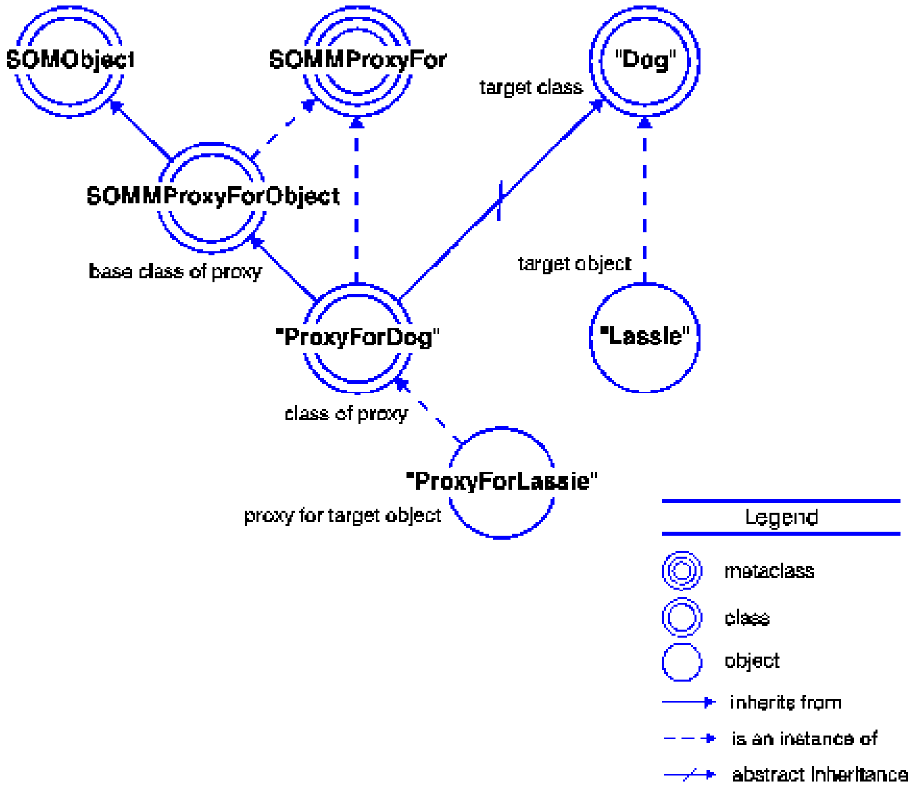

Figure 23. Example of a proxy for the "Dog" class

If a method in the class of a proxy is overridden, the method is not forwarded from the target to the proxy. If the method should be forwarded, in the override implementation a call is made to **sommProxyDispatch** to forward it. (You must be careful, however, as this interface is not quite the same as that of **somDispatch**).

Static Creation of Proxy Classes

A specialized proxy class can be defined by subclassing **SOMMProxyForObject**. In doing this, there is one primary rule that must be obeyed: A user-created proxy has at most two parents, and the first parent must be a descendant of **SOMMProxyForObject**.

Dynamic Creation of Proxy Classes

A proxy class can be created dynamically with **sommMakeProxyClass**, which is a method introduced by the class **SOMMProxyFor**. Thus, given that `P` is a pointer to a descendant of **SOMMProxyForObject** (which is a **SOMMProxyFor**), then a proxy class for the class `Dog` can be created with the command:

```
ProxyForDog = _sommMakeProxyClass( P, Dog );
```

Keep in mind that the preceding command creates the class object for proxies for the instances of class `Dog`. To create a proxy, you must create an instance of the proxy class and then set the target of the proxy with **_set_sommProxyTarget**.

Implementation Revealing Methods

A proxy forwards all methods except those that reveal implementation information. There are six such methods: **somGetClass**, **somGetClassName**, **somIsA**, **somIsInstanceOf**, **somRespondsTo** and **somGetSize**. These methods return the information associated with the proxy. For example, for **Figure 23**, the following method could be invoked:

```
_somGetClassName( proxyForLassie )
```

This method returns `ProxyForDog` rather than `Dog`, which would be the returned value if the method were forwarded from the proxy to the target.

Proxies and the Composition of Metaclasses

The metaclass constraints of a target do not propagate to a proxy. Consider the class `TracedDog`, which is a subclass of `Dog`, whose metaclass is **SOMMTraced**. When a proxy is created for this class, its name should be `ProxyForTracedDog`, which implies that the tracing occurs at the site of the target when the method is forwarded to the target.

On the other hand, suppose a subclass for **SOMMProxyForObject** and `Dog` is created with a class whose metaclass is **SOMMTraced**. In this case, the result is a `TracedProxyForDog` class, which implies that the tracing occurs at the proxy object. This may not seem like a large difference when both the proxy and the target are in the same address space, but consider the situation when the proxy is used to forward a method to another address space (as occurs with DSOM applications). With respect to distributed systems, the distinction is between actions that occur on the server versus actions that occur on the client.

Chapter 11. Emitter Framework

The SOM Compiler translates an IDL into other useful forms. An interface definition can be translated into a programming language binding file, an implementation template file, a documentation file, a description that can drive a class browser or a pretty-printed interface specification. Given the number of programming languages with which SOM can be used and the many development-support tools that can leverage object interface definitions, the SOM Compiler needs to produce a large number of output forms. Therefore, an important structural feature of the SOM Compiler is that it minimizes the effort involved in developing and maintaining new compiler back-ends.

The following figure shows how the SOM Compiler is structured to use emitters. Each emitter produces a different output file. As shown in the figure, the only part of the SOM Compiler that varies with different output targets is the emitter. A new emitter must be developed and maintained for each output target, but the IDL parser remains unchanged.

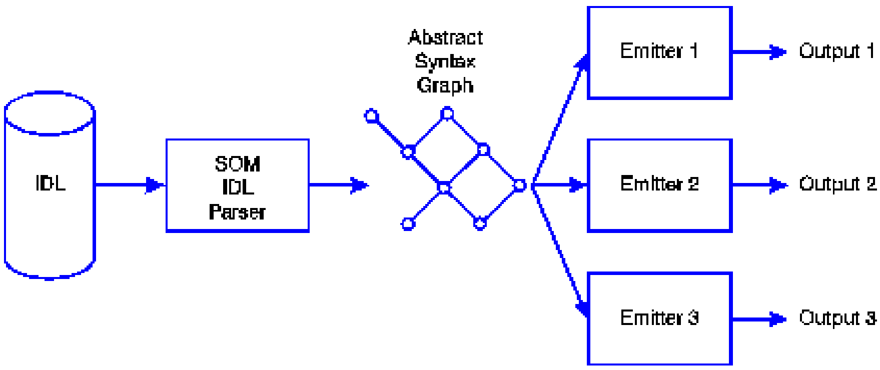

Figure 24. Structure of the SOM Compiler

To make it easier to develop new emitters for use with the SOM Compiler, the SOMobjects Toolkit provides a collection of classes called the Emitter Framework. The Emitter Framework consists of several support classes and a general emitter class that can be subclassed to produce a specific emitter. The SOMobjects Developer Toolkit provides **The newemit Facility** on page 381 for automatically generating new emitters. This automatically generated emitter is then easily customized as needed for a particular output format.

The goals of the emitter classes are to provide an object-oriented framework for emitter development that:

- Insulates new emitter code from changes to SOM's interface definition language.
- Separates design concerns, to improve the ease of development and maintenance of emitters. The designers of a new emitter should not have to understand the full emitter process. Rather, they simply override methods from one of the Emitter Framework classes. The logic of the Emitter Framework causes the methods to be invoked at the correct time.
- Supports a template facility that allows developers to specify the form of an output file in a highly readable and maintainable manner.
- Breaks up the control logic for output-file construction into small, easily maintained units.

Structure of the Emitter Framework

The Emitter Framework is structured as depicted below.

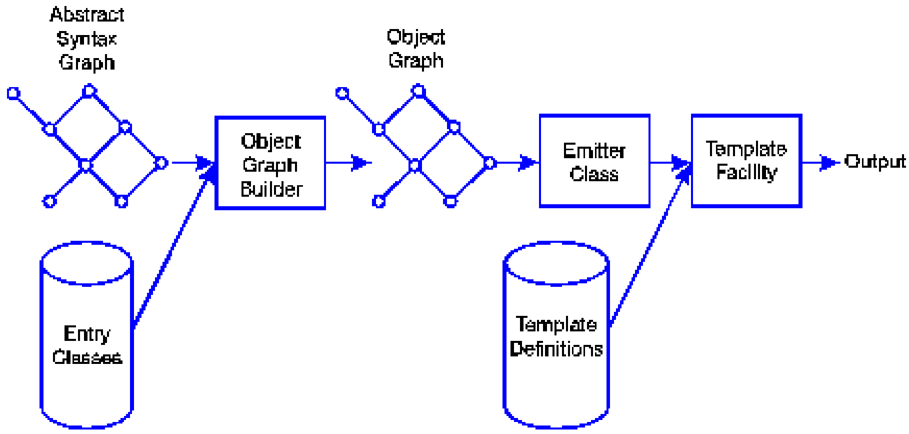

Figure 25. Structure of the SOM Emitter Framework

The Object Graph Builder

The input to the Emitter Framework is an abstract syntax graph of data structures. As shown in **Figure 24, Structure of the SOM Compiler** on page 369, the abstract syntax graph is produced by the SOM Compiler. The SOM Compiler's IDL parser reads the input **.idl** file and converts the interface descriptions that are included in it (either directly or indirectly) into the abstract syntax graph. Each node of the abstract syntax graph represents a syntactic unit of the interface definition.

Once the abstract syntax graph has been constructed by the SOM Compiler, the object graph builder, the front-end of the Emitter Framework, traverses the abstract syntax graph, building an isomorphic graph of entry objects. An *entry* object represents a syntactic unit of the interface definition. For example, a **SOMTClassEntryC** object represents an entire interface definition, **SOMTMethodEntryC** objects represent method declarations, **SOMTParameterEntryC** objects represent method parameter declarations, and so on.

Both the IDL parser and the object graph builder are closed parts of the Emitter Framework; they cannot be extended or modified by programmers using the Emitter Framework. Two important forms of flexibility are provided:

- SOM IDL syntax provides for an open-ended set of modifiers that can be associated with most syntactic elements in an interface definition. Modifiers specified in an **.idl** file are accessible to emitters written using the Emitter Framework.
- Before the object graph builder is run, an emitter can cause some or all of the Emitter Framework classes to be shadowed effectively replaced by user-defined subclasses. When the programmer shadows a particular entry class, the object graph builder uses instances of the programmer's subclass of that entry class, rather than instances of the original entry class. Thus, the programmer can modify the object graph even though the object graph builder creates all the entry class instances in code that is not open to the programmer.

The Entry Classes

The entry classes are used to construct the object graph produced by the object graph builder described above. Each node of the object graph is an instance of one of the entry classes. Each instance of an entry class represents one syntactic unit of an IDL interface definition. One piece or one entry from the complete IDL interface definition. An entry object serves two important functions:

- Holding information about the corresponding syntactic element of an IDL specification.
- Defining symbols that can be used as placeholders in an emitter's output template.

Emitter Framework Classes describes the entry classes and the use of symbols in more detail.

The Emitter Class

The emitter class, **SOMTEmitC Class**, is the class that drives the process of producing an output file. Constructing a new emitter requires creating a new subclass of **SOMTEmitC** and overriding one or more of its methods so that it produces the desired output. The new emitter is run by creating an instance of the new subclass of **SOMTEmitC** and invoking the **somtGenerateSections Method** on it.

The Template Class and Template Definitions

The template class, **SOMTTemplateOutputC Class**, is used by an emitter to produce output. It recognizes template descriptions of the output, so that most of the information about how the output file should look can be placed in a template definition and does not need to be embedded in the emitter code.

The template definitions describe the content and format of various sections of the output file, and the emitter controls which of these sections are output and in what order. The emitter calls on an instance of the **SOMTTemplateOutputC** class to have a particular section produced from that section's template definition.

The following section describes the components of the Emitter Framework and shows the recommended procedure for producing a new emitter.

Emitter Framework Classes

The Emitter Framework consists almost entirely of classes.

- The emitter class, **SOMTEmitC Class**, manages the overall activity of an emitter, obtaining information from the entry objects and directing the template object to produce specific sections.
- The template class, **SOMTTemplateOutputC Class,** manages the output of specific sections to the target file. It provides a template facility to make the specification of the output file simple.
- Most of the classes are the entry classes, **SOMTEntryC Class** and its subclasses, each of which represents some syntactic unit of an IDL definition.

The Emitter Framework classes you will explicitly instantiate (create instances of) are your own subclasses of **SOMTEmitC**. The remaining classes are instantiated automatically by the Emitter Framework.

The remainder of this section describes each of these classes. For more information, the Reference portion of this book provides detailed information on all of the attributes and methods supported by each class in the Emitter Framework.

SOMTEmitC

SOMTEmitC is the primary class of the Emitter Framework. It provides overall control for the emitting process. An emitter writer will always need to subclass this class or one of its subclasses, override some of its methods, and perhaps add a few new methods. The next section describes this process in more detail.

An instance of **SOMTEmitC** (an emitter) has as attributes a target file, a target class or target module, a template object, and a name, as follows:

- The target file is the file to which output will be directed.
- The target class (the class that information will be emitted) is represented by a **SOMTClassEntryC** object. This object is the root of the object graph built by the object graph builder when the emitter is invoked on a class definition. See **Figure 25, Structure of the SOM Emitter Framework** on page 370.
- The target module, the module that information will be emitted, is represented by a **SOMTModuleEntryC** object. This object is the root of the object graph built by the object graph builder when the emitter is invoked on a module definition. See **Figure 25, Structure of the SOM Emitter Framework** on page 370.
- The template object of the emitter, an instance of **SOMTTemplateOutputC Class**, maintains the symbol table and controls the format and content of the sections the emitter produces. The emitter itself controls which sections are emitted and their order. The template object is initialized from the output template file.
- The emitter name is the name by which the emitter is invoked via the **-s** option of the **sc** command. The emitter name is used to determine which passthrus in the input **.idl** file are directed to that emitter.

The **SOMTEmitC Class** provides methods for:

- Opening the output template file (**somtOpenSymbolsFile Method**).
- Getting the value of global modifiers those specified via the **-m** option of the **sc** command).
- Setting standard symbols associated with the target class and its metaclass, **somtFileSymbols Method**, as well as setting standard section-name symbols, **somtSetPredefinedSymbols Method**).
- Generating the output file from the output template, **somtGenerateSections Method**. **somtGenerateSections** is the primary method that a new emitter will override from **SOMTEmitC**. This method controls which sections will be emitted and in what order.

The **SOMTEmitC** class provides methods for emitting different standard sections of an output file. The standard sections are as follows. The default name is in parentheses:

Prolog
Describes text to be emitted before any other sections (**prologS**).

Base Includes
Determines how base (parent) class #include statements are emitted (**baseIncludesS**).

Meta Include
Determines how a metaclass #include statement is emitted (**metaIncludeS**).

Class
Determines what information about the class as a whole is emitted (**classS**).

Base
Determines what information about the base (parent) classes of a class is emitted (**baseS**).

Meta
Determines what information about the class's metaclass is emitted (**metaS**).

Constant
Determines what information about user-defined constants is emitted (**constantS**).

Typedef
Determines what information about user-defined types is emitted (**typedefS**).

Struct
Determines what information about user-defined structs is emitted (**structS**).

Union
Determines what information about user-defined unions is emitted (**unionS**).

Enum
Determines what information about user-defined enumerations is emitted (**enumS**).

Attribute
Determines what information about the class's attributes is emitted (**attributeS**).

Methods
Determines what information about the methods of a class is emitted (**methodsS**). More specialized method sections can be specified using **inheritedMethodsS** or **overrideMethodsS**.

Release
Determines how information about the release order statement of a class definition is emitted (**releaseS**).

Passthru
Determines what information about passthru statements is emitted (**passthruS**).

Data
Determines what information about internal instance variables of a class is emitted (**dataS**).

Interface
Determines what information about the interfaces in a module is emitted (**interfaceS**).

Module
Determines what information about a module is emitted (**moduleS**).

Epilog
Describes text to be emitted after all other sections are emitted (**epilogS**).

Some sections apply to a variable number of items that must be dealt with iteratively. This can be true of the *base* section (since a class can have more than one base class), as well as the sections for base includes, data, passthru, attribute, constant, typedef, struct, union, enum, interface, module, and method. These repeating sections can be preceded by a prolog (information to be emitted prior to iterating through the items), and followed by an epilog (information to be emitted after iterating through the items). Names for the standard prolog and epilog sections are as follows:

basePrologS, baseEpilogS, baseIncludesPrologS, baseIncludesEpilogS, constantPrologS, constantEpilogS, typedefPrologS, typedefEpilogS, structPrologS, structEpilogS, unionPrologS, unionEpilogS, enumPrologS, enumEpilogS, passthruPrologS, passthruEpilogS, dataPrologS, dataEpilogS, attributePrologS, attributeEpilogS, methodsPrologS, methodsEpilogS, interfacePrologS, interfaceEpilogS, modulePrologS, and moduleEpilogS**.**

The **SOMTEmitC** class provides methods for emitting each of the sections described above. For example, the **somtEmit<Section> Methods** emits the **prologS** section, the **somtEmitClass** method emits the classS section, and so on. For repeating sections, the **SOMTEmitC** class provides scanning methods. These scanning methods first emit the appropriate prolog section, then iterate through the appropriate items in the interface definition, emitting the appropriate section for each item, then emit the appropriate epilog section.

The following **somtScan<Section> Methods** are provided by **SOMTEmitC**:

somtScanBases
somtScanBasesF
somtScanConstants
somtScanTypedefs
somtScanStructs
somtScanUnions
somtScanEnums
somtScanAttributes
somtScanMethods
somtScanData
somtScanDataF
somtScanPassthru
somtScanInterfaces
somtScanModules.

The **somtScanBasesF**, **somtScanDataF** and **somtScanMethods** methods accept a filter argument, for selective scanning. **The Section-Name symbols** on page 395 lists all the section-emitting methods defined by **SOMTEmitC** and the sections that they output. The Reference portion of this book describes each section-emitting method in more detail.

User-defined subclasses of **SOMTEmitC** can override the section-emitting methods to change the way that a particular section is emitted. They can also define new section-emitting methods (see **Customizing Section-Emitting Methods** on page 387).

Finally, the **SOMTEmitC** class provides several filter methods. These methods return TRUE or FALSE depending on some characteristic of a specified entry object. For example, **somtNew** determines whether the specified method is introduced by the emitter's target class. These filter methods can be used as arguments to the **somtScan<Section> Methods** to control which methods are processed in a repeating section.

Filter methods provided by **SOMTEmitC** include:

somtNew Method
somtImplemented Method
somtOverridden Method
somtInherited Method
somtAll Method
somtNewProc Method
somtNewNoProc Method
somtVA Method

The reference portion of this book describes each of these methods in more detail.

SOMTTemplateOutputC

The **SOMTTemplateOutputC Class** handles as much as possible of the formatting part of emitter writing, largely by using symbol-based output templates. A *symbol* is a name used to represent a corresponding value. For example, the symbol (or symbol name) `className` is recognized by the Emitter Framework as representing the name of the target class.

An emitter writer uses symbol names as placeholders in a text template that patterns the desired output. The template object (of class **SOMTTemplateOutputC**) takes a text template containing symbol names and produces output by substituting data for the symbols that occur in the text template. The values that replace the symbol names come from a symbol table maintained by the template object.

The template file is divided into sections that specify the desired output for each syntactic unit of the input IDL specification. To generate a particular section of an output file, an emitter first sets the values of appropriate symbols in its template-object's symbol table, and then specifies to the template object the name of a section to be output. This design results a good separation between decision logic and format specification. Also, because the format specification is isolated, its readability and maintainability are greatly enhanced.

Following is an example fragment of a text template containing two template sections, classS and metaS. The text template is stored in a template file associated with the emitter. (The subsequent paragraphs provide further explanation for preparing the text template.)

```
:classS
class: <className><, classMods, ...>;
?<-- classComment>
:metaS
metaclass: <metaName>
```

New sections are denoted by lines that begin with a colon. The above fragment contains two sections, classS and metaS. (By convention, section names end in capital "S".) An emitter uses the section name to specify to the template object which part of the output file to emit. Lines that begin with a question mark are emitted only if at least one symbol appearing on the line is defined with a nonblank value. Other lines are emitted unconditionally.

Symbols are specified in a template file in angle brackets. Thus, the template above contains the symbols className, classMods, classComment, and metaName. (A backslash can be used to escape an angle bracket when it is not intended to indicate a symbol.) When a template section is emitted, symbols are replaced with their values. If the symbol has no value, then the symbol is replaced by the string `symbol <...> is not defined`, but no error is raised.

In addition to simple symbol substitution, two forms of complex symbol substitution are supported: list substitution and comment substitution. Each of these involves special syntax, as follows.

Comment substitution is specified with two dashes preceding the symbol name (for example, `<-- symbolName>`). When comment substitution is used to emit a symbol, the symbol's value is emitted in comment form. The emitter controls the format for comments by setting the values of its template object's **somtCommentStyle** and **somtCommentNewline** attributes:

- The **somtCommentStyle** attribute determines whether comments are emitted with "- -" at the start of each line, with "//" at the start of each line, in simple C style with each line wrapped in "/*" and "*/", or in block C style with a leading "/*", then a "*" on each line and a final "*/".

- The **somtCommentNewline** attribute is a boolean that determines whether the comment starts on a new line.

List substitution replaces a symbol with its value expressed in list form, using specified delimiters. The symbol's value must consist of a sequence of items, separated by newline characters. The list substitution specification consists of two pieces of information in addition to the symbol name: the prefix to put in front of non-empty lists, and the delimiter to put between list items.

All characters before the symbol name are taken as the prefix, and all characters after the symbol name and before the required "..." (which indicates that list substitution is to be used) are taken as the separator characters. Thus <: symbolName, ...> specifies a prefix of ": " and a separator of ", ". The prefix and separator characters must consist of blanks, commas, colons, and semicolons. The value of the template object's **somtLineLength** attribute controls how many list items are emitted on each line.

Within an output template, tabbing can be specified by `@dd`, where `dd` is a valid positive integer representing a column number. After a `@dd` is encountered in the output template, the next character emitted will appear in the specified column.

To emit the "classS" and "metaS" sections from the above template, the following IDL specification could be used as input:

```
#include <somobj.idl>
#include <mhello.idl>

interface Hello : SOMObject  /* This is the Hello interface. */
{
    implementation {
    metaclass = M_Hello;
    functionprefix = "hello_";
    filestem = hello;
...}
;
};
```

The preceding IDL specification would produce the following output:

```
class: Hello, functionprefix = hello_, filestem = hello;

// This is the Hello interface.
metaclass: M_Hello
```

The formatting of comments varies, depending on the attributes of the emitter's template.

The **SOMTTemplateOutputC Class** provides methods that:

- Set and get the value of symbols in a template object's symbol table.
- Emit a particular section of the output template (**somtOutputSection Method**).
- Emit a comment, **somtOutputComment Method**.
- Read the output template file (**somtReadSectionDefinitions Method**), and others.

Defining New Symbols on page 385 describes how to use the symbol-setting methods to define new symbols.

SOMTEntryC and SOMTClassEntryC

The purpose of these classes is to hide the syntax of the **.idl** file. They return information about an IDL interface definition in a way that is neutral to the source syntax of the IDL definition and to the nature of the emitter in which the information will be used.

The entry classes are arranged into the class hierarchy shown below.

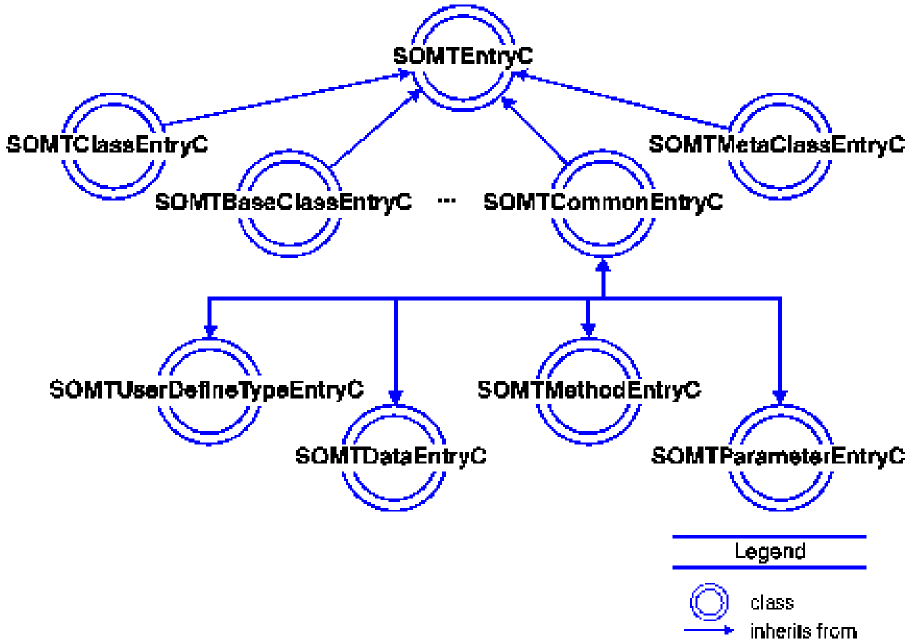

Figure 26. Entry Class Hierarchy

With the exception of **SOMTEntryC** and **SOMTCommonEntryC**, all of the entry classes correspond to a specific unit of information in an IDL interface definition. This correspondence is summarized in the following topics.

SOMTEntryC

The **SOMTEntryC Class** provides attributes for accessing the name of an entry, its entry type, its comment, the line number in the **.idl** file where the entry is defined, its type code and whether the entry represents a reference to an entry rather than its definition.

The **SOMTEntryC** class also provides methods for accessing the SOM IDL modifiers specified in the implementation section of an interface statement. Included are:

- **somtGetModifierValue Method**
- **somtGetFirstModifier Method**
- **somtGetNextModifier Method**
- **somtFormatModifier Method**
- **somtGetModifierList Method**

When invoked on an instance of **SOMTClassEntryC** SOMTClassEntryC, these methods pertain to the class's modifiers; when invoked on an instance of **SOMTMethodEntryC**, they pertain to the method's modifiers, and so on.

The **SOMTEntryC** class provides the **somtSetSymbolsOnEntry Method** you can use to create symbols and define their corresponding values for use in the output template. For example, **SOMTClassEntryC** class's implementation of **somtSetSymbolsOnEntry** establishes the symbol className containing the name of the current class, **SOMTMethodEntryC**'s implementation of **somtSetSymbolsOnEntry** defines the `methodName` symbol and so on.

SOMTCommonEntryC

Entry objects that an emitter uses are instances of one of **SOMTCommonEntryC Class** subclasses, rather than of **SOMTCommonEntryC** itself. These subclasses are:

- **SOMTMethodEntryC Class**
- **SOMTDataEntryC Class**
- **SOMTUserDefinedTypeEntryC Class**
- **SOMTParameterEntryC Class**

The **SOMTCommonEntryC** class provides attributes and methods for obtaining information about the type of a method, parameter, user-defined type, attribute declarator, struct member declarator, or instance variable. For example, it provides the attribute whose value is a pointer to a **SOMTEntryC** object representing the type, the attribute **somtType** that gives a string representation of the type, the attribute **somtArrayDimsString** that indicates array dimensions, and the attribute **somtPtrs** that gives the number of stars associated with a pointer type.

The **SOMTCommonEntryC** class also provides methods for accessing type information.

SOMTClassEntryC

A **SOMTClassEntryC** object anchors the entire interface definition for a class. All the parts of a class's interface definition are reachable from the **SOMTClassEntryC** object entry. When an emitter is run on a class's interface definition, the emitter has a distinct class entry called the target class entry which represents that class.

The **SOMTClassEntryC** class provides attributes corresponding to the following characteristics of an IDL interface specification:

- Its source file name
- Its metaclass
- The class this class is a metaclass for, if any
- Whether the entry represents a forward declaration of the class, rather than its definition
- The module that contains the class, if any
- The number of methods the class introduces or overrides
- The number of static methods the class introduces
- The number of procedure methods the class introduces
- The number of variable argument methods the class introduces
- The number of parent (base) classes

The class also provides **methods** for accessing each of a class's: parent classes, release order names, data items, passthrus, methods, constants, attributes, typedefs, structs, unions, enumerations and sequences

SOMTClassEntryC provides methods for accessing all type and constant definitions in the order in which they were defined, including structs, unions, enumerations. These methods are **somtGetFirstPubdef** and **somtGetNextPubdef**. Finally, the **SOMTClassEntryC** class provides filter methods for determining whether a method is new or overridden.

SOMTBaseClassEntryC

Every class entry holds a pointer to a base class entry (**SOMTBaseClassEntryC** object) for each of the class's direct base (parent) classes. The base class entry is not the class entry for a base class but is an object that has an attribute (**somtBaseClassDef**) whose value is the class entry for the base class.

SOMTMetaClassEntryC

Every class entry holds a pointer to its metaclass entry (**SOMTMetaClassEntryC** object) if the class **#includes** the **.idl** file for its metaclass. A metaclass entry is like a base class entry in that it is not the class entry for the metaclass. Rather, it is an object that has an attribute (**somtMetaClassDef**) whose value is the class entry for the metaclass. The metaclass entry also has an attribute (**somtMetaFile**) that specifies the file in which the metaclass's interface is defined.

SOMTModuleEntryC

A **SOMTModuleEntryC** object represents a module within an IDL specification. It provides methods for accessing each of the module's: interfaces, nested modules, constants, typedefs, structs, unions, enumerations and sequences.

SOMTModuleEntryC provides methods for accessing the definitions in the order in which they were defined.

SOMTPassthruEntryC

Every class entry holds a pointer to a passthru entry (**SOMTPassthruEntryC** object) for each passthru specification in the implementation section of the class's SOM IDL interface specification. Each passthru entry has attributes representing the target, the target language, and the passthru's contents, and a method to determine whether the passthru is a before or a after passthru.

SOMTTypedefEntryC

Every class entry holds a pointer to a typedef entry (**SOMTTypedefEntryC** object) for each typedef introduced within the class's interface specification and for each member of a user-defined struct. Each typedef entry provides an attribute representing the base type of the typedef and methods for accessing each of the declarator names of the typedef. Because a single typedef may have several declarators, the **somtTypedefType** attribute of a typedef gives only the base type of the user-defined types; to get the full type, users should access each declarator in turn and get its **somtType** attribute.

SOMTDataEntryC

Every class entry holds a pointer to a data entry (**SOMTClassEntryC** object) for each of the data members (internal instance variables) specified in the implementation section of the class's interface definition, and for each attribute declarator or struct member declarator. The **SOMTDataEntryC** class provides an attribute that indicates whether a struct member declarator is self-referential (pointing to the same type of structure for which it is a declarator).

SOMTAttributeEntryC

Every class entry holds a pointer to an attribute entry (**SOMTAttributeEntryC** object) for each of the attribute definition statements within the class's interface specification. Each attribute entry has attributes representing the base type and whether the attribute is readonly. It also provides methods for accessing the attribute declarators and their get/set methods.

Because a single attribute definition statement may have several declarators (that introduce several attributes), the **somtAttribType** attribute gives only the *base* type of the attributes being defined; to get the full type, users should access each declarator in turn and get its **somtType** attribute.

SOMTMethodEntryC

A class entry holds a pointer to a method entry (**SOMTMethodEntryC** object) for each of the methods the class supports (both new and inherited methods). Each method entry has attributes representing:

- The C/C++ form of the method's return type (**somtCReturnType**)
- Whether the method has a **va_list** parameter (**somtIsVarargs**)
- For overriding methods, the class whose implementation is being overridden (**somtOriginalClass**) and the method being overridden (**somtOriginalMethod**)
- Whether the method is oneway (**somtIsOneway**)
- The number of arguments to the method (**somtArgCount**)
- The context string literals of the method (**somtContextArray**)

The **SOMTMethodEntryC** class also provides methods for getting the parameters.

SOMTParameterEntryC

Method entries contain a reference to a parameter entry (**SOMTParameterEntryC** object) for each of the explicit parameters to the method. (The receiver of the method does not have a corresponding parameter entry; neither do the **Environment** and **Context** parameters, if any.) Each parameter entry has an attribute that indicates whether it is an in, out, or inout parameter, and attributes that give the parameter's declaration within a prototype.

SOMTConstEntryC

Every class entry holds a pointer to a constant entry (**SOMTConstEntryC** object) for each constant defined within the class's interface specification. Each constant entry has attributes that represent the type and the value of the constant.

SOMTEnumEntryC

Every class entry holds a pointer to an enum entry (**SOMTEnumEntryC** object) for each enumeration defined within the class's interface specification. Each enum entry provides methods for getting the enumerator names for the enumeration.

SOMTSequenceEntryC

Every class entry holds a pointer to a sequence entry (**SOMTSequenceEntryC** object) for each sequence defined within the class's interface specification. Each sequence entry has attributes representing the sequence's length and type.

SOMTStringEntryC

Every class entry holds a pointer to a string entry (**SOMTStringEntryC** object) for each string defined within the class's interface specification. Each string entry has an attribute representing the string's length.

SOMTUnionEntryC

Every class entry holds a pointer to a union entry (**SOMTUnionEntryC** object) for each union defined within the class's interface specification. Each union entry provides an attribute representing the union's switch type and methods for accessing each of its cases.

SOMTEnumNameEntryC

Every enumeration entry (of type **SOMTEnumNameEntryC**) holds a pointer to a **SOMTEnumNameEntryC** object for each enumerator name defined within it. Each **SOMTEnumNameEntryC** entry has attributes representing the enumerator name's value and a pointer to the enumeration that defines the enumerator name.

SOMTStructEntryC

Every class entry holds a pointer to a struct entry (**SOMTStructEntryC** object) for each struct defined within the class's interface specification and for each exception the class defines. Each struct entry provides attributes that represent the class in which the struct was defined and whether the struct actually represents an exception, and methods for accessing each of the struct members.

SOMTUserDefinedTypeEntryC

Every class entry holds a pointer to a user-defined type entry (**SOMTUserDefinedTypeEntryC** object) for each type defined within the class's interface specification via a typedef statement. Each user-defined type entry provides attributes representing the typedef statement that defined the type and the base type of the user-defined type. The **somtBaseTypeObj** attribute gives the primitive IDL type that underlies a user-defined type, skipping over any intermediate user-defined types.

Writing an Emitter: the Basics

This section describes writing an emitter using the **newemit** program.

The newemit Facility

The **newemit** emitter generator is a program, written using the Emitter Framework, that generates a complete, working emitter. This emitter is easily customized as needed.

The following steps outline the recommended approach to writing an emitter.

Running the newemit Program

The command to execute the **newemit** program takes the following syntax:

newemit [**-C** | **-C++**] ***className filestem***

Required arguments are the name for a new subclass of **SOMTEmitC Class** to be created and a file stem for the subclass. The optional `-C` or `-C++` specifies the language in which the emitter will be written; the default is C. The program produces the following files:

*filestem***.idl**
: An IDL definition of the subclass of **SOMTEmitC** with the specified name. This IDL definition specifies that the **somtGenerateSections Method** will be overridden by the new subclass.

filestem.c
The C or C++ implementation file for the new subclass of **SOMTEmitC**. Initially, this implementation file has the same code for **somtGenerateSections** as defined by **SOMTEmitC**. (The C++ extension is .C on AIX or **.cpp** on OS/2.)

emit*filestem***.c or emit***filestem***.C or emit***filestem***.cpp**
A C or C++ emitter driver program. The driver-program name is always “emit” followed by the specified file stem. (The C++ extension is **.C** or on AIX or **.cpp** on OS/2.)

Makefile
A Makefile for creating a DLL for the new emitter.

*filestem***.efw**
A sample output template file.

The *filestem***.c** (or **.C** or **.cpp**) and *filestem***.efw** files will be customized to produce a particular output format. For example, to create a documentation emitter class called `DocEmitter` whose related files (written in C by default) have a file stem of `doc`, we would execute the following command:

```
newemit DocEmitter doc
```

The result would be files **doc.idl**, **doc.c**, **emitdoc.c**, Makefile, and **doc.efw**. The remaining steps involve customizing the doc.c and doc.efw files.

Note: For AIX, the <*filestem*> argument to the **newemit** program should consist of *only* lowercase characters.

Designing the Output File

Look at a typical IDL interface definition, and hand-construct the desired output file for that interface. For example, suppose we want our `DocEmitter` to construct a documentation file that simply lists the class name and the return types for the methods it introduces or overrides. Thus, given the following IDL specification,

```
#include <somobj.idl>
interface Animal: SOMObject {
void setSound(in string sound);
void makeSound();
};
```

we would want the output file to look like this:

```
The following methods:
  setSound, of type void
  makeSound of type void
are implemented by class Animal.
```

Constructing an Output Template

The next step is to construct an output template, based on the sample output file. Separate the sample output file into different sections, based on the aspect of the IDL specification to which they most closely correspond. For example, the first three lines of the output file above correspond to method declarations, and the last line corresponds to the class interface definition as a whole.

Although the first three lines all correspond to method declarations, they should be further divided into the portion that constitutes the prolog (to be emitted only once, regardless of how many methods are to be described), the repeating portion (which should be emitted once per method), and the portion that constitutes the epilog (also emitted only once). The first line in the sample output above constitutes the prolog for the method-describing section, and the second two lines are representative of the repeating method-describing section. There is no epilog section used in this example, although the last line of the output shown could be made part of the methods epilog section, rather than the class section.

Next, assign the section names. By convention, section names end in uppercase “S”. Each section name is given on a separate line, preceded by a colon and followed by the text lines that make up the section.

The most appropriate section names for the DocEmitter output template are **methodsPrologS**, **methodsS** and **classS**.

Then, generalize the text lines within each section into a generic template. That is replace strings that are specific to a particular interface definition with symbols that represent the syntactic unit of the interface definition from which the string was taken. For example, string Animal in the current example can be replaced by the standard symbol `className`.

In sum, the output file shown above could be generalized to the following output template:

```
:methodsPrologS
The following methods:
:methodsS
  <methodName>, of type <methodType>
:classS
are implemented by class <className>.
```

Output templates are typically stored in files having a **.efw** extension. The **newemit** program creates a generic template file, *filestem*.**efw**. It contains all the standard sections, and each section contains sample template text that exercises all the standard symbols available within that section. This generic template can be edited to contain only those sections needed by the new emitter, with appropriate text. For the current example, file **doc.efw** would be edited to contain the generalized output template shown above.

Note: Each of the entry classes defines a set of standard symbols based on the kind of entry they represent. These symbols are discussed in **Standard Symbols** on page 390. **The Section-Name symbols** on page 395 lists all standard section names, along with the method that emits the section having that section name. New symbols (and new section names) can also be defined, if needed, as described in **Defining New Symbols** on page 385.

Customizing Emitter Control Flow

The **newemit** program creates a subclass of **SOMTEmitC Class** (in this example, `DocEmitter`) that overrides the **somtGenerateSections Method**. The **newemit** program also provides a default implementation of **somtGenerateSections** for `DocEmitter` in the **doc.c** file. This implementation should be customized.

The **somtGenerateSections** method determines which sections of the output template are emitted and in what order. The output template only specifies which sections are available to the emitter and their contents; it does not control which sections are actually emitted or their order.

For the current example, we want our emitter to first emit the **methodsPrologS** section of the output template, then the **methodsS** section, once for each method introduced by the class, followed by the classS section. However, the default implementation of **somtGenerateSections,** provided by **newemit**, emits the **classS** section first; thus, we must switch the order in which the sections are emitted.

The default implementation of **somtGenerateSections** also emits other sections; however, because those sections are not defined in our example output template, those portions of code should be removed.

The crucial portions of `DocEmitter`'s implementation of **somtGenerateSections** (in **doc.c**, and after the class and methods section order has been switched) are shown below:

```
SOM_Scope boolean SOMLINK somtGenerateSections(DocEmitC
                                               somSelf)
{
  /* Define symbols available in all sections of the
   * output template.
   */
  _somtFileSymbols(somSelf);

  if (cls != (SOMTClassEntryC *) NULL) {
  /* Emit "methods" section for each method of the class.
   * If a "methodsProlog" section is defined it precedes
   * the first method; if a "methodsEpilog" section is
   * defined it will follow the last method.
   */
    _somtScanMethods(somSelf, "somtImplemented",
           "somtEmitMethodsProlog", "somtEmitMethod",
           "somtEmitMethodsEpilog", 0);
    _somtEmitClass(somSelf);  /* emit "class" section */
  }
   return (TRUE);
}
```

Use of the **somtScan<Section> Methods** to iterate through the class's methods and emit the methodsS section for each method. **SOMTEmitC** defines scanning methods for data items, base classes, passthrus, attributes, constants, typedefs, struct, enums, union, interfaces and nested modules.

Compiling and Running the New Emitter

Compile the driver program provided by **newemit** and the implementation of your emitter together to create a dynamically linked library (a DLL) for a new emitter. The **newemit** program provides a Makefile to perform this step (simply enter "make").

The new emitter can now be invoked by the SOM Compiler via the **-s** option (which overrides the SMEMIT variable, for the current **sc** command, with the specified emitter). For example, DocEmitter, packaged in **emitdoc.dll**, can be invoked by running the SOM Compiler with the -sdoc option. (The value of the **-s** option is the file stem specified earlier to **newemit**.) Invoking the following **sc** command will produce an animal.doc file just like the one shown at the beginning of this section:

```
sc –sdoc animal.idl
```

Debugging an Emitter

To debug an emitter, run the SOM Compiler as always, but include the **-v** option. This flag causes the SOM Compiler to tell you how it runs the various programs that make up the SOM Compiler. The first program run is the preprocessor, usually **somcpp**, which you can ignore. The second program is the compiler front end, **somipc**. The front end loads and runs the individual emitter DLLs that you request via the **-s** flag. To debug an emitter, you must debug **somipc**. You can run the programs the SOM Compiler ran by copying the information supplied by the **-v** option into an executable file. Modify the line that runs the **somipc** program so that the debugger is used to run **somipc**.

OS/2 example:

Here is the command to run the **def** emitter with the **-v** option included:

```
sc –v –sdef foo.idl
```

Here are the results obtained on one OS/2 system (Some of the lines have wrapped):

```
Running shell command:
  somcpp –D__OS2__ –I. –C foo.idl > C:\tmp\4150000000.CTN
  somipc –mppfile=C:\tmp\4150000000.CTN –v –e emitdef
        –o foo foo.idl
Loading emitdef.
”foo”
Unloading emitdef.
Removed ”C:\tmp\4150000000.CTN”
```

Most debuggers require the full path to the program you will debug, so this example includes the path to **somipc** on the system where the command was run. Here is the **.cmd** file created to run IBM's **ipmd** debugger:

```
somcpp –D__OS2__ –I. –C foo.idl > C:\tmp\4150000000.CTN
ipmd r:\bin\somipc –mppfile=C:\tmp\4150000000.CTN –v
    –e emitdef –o foo foo.idl
```

All emitters begin with a function named **emit**. You should set a breakpoint there using the debugger.

Writing an Emitter: Advanced Topics

Defining New Symbols

The Emitter Framework defines a number of symbols that can be used in output templates. (See **Standard Symbols** on page 390.) Programmers can define additional symbols as needed. This is usually done within an overriding implementation of the **somtGenerateSections** method or some other method of a user's subclass of **SOMTEmitC**.

Symbol names defined by the Emitter Framework have been chosen to maintain the readability of the template file. There is very little cost associated with the length of a symbol name, nor is there any practical limit on the length of a symbol name. To maintain the readability of template files, it is suggested that emitter writers follow the pattern of the standard symbol names. Symbol names may not consist of double-byte characters.

The value of a defined symbol can be obtained from a template object using the **somtGetSymbol** method. For example, in a C program,

```
_somtGetSymbol(t, ”className”);
```

returns the value of the "className" symbol, assuming that *t* points to a template object for an emitter. The **somtTemplate** attribute of an emitter refers to its template object. The **somtCheckSymbol** method can be used to determine whether or not a given symbol is defined.

New symbols are defined using one of the following methods, invoked on a template object:

- **somtSetSymbol Method**
- **somtSetSymbolCopyValue Method**
- **somtSetSymbolCopyName Method**
- **somtSetSymbolCopyBoth Method**

Arguments for each method are the name of the symbol and its value. The differences among the four symbol-setting methods are if they make a copy of the name/value to store in the symbol table, or if the passed strings are stored. If no copy is made, the string must not be subsequently freed by the calling program. The **somtSetSymbolCopyValue** method is useful for redefining a symbol that already has a value in the symbol table. The **somtSetSymbolCopyName** method is useful when passing a string value that has been allocated and will not be freed. The **somtSetSymbol** method is useful when both situations co-occur. Typically, however, the **somtSetSymbolCopyBoth** method is used. For example, to set the value of the `NewSym` symbol to value `Hello!`, use the following C method call:

```
_somtSetSymbolCopyBoth(t, ”newSym”, ”Hello!”);
```

where *t* is the template object for an emitter.

Another method that can be used to define symbols is **somtExpandSymbol**. This method can be used to set a symbol to a value specified within the output template. Given a symbol representing the name of a section in the output template, the **somtExpandSymbol** method expands that section into a buffer by substituting symbol values for symbol names in the template. The result can then be assigned as the value of a symbol, using one of the symbol-setting methods above. In this way, the values of emitter symbols can be defined declaratively in the template file, rather than procedurally within the emitter's code. For example, if the template (.efw) file for an emitter contains the following section definition:

```
:methodPrefixS
<functionprefix>_
```

then the following C code within the implementation of an emitter's method will set symbol "methodPrefix" to be the expansion of the "methodPrefixS" section in the template file (that is, the value of symbol "functionprefix," if defined by the emitter, followed by an underscore).

```
SOMTTemplateOutputC t = __get_somtTemplate(somSelf);
char buf[MAX_BUF_SIZE];
...
_somtSetSymbolCopyBoth(t, ”methodPrefix”,
                     _somtExpandSymbol(t, ”methodPrefixS”, buf));
```

In addition to defining new symbols within a programmer's subclass of **SOMTEmitC**, new symbols can also be defined within user-defined subclasses of an entry class. In this way, the new symbols will be defined at the same time the *standard* symbols are defined. This technique is helpful when the symbols will be useful to multiple emitters. Each of the emitters can use the new entry class rather than defining the symbols.

To define new symbols within a user-defined subclass of an entry class, override the **somtSetSymbolsOnEntry** method. For example, to define some new symbols to be used in the "classS" section, and to have these symbols automatically defined for every class entry (rather than requiring every emitter to define them), create a subclass of **SOMTClassEntryC** that overrides the **somtSetSymbolsOnEntry** method. Within the overriding method, invoke the parent method, then use one of the symbol-setting methods to define the new symbols.

For instance, a user-defined subclass `SMPClassEntryC` of **SOMTClassEntryC** might override **somtSetSymbolsOnEntry** as follows:

```
SOM_Scope long SOMLINK somtSetSymbolsOnEntry(
                        SMPClassEntryC somSelf,
                        SOMTEmitC emitter, string prefix)
{
  SOMTTemplateOutputC t = __get_somtTemplate(emitter);
  long status;
  status = parent_SOMTClassEntryC_somtSetSymbolsOnEntry
                           (somSelf, emitter, prefix);
  _somtSetSymbolCopyBoth(t, _somtNewSymbol(prefix,
                                           ”ExtPrefix”),
      _somtGetModifierValue (somSelf, ”externalprefix”);
```

```
    return(status);
}
```

The parent method is invoked first to define the standard symbols, then a new symbol is defined whose value is the externalprefix modifier for the class.

The prefix parameter of the **somtSetSymbolsOnEntry** method is prefixed to each of the standard symbol names that the method defines. The prefix is set by the emitter framework to match the role of that entry in the class interface definition. For example, the standard symbols defined by a class entry that corresponds to the target class of the emitter will be prefixed with “class”, the standard symbols set by the metaclass entry of the target class will be prefixed with “meta”, and so on. The **somtNewSymbol** function is provided for users to create new symbols from the prefix passed to **somtSetSymbolsOnEntry** in overriding implementations of **somtSetSymbolsOnEntry.**

It is important that overriding implementations of **somtSetSymbolsOnEntry** in subclasses of **SOMTClassEntryC** in particular use **somtNewSymbol** and the *prefix* parameter to define new symbols because that method will be invoked not only to define symbols for the target class, but also to define symbols for its base classes and metaclass (for which “prefix” will be “base” or “meta”).

When subclassing one of the entry classes, it is necessary to use shadowing to have the object graph builder use the new subclass when constructing the object graph, rather than the original entry class. Otherwise, subclassing the entry class will have no effect. See **Shadowing** on page 388.

Customizing Section-Emitting Methods

The **somtGenerateSections** method invokes section-emitting methods. To specialize the behavior of one of these methods, we could override them. This would allow us, for instance, to set symbols differently before emitting the methodsS section, depending on the characteristics of the method.

An emitter can define new section-emitting methods. For example, an emitter could introduce a new section-emitting method, `somtEmitMethod2`. The new section-emitting method can be passed by **somtGenerateSections** as an argument to **somtScanMethods**, so that for each of the target class's methods, `somtEmitMethod2` is invoked. User-defined sections can also be emitted by changing the value of one of the predefined section-name symbols, as described under **Changing Section Names** on page 387.

Section-emitting methods take as an argument the entry object about which information is to be emitted. For example, an argument to **somtEmitMethod** is a method entry object. Each such entry object supports methods for obtaining information about the portion of the IDL interface specification it represents. For example, a method entry object has an attribute, **somtArgCount**, that gives the number of parameters the method has, and every entry also supports the **somtGetModifierList** and **somtGetModifierValue** methods for obtaining specific information about SOM IDL modifiers. This information can be used to guide the behavior of the section-emitting methods.

User-defined implementations of section-emitting methods typically define new symbols as needed, as described above, and then invoke the **somtOutputSection** method to produce output from the appropriate template section.

Changing Section Names

Each predefined section-emitting method in the Emitter Framework determines which section of the output template to emit based on the value of a predefined section-name symbol. For example, the **somtEmitProlog** method emits the section whose name is specified by the `prologSN` symbol. The default value of the `prologSN` symbol is `prologS`. Thus, **somtEmitProlog** emits the `prologS` section of the output template by default. The table at the end of the next section lists the default values of all predefined section-name symbols and indicates which section emitting methods use them.

To change the section that a section-emitting method emits, simply change the value of the appropriate section-name symbol. For example, to have **somtEmitProlog** emit section `myPrologS` instead of section `prologS`, invoke the **somtSetSymbol** method as follows from within the **somtGenerateSections** method of your emitter, prior to invoking **somtEmitProlog**:

```
_somtSetSymbolCopyValue(t, "prologSN", "myPrologS");
```

This technique allows an emitter to use the same section-emitting method to emit multiple sections of the output file. For example, we could have both a `prologS` section and a `myPrologS` section in the output template. The first time **somtGenerateSections** invokes **somtEmitProlog**, it will emit the prologS section. Prior to invoking **somtEmitProlog** a second time, the emitter changes the value of the `prologSN` symbol, as above, to `myPrologS`. Thus, the second time the emitter invokes **somtEmitProlog**, it will emit the `myPrologS` section.

Note: User-defined section names may not contain double-byte characters.

Shadowing

Some emitters may require subclassing one or more of the entry classes to add new methods or override existing methods. For example, changing the behavior of the **somtGetNextParameter** method would require subclassing the **SOMTMethodEntryC** class. As another example, if an emitter needs symbols that are not predefined by the Emitter Framework, and these symbols would be useful to multiple emitters, then, rather than defining these symbols in **somtGenerateSections** for every emitter, it may be advantageous to subclass one or more of the entry classes and to override the **somtSetSymbolsOnEntry** method in that subclass.

When an emitter subclasses one or more of the entry classes, the driver program that instantiates the emitter must be modified to use *shadowing*. Shadowing instructs the object graph builder to create instances of the new subclass as it builds the object graph to represent the input.

Shadowing allows an emitter to substitute a subclass of an entry class for the parent class without having to recompile the library routines that use the original class. The library routines will automatically pick up all of the changes made in the new subclass when shadowing is used.

Shadowing is accomplished using the **SOM_SubstituteClass** macro. For each user-defined subclass of an entry class, modify the **emit** function in the driver program to include the following instruction, just after the call to **somtopenEmitFile**:

```
SOM_SubstituteClass(<existing entry class name>,
                    <new subclass name>);
```

For example, to shadow entry class **SOMTClassEntryC** with user-defined subclass **SMPClassEntryC,** add the following instruction to the emit function, just after the call to **somtopenEmitFile**:

```
SOM_SubstituteClass(SOMTClassEntryC, SMPClassEntryC);
```

When shadowing an entry class, the header file for the class being shadowed must be included in the driver program. For example, shadowing **SOMTClassEntryC** would require adding the directive

```
#include <scclass.h>
```

in the driver program (contained in **emit***filestem***.c)**.

Handling Modules

When an emitter is run on a **.idl** file that contains a module, the *cls* argument to the emit function in the emitter's driver program will be a structure (`cls->type== SOMTModuleE`), rather than (`cls->type == SOMTCLassE`). The default implementation of the driver program, provided by **newemit**, creates an emitter having a *target module*, rather than a *target class*, then invokes **somtGenerateSections** on that emitter as usual. The default implementation of **somtGenerateSections** method, in turn, invokes different section-emitting methods depending on whether the emitter has a target module or a target class.

When an emitter is invoked on a module, the emitter should emit only the information pertaining to the module as a whole and any typedef and constant definitions within it. Information pertaining to each of the *interface* specifications contained in the module will be emitted subsequently, on a separate invocation of the emitter.

In other words, when the SOM Compiler processes a **.idl** file containing a module that includes multiple interface statements, it first runs the requested emitters, passing a structure representing the module. It then runs the same emitters again, passing a structure representing the first interface in the module. It then runs the same emitters again, passing a structure representing the next interface in the module, and so on.

All output goes to the same output file, even though the output is produced by multiple invocations of the emitter. This is controlled by a global variable, set by the SOM Compiler, that indicates to the **somtopenEmitFile** function whether the output file should be opened for writing or for appending. Thus, the first time the emitter is invoked on a particular input file, a new output file is created, but subsequent invocations of the emitter on the same input file simply append to the same output file.

Because an emitter that is handling a module has no target class, users should avoid invoking any method of **SOMTEmitC** that requires a target class if the emitter is handling a module.

Error Handling

The Emitter Framework provides a set of functions to facilitate error handling within user-written emitters. The following functions can be used to issue informational or error messages of differing levels of severity: **somtmsg, somtwarn, somterror, somtfatal** and **somtinternal**. These functions optionally take the file name and line number where the error occurred and a format string and arguments to be passed to the **printf** C library function. The functions increment the relevant error count and print a message that contains the file name and line number (if specified), an indication of the severity of the message, and the message itself. In addition, the **somtfatal** and **somtinternal** functions remove the output file being constructed and terminate the process. Below is an example of producing an error message using the **somterror** function:

```
somterror(__get_somtSourceFileName(cls),
          __get_somtSourceLineNumber(entry),
          "I don't understand the entry named %s.\n",
          __get_somtEntryName(entry));
```

When the **somtfatal** or **somtinternal** function is invoked, the output file being constructed is removed and the process is terminated. These actions are also taken if the SOM Compiler detects an internal error within the emitter or if a user-generated interrupt occurs. It may be necessary to prevent these signals from being detected in certain sections of an emitter's code. The Emitter Framework provides two functions, **somtunsetEmitSignals** and **somtresetEmitSignals**, to protect such critical portions of emitter code. These functions take no arguments and return no value. An example is shown below of using these functions to protect a portion of code from signal processing:

```
somtunsetEmitSignals();
/* do some protected processing */
somtresetEmitSignals();
```

Standard Symbols

The following lists of standard symbols are organized in two ways:

- The first category groups the symbols by what sections of the output template may reference them. These are the lists to reference when writing an output template.
- The second category groups the symbols by which entry class defines them. These are the lists to reference when changing the value of a predefined symbol through shadowing as this list indicates which entry class to subclass.

Each symbol is described in more detail in the second part of this section.

Symbols by Section Validity

Valid in **all output template sections**, when an emitter has a target class, and in the **interfaceS** section when an emitter has a target module**:**

```
className
classIDLScopedName
classCScopedName
classComment
classInclude
classLineNumber
classMods
classMajorVersion
classMinorVersion
classSourceFile
classSourceFileStem
classReleaseOrder
timeStamp  (the date and time the emitter was run)
```

If a metaclass is explicitly defined for the class, the following symbols are defined:

```
metaName
metaIDLScopedName
metaCScopedName
metaComment
metaInclude
metaLineNumber
metaMajorVersion
metaMinorVersion
metaMods
metaReleaseOrder
metaSourceFile
metaSourceFileStem
```

Valid within the **baseIncludesS** and **baseS** sections:

```
baseName
```

```
baseIDLScopedName
baseCScopedName
baseComment
baseInclude
baseLineNumber
baseMajorVersion
baseMinorVersion
baseMods
baseReleaseOrder
baseSourceFile
baseSourceFileStem
```

Valid in the **methodsS**, **overrideMethodsS** and **inheritedMethodsS** sections:

```
methodName
methodIDLScopedName
methodCScopedName
methodComment
methodLineNumber
methodMods
methodType
methodCReturnType
methodContext
methodRaises
methodClassName
methodCParamList
methodCParamListVA
methodIDLParamList
methodShortParamNameList
methodFullParamNameList
```

Valid in the **dataS** section:

```
dataName
dataIDLScopedName
dataCScopedName
dataComment
dataLineNumber
dataMods
dataType
dataArrayDimensions
dataPointers
```

Valid in the **passthruS** section:

```
passthruName
passthruComment
passthruLineNumber
passthruMods
passthruLanguage
passthruTarget
passthruBody
```

Valid in the **constantS** section:

```
constantName
constantIDLScopedName
constantCScopedName
constantComment
constantLineNumber
constantMods
constantType
constantValueUnevaluated
constantValueEvaluated
```

Valid in the **typedefS** section:

```
typedefDeclarators
```

```
typedefBaseType
typedefComment
typedefLineNumber
typedefMods
```

Valid in the **structS** section:

```
structName
structIDLScopedName
structCScopedName
structcomment
structLineNumber
structMods
```

Valid in the **unionS** section:

```
unionName
unionIDLScopedName
unionCScopedName
unionComment
unionLineNumber
unionMods
```

Valid in the **enumS** section:

```
enumName
enumIDLScopedName
enumCScopedName
enumComment
enumLineNumber
enumMods
enumNames
```

Valid in the **attributeS** section:

```
attributeDeclarators
attributeBaseType
attributeComment
attributeLineNumber
attributeMods
```

Valid in the **moduleS** section:

```
moduleName
moduleIDLScopedName
moduleCScopedName
moduleComment
moduleLineNumber
moduleMods
```

Symbols by Entry Class Availability

The following symbols are established and defined for each object of the indicated entry class when the **somtSetSymbolsOnEntry Method** is invoked on that object.

For SOMTEntryC Class

*prefix***Name**
The unscoped name of the entry.

*prefix***IDLScopedName**
The scoped name of the entry, using "::" as delimiters.

*prefix***CScopedName**
The scoped name of the entry, using "_" as delimiters.

*prefix***Comment**
The comment that follows the entry in the IDL specification.

*prefix***LineNumber**
The line number where the IDL specification of the entry ends.

*prefix***Mods**
The SOM IDL modifiers of the entry.

where *prefix* is replaced by the corresponding IDL syntactic unit being defined, as **module**, **attribute**, **constant**, **typedef**, **struct**, **enum**, **union**, **class**, **base**, **meta**, **method**, **data**, **passthru** or a user-specified prefix.

For SOMTCommonEntryC Class

*prefix***Type**
The type of the entry. For methods, the return type.

*prefix***ArrayDimensions**
The array dimensions of the entry, if it is an array.

*prefix***Pointers**
The pointer stars for the entry, if it is a pointer type.

where *prefix* is replaced by either **method**, **data** or a user-specified prefix.

For SOMTAttributeEntryC Class

attributeDeclarators
The list of attribute declarators.

attributeBaseType
The base type of the attributes, not including pointer stars or array dimensions, if any.

For SOMTEnumEntryC Class

enumNames
The list of enumerator names of the enumeration.

For SOMTClassEntryC Class

classMajorVersion
The class's major version number.

classMinorVersion
The class's minor version number.

classSourceFile
The name of the IDL source file.

classSourceFileStem
The file stem of the binding files to be produced from the input IDL file. If the input IDL has a has a *filestem* modifier, then its value defines the symbol. Otherwise, the symbol will be the filestem of the input IDL file.

classReleaseOrder
The release order list for the class.

classInclude
The expression to be used in include statements to access the appropriate file for this class such as **somobj.idl**.

For SOMTConstEntryC Class

constantType
The type of the constant.

constantValueEvaluated
The evaluated value of the constant. For constants of type string or char, this value includes the quotes.

constantValueUnevaluated
The unevaluated value of the constant. Constants within the expression are, however, replaced with their values. For constants type string or char, this value does not include the quotes.

For SOMTMethodEntryC Class

methodCReturnType
The C or C++ return type of the method.

methodClassName
For an overriding method, the class whose method is overridden. For new methods, the introducing class.

methodIDLParamList
The formal parameter list (including types) for the method, in IDL form (includes only *explicit* parameters).

methodCParamList
The formal parameter list (including types) for the method's procedure, in C or C++ form (including all parameters).

methodCParamListVA
The formal parameter list (including types) for the method's procedure, in C or C++ form (including all parameters), with any **va_list** parameter replaced by "...".

methodShortParamNameList
A list consisting of the names of the method's explicit parameters (excluding *somSelf, ev,* and *ctx*).

methodFullParamNameList
A list consisting of the names of all of the method procedure's formal parameters (including implicit method parameters and *somSelf)*.

methodRaises
A list of the exceptions the method may raise.

methodContext
A list of the context string literals for the method.

For SOMTParameterEntryC Class

parameterDirection
Whether the parameter is an in, out, or inout parameter.

parameterIDLDeclaration
The declaration of the parameter, including type, in IDL form.

parameterCDeclaration
The declaration of the parameter, including type, in C or C++ form. This may differ from the IDL declaration, particularly when the parameter is an out or inout parameter.

For SOMTPassthruEntryC Class

passthruLanguage
The target language of the passthru, in upper case. For example **.C**.

passthruTarget
The file type for this passthru; for example, **.h** or **.ih**

passthruBody
The full contents of the passthru entry, including newlines.

For SOMTSequenceEntryC Class

sequenceLength
The maximum length of the sequence, as declared in IDL, or zero if unspecified.

For SOMTStringEntryC Class

stringLength
The maximum length of the string, as declared in IDL, or zero if unspecified.

For SOMTTypedefEntryC Class

typedefDeclarators
The list of declarators.

typedefBaseType
The base type of the new user-defined type(s), not including pointer stars or array dimensions, if any.

The Section-Name symbols

The Emitter Framework recognizes a set of special symbols known as se*ction-name symbols*, which correspond to the various sections that can be emitted from an output template. The value of each section-name symbol is the name of a section to be emitted.

Each predefined section-emitting method in the Emitter Framework determines which section of the output template to emit based on the value of a predefined section-name symbol. For example, **somtEmitProlog** emits the section whose name is specified by the `prologSN` symbol. The default value of the `prologSN` symbol is `prologS`. Thus, **somtEmitProlog** emits the `prologS` section of the output template by default.

The value of a section-name symbol can be changed to cause the corresponding section-emitting method to emit a section of a different name. For example, to have the **somtEmitProlog** method emit a section named `myPrologS` rather than `prologS`, set the value of the `prologSN` symbol to `myPrologS`, using the **somtSetSymbolCopyValue** method as described in **Defining New Symbols** on page 385, before invoking **somtEmitProlog.**

The following table lists all symbol names, their initial value, and the method that uses them.

Symbol Name	Initial Value (Section Name)	Used by Method
attributeSN	attributeS	somtEmitAttribute
attributeEpilogSN	attributeEpilogS	somtEmitAttributeEpilog
attributePrologSN	attributePrologS	somtEmitAttributeProlog
baseSN	baseS	somtEmitBase
baseEpilogSN	baseEpilogS	somtEmitBaseEpilog
baseIncludesSN	baseIncludesS	somtEmitBaseIncludes
baseIncludesEpilogSN	baseIncludesEpilogS	somtEmitBaseIncludesEpilog
baseIncludesPrologSN	baseIncludesPrologs	somtEmitBaseIncludesProlog
basePrologSN	basePrologS	somtEmitBaseProlog
classSN	classS	somtEmitClass
constantSN	constantS	somtEmitConstant
constantPrologSN	constantPrologS	somtEmitConstantProlog
constantEpilogSN	constantEpilogS	somtEmitConstantEpilog
dataSN	dataS	somtEmitData
dataEpilogSN	dataEpilogS	somtEmitDataEpilog
dataPrologSN	dataPrologS	somtEmitDataProlog
enumSN	enumS	somtEmitEnum
enumEpilogSN	enumEpilogS	somtEmitEnumEpilog
enumPrologSN	enumPrologS	somtEmitEnumProlog
epilogSN	epilogS	somtEmitEpilog
inheritedMethodsSN	inheritedMethodsS	somtEmitMethod
interfaceSN	interfaceS	somtEmitInterface
interfaceEpilogSN	interfaceEpilogS	somtEmitInterfaceEpilog
interfacePrologSN	interfacePrologS	somtEmitInterfaceProlog
metaSN	metaS	somtEmitMeta
metaIncludeS	metaIncludeS	somtEmitMetaIncludes
methodsSN	methodsS	somtEmitMethod
methodsSN	methodsS	somtEmitMethods
methodsEpilogSN	methodsEpilogS	somtEmitMethodsEpilog
methodsPrologSN	methodsPrologS	somtEmitMethodsProlog
moduleSN	moduleS	somtEmitModule
moduleEpilogSN	moduleEpilogS	somtEmitModuleEpilog
modulePrologSN	modulePrologS	somtEmitModuleProlog
overrideMethodsSN	overrideMethodsS	somtEmitMethod
passthruSN	passthruS	somtEmitPassthru
passthruEpilogSN	passthruEpilogS	somtEmitPassthruEpilog
passthruPrologSN	passthruPrologS	somtEmitPassthruProlog
prologSN	prologS	somtEmitProlog
releaseSN	releaseS	somtEmitRelease
structSN	structS	somtEmitStruct
structEpilogSN	structEpilogS	somtEmitStructEpilog
structPrologSN	structPrologS	somtEmitStructProlog
typedefSN	typedefS	somtEmitTypedef
typedefEpilogSN	typedefEpilogS	somtEmitTypedefEpilog
typedefPrologSN	typedefPrologS	somtEmitTypedefProlog
unionEpilogSN	unionEpilogS	somtEmitUnionEpilog
unionSN	unionS	somtEmitUnion
unionPrologsN	unionPrologs	somtEmitUnionProlog

Appendix A. Error Codes

This appendix provides information about error codes generated from the SOMobjects Developer Toolkit. These error codes pertain to SOM and DSOM as well as Object Services. Informational-only messages (as opposed to error messages) are not documented.

Special Error Codes

The following error messages can be used by any Object Service:

00000 **UNDEFINED**
Explanation: This error code has no specific meaning. Any object service can use this error code if there is not a service-specific error code that would provide additional information. If an object service makes an error log entry to save data for service personnel, but none of the data is likely to be useful, you can choose this error code.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

00001 **EXCEPTION_MAPPED**
Explanation: An object service received a user-exception from a sub-service method call. When the receiving object service does not list this exception on its own **raises** keyword, it must either map the received exception into the user-exception that is listed on its **raises** keyword, or a system exception. The object service can then raise the new, mapped exception to its caller.
Programmer Response: Determine the original exception that was raised. Review the error log entries or messages for a corresponding message that contains the text . . . `Raised USER_EXCEPTION` . . . Once you locate this message, follow the action list in this appendix for the error code it contains. This error code is usually the best place to look to find information for resolving the problem.

SOM Kernel Error Codes

You might encounter the following error codes from the SOM kernel and the various frameworks of the SOMobjects Developers Toolkit while an application is running.

20011 **SOMERROR_CCNullClass**
Explanation: A null class argument is being passed to a **somDescendedFrom Method**.

20029 **SOMERROR_SompntOverflow**
Explanation: The internal buffer used in the **somPrintf Function** is overflowing.

20039 **SOMERROR_MethodNotFound**
Explanation: **somFindMethod(Ok) Methods** is failing to find the indicated method.

20049 **SOMERROR_StaticMethodTableOverflow**
Explanation: A method-table overflow is occurring in **somAddStaticMethod**.

20059 **SOMERROR_DefaultMethod**
Explanation: A defined method was not added before calling the **somDefaultMethod**.

20069 **SOMERROR_MissingMethod**
Explanation: The specified method is not defined on the target object.

20079 **SOMERROR_BadVersion**
Explanation: An attempt is being made to load, create, or use a version of a class-object implementation that is incompatible with the program.

20089 **SOMERROR_NullId**
Explanation: The **som_CheckId** is being given a NULL ID to check.

20099 **SOMERROR_OutOfMemory**
Explanation: Memory is exhausted.

20109 **SOMERROR_TestObjectFailure**
Explanation: The **somObjectTest** is finding problems with the object it is testing.

20119 **SOMERROR_FailedTest**
Explanation: The **somTest** is detecting a failure.

20121 **SOMERROR_ClassNotFound**
Explanation: The **somFindClass Method** cannot find the requested class.

20131 **SOMERROR_OldMethod**
Explanation: An old-style method name is being used.
Programmer Response: Change the method to the appropriate name.

20149 **SOMERROR_CouldNotStartup**
Explanation: The **somEnvironmentNew Function** is failing to complete.

20159 **SOMERROR_NotRegistered**
Explanation: The **somUnloadClassFile Method** argument is not a registered class.

20169 **SOMERROR_BadOverride**
Explanation: The **somOverrideSMethod** is being invoked for a method that is not defined in a parent class.

20179 **SOMERROR_NotImplementedYet**
Explanation: The method raising the error message is not implemented.

20189 **SOMERROR_MustOverride**
Explanation: The method raising the error message should be overridden.

20199 **SOMERROR_BadArgument**
Explanation: An argument to a core SOM method is failing a validity test.

20219 **SOMERROR_NoParentClass**
Explanation: While creating a class object, the parent class cannot be found.

20229 **SOMERROR_NoMetaClass**
Explanation: While creating a class object, the metaclass object cannot be found.

DSOM Error Codes

You might encounter the following error codes from DSOM while an application is running. Obsolete messages have been removed, so message numbers are not in sequential order.

30001 **SOMDERROR_NoMemory**
Explanation: DSOM run time is failing to allocate the necessary memory.
Programmer Response: Bring down the DSOM application and free system resources.

30002 **SOMDERROR_NotImplemented**
Explanation: The function or method is not yet supported.
Programmer Response: Do not call the function or method in the application.

30003 **SOMDERROR_InvalidProtocolInformation**
Explanation: Invalid protocol information is being specified in the configuration file.
Programmer Response: See **Chapter 2, Configuration and Startup** for a description of **SOMDPROTOCOLS** and stanza definitions for each protocol.

30004 **SOMDERROR_SOMDDAlreadyRunning**
Explanation: The DSOM daemon is already started. That is, an instance of **somdd** is already running.
Programmer Response: If the current instance of **somdd** is not responding properly, delete all instances of **somdd** and restart a new copy.

30006 **SOMDERROR_InvalidConfigSetting**
Explanation: The configuration variable setting is not valid.
Programmer Response: See **Chapter 2, Configuration and Startup** for a description of the environment variable and set it to a valid value. Use the **somdchk** command to display the current environment.

30007 **SOMDERROR_BadEnvironment**
Explanation: One of the following applies:

- An environment is being passed to a DSOM function or method and the major field is not set to NO_EXCEPTION.
- An invalid environment structure is being returned for a user exception.

Programmer Response: If the first scenario in the Explanation applies, verify that a valid environment is being passed to DSOM. A valid environment has the major field set to NO_EXCEPTION. If the second scenario in the Explanation applies, use **somSetException** to set the exception value in the Environment structure.

30008 **SOMDERROR_HostAddress**
Explanation: DSOM run time is encountering an invalid host address.
Programmer Response: Verify the network setup. Verify that the network database that associates host names and addresses contains the correct entries for client and server machines.

30009 **SOMDERROR_CouldNotStartProcess**
Explanation: The DSOM daemon cannot start a new process to execute the server program. Server programs are specified in the Implementation Repository.
Programmer Response: Ensure that a fully-qualified path name is specified or that the server program resides in the directory specified in the path.

30015 **SOMDERROR_BadTypeCode**
Explanation: An invalid type code is being encountered while processing the request.
Programmer Response: Use a valid type codes.

30016 **SOMDERROR_BadDescriptor**
Explanation: The DSOM run time is encountering bad or missing information in the Interface Repository.
Programmer Response: Follow these steps:

1. Run the IR emitter to update the Interface Repository with the specified identifier.
2. Use the **irdump** utility to verify that the Interface Repository has been correctly updated.

30017 **SOMDERROR_InvalidBaseProxyClass**
Explanation: An invalid class is specified for a user-defined proxy.
Programmer Response: Modify the user-defined proxy so that it is derived from **SOMDClientProxy**.

30018 **SOMDERROR_CouldNotStartThread**
Explanation: The DSOM run time cannot create a thread.
Programmer Response: Bring down the DSOM application and free system resources.

30019 **SOMDERROR_NoMessages**
Explanation: **SOMOA::execute_next_request**, **Request::get_response**, or **Request::get_next_response** is calling, indicating not to wait but no requests or responses are pending. This might not be an error.
Programmer Response: Ignore the exception or change the method invocation to request a block until the message is available.

30020 **SOMDERROR_UndeclaredException**
Explanation: The application is raising an exception that is not specified in the method definition.
Programmer Response: Modify the method IDL to contain a raises expression for the exception.

30021 **SOMDERROR_Marshaling Error**
Explanation: The data passed to the DSOM marshaler is not valid or did not match the expected data type.
Programmer Response: Follow these steps:

1. Ensure that the data passed to the DSOM marshaler (for example, the parameters passed to a remote method invocation or the results returned from a remote invocation) are correctly initialized.
2. Ensure that all data structures are constructed according to the data type definitions in IDL for the method being invoked.

30022 **SOMDERROR_ServerInterrupt**
Explanation: The **SOMOA::interrupt_server** method is being invoked on the **SOMOA** object.
Programmer Response: Do not invoke **interrupt_server** on the **SOMOA** object.

30023 **SOMDERROR_CommTimeOut**
Explanation: Client and server processes are not communicating properly, which can happen if the **somdd** processes are not started on all participating server machines prior to starting the application processes.
Programmer Response: Follow these steps:

1. Check the network setup and verify that your networking software is running.
2. Increase the values of **SOMDRECVWAIT** and **SOMDSENDWAIT**.

30024 **SOMDERROR_CannotConnect**
Explanation: There is no protocol over which the client can connect to the server.
Programmer Response: Ensure that the **SOMDPROTOCOLS** setting of the client has at least one entry in common with the **SOMDPROTOCOLS** setting under which the server was registered. Use the **regimpl** command to view the **HOSTNAME** setting for the server.

30025 **SOMDERROR_BadConnection**
Explanation: A previous connection with a server is being terminated because of a communications error. The request might not be successful.
Programmer Response: Check the network setup and verify that your networking software is running.

30034 **SOMDERROR_BadObjref**
Explanation: The function or method is calling on an invalid object reference.
Programmer Response: Use or generate another reference to the target object.

30043 **SOMDERROR_NoSOMDInit**
Explanation: The application is attempting to create or access remote objects before DSOM initializes.
Programmer Response: Modify the application to call function **SOMD_Init** before making any run-time calls.

30044 **SOMDERROR_CommunicationsError**
Explanation: A communications error is occurring, which can happen if the **somdd** processes are not started on all participating server machines prior to starting the application processes.
Programmer Response: Check the network setup and verify that your networking software is running.

30045 **SOMDERROR_ImplRepIO**
Explanation: An error is occurring while accessing Implementation Repository files. The files might be corrupted. This might indicate that the Implementation Repository files cannot be found or cannot be accessed.
Programmer Response: Follow these steps:

1. Verify that the **SOMDDIR** environment variable is set to a directory that grants read and write permissions to the DSOM user. (It is best if the directory name is fully qualified.)
2. If the **SOMDDIR** environment variable is not set, verify that the default directory, ($SOMBASE/etc/dsom on AIX, and `%SOMBASE%\etc\dsom,` on OS/2 or Windows NT is set with the correct permissions.
3. Ensure that the files contained in the directory all have read and write permissions granted to the DSOM user.
4. If the preceding steps did not work, restore a backup version of the files, if available. If a backup version of the files is not available, rebuild the Implementation Repository.

30046 **SOMDERROR_EntryNotFound**
Explanation: The requested entry in the Implementation Repository cannot be found.
Programmer Response: Use the **regimpl** command to examine the contents. Reissue the command with the correct implementation ID, alias, or class.

30047 **SOMDERROR_ClassNotFound**
Explanation: One of the following applies:

- DSOM run time is failing to load the specified class, which can occur if the class name specified in calls to **somdCreate** or **find_any** is not associated with any server that is registered.
- The class libraries (DLLs) used to build the proxy class are statically linked to the program, but the DLL's **SOMInitModule** function is not properly initializing the class object, or has no **SOMInitModule**.
- A process cannot load the DLL associated with a particular class.

Programmer Response: Follow these steps:

1. Ensure that the class name is associated with at least one of the server implementations.
2. Ensure that the DLL resides in the directory specified in **LIBPATH** or **PATH** and that the class has an entry in the Interface Repository.
3. Verify that the DLL contains the **SOMInitModule** initialization function.

4. Ensure that the IDL for the class contains the **dllname** modifier, that this IDL has been compiled into the Interface Repository, and that the DLL name given by the **dllname** modifier can be loaded. (Use the **irdump** utility to determine whether a particular class appears in the IR.)

30048 **SOMDERROR_ServerNotFound**
Explanation: The input server alias cannot be found, or the **SOMDObjectMgr** object is raising an exception in response to a **somdFindServer**, **somdFindServerByName**, **somdFindServerByClass**, or **somdFindAnyServerByClass** invocation to indicate that the requested server cannot be found.
Programmer Response: Follow these steps:

1. Use the **regimpl** command to ensure that the requested server has been registered with the appropriate classes (in the case of **somdFindServerByClass** or **somdFindAnyServerByClass**).
2. Ensure that the application receiving the error can successfully contact the Naming Server into which this information is registered.

30049 **SOMDERROR_ServerAlreadyExists**
Explanation: A server process that is running has already registered itself with the DSOM daemon (**somdd**) using the implementation ID of the desired server program.
Programmer Response: If another instance of the server is not actually running, restart the DSOM daemon. (This exception can occur if a user-written server program terminates without notifying the DSOM daemon via a call to **deactivate_impl**).

30061 **SOMDERROR_CtxNoPropFound**
Explanation: The property being passed to **get_values** or **delete_values** is not in the **Context** object.
Programmer Response: Modify the application to pass a valid property name.

30066 **SOMDERROR_BadParm**
Explanation: An invalid parameter value is being passed to a function or method.
Programmer Response: See *SOMobjects Developer Toolkit Programmer's Reference* for a description of the parameters to the specified function or method.

30070 **SOMDERROR_AuthnFail**
Explanation: DSOM cannot initialize the security run time in a client or server.
Programmer Response: If you do not want to use any secure DSOM servers, disable authentication by setting `LOGIN_INFO_SOURCE=` (with no setting specified) in the [somsec] stanza of the **somenv.ini** file and restart your application. If you want to communicate with secure DSOM servers, ensure that the Naming Server is running correctly and that you have successfully configured SOMobjects on your machine. Restart your application.

30072 **SOMDERROR_SecurityFail**
Explanation: The security initialization is failing.
Programmer Response: Ensure that the Security Server is running correctly and that you are logged in. If the Security Server is running on an OS/2 platform, ensure that LAN Server 4.0 is configured properly. Also, make sure the client has been registered in the user registry for the platform where the Security Server resides.

30080 **SOMDERROR_DuplicateEntry**
Explanation: DSOM cannot update the Implementation Repository. If you are attempting to add a new implementation definition, an alias already exists. If you are attempting to add a new class to an existing entry, a class is already associated with entry.
Programmer Response: Reissue the command with a new alias or class name.

30081 **SOMDERROR_Internal**
Explanation: An internal DSOM error is occurring.
Programmer Response: Retry the scenario. If the problem persists, gather information about the problem and follow your local procedures for resolving problems.

30082 **SOMDERROR_BadUnionTag**
Explanation: The union discriminant value does not match any of the defined cases and no default case is defined.
Programmer Response: See **Union Type** on page 121 in *SOMobjects Developer Toolkit Programmer's Guide* for additional information on the union IDL type.

30083 **SOMDERROR_BadSequence**
Explanation: An invalid sequence is being found while marshalling.
Programmer Response: See **Template Types (Sequences and Strings)** on page 122 in *SOMobjects Developer Toolkit Programmer's Guide* for a description of the sequence IDL type. (It is an error for _length to be greater than _maximum. For bound sequences, it is an error to set _length or _maximum to be larger than the specified bound.)

30084 **SOMDERROR_NotStreamable**
Explanation: DSOM run time cannot marshal object as **pass_by_copy**. Processing will continue. An object reference is being sent instead.
Programmer Response: Verify that the **pass_by_copy** object is derived from **CosStream::Streamble** or that it is local (not a proxy).

30085 **SOMDERROR_BadForeign**
Explanation: A static foreign type is being produced through run-time conversion. Static foreign types should be used only in IDL.
Programmer Response: Do not use an **any** type that contains a static foreign type.

30086 **SOMDERROR_NotForeignMarshaler**
Explanation: An invalid class is specified for the dynamic foreign type.
Programmer Response: Derive the class from **SOMDDataMarshaler::ForeignMarshaler**.

30088 **SOMDERROR_NamingNotActive**
Explanation: The Naming Service is not active. Entries can be added to the Implementation Repository without updating the Naming Service. However, entries with information in both the Implementation Repository and the Naming Service must be kept consistent. Such entries cannot be updated or deleted if the Naming Service is not active.
Programmer Response: Verify that **som_cfg** has been run by checking for **INSTALL** and **SOMNMOBJREF** entries in the [somnm] stanza of the configuration file.

30089 **SOMDERROR_WrongRefType**
Explanation: The function or method is being invoked on an incompatible object reference. For example, **SOMOA::get_id** cannot be invoked on an object reference that is NULL, generated by the **create_SOM_ref** method, or a proxy.
Programmer Response: Modify the application to pass a compatible object reference.

30090 **SOMDERROR_AbstractClass**
Explanation: The method is being invoked on an abstract class.
Programmer Response: Modify the application code to invoke the method on a subclass.

30109 **SOMDERROR_SOMDDNotRunning**
Explanation: One of the following applies:

- The DSOM daemon is not started.
- The DSOM daemon is running, but is not using the same:
  - SOMDPROTOCOLS setting
  - HOSTNAME setting

- SOMDPORT setting

that was in use by the server that created the object reference that the client application or protocols are trying to use.

- You are using IPC and using an invalid transient object to establish a connection with the server.

Programmer Response:

- Ensure that the DSOM daemon, **somdd**, is running on the server machine. The daemon prints a message when it is ready to accept incoming requests.
- Ensure that the SOMDPROTOCOLS, HOSTNAME, and SOMDPORT settings in the **somenv.ini** file used by the DSOM daemon are consistent with those used by the server that created the object reference that the client is trying to use. Be aware that each SOMDPROTOCOLS setting has a corresponding stanza in the **somenv.ini** file, each of which can have its own HOSTNAME and SOMDPORT settings.
- If you are using IPC and invoking on a transient object reference, verify that the instance of the server that generated the reference is still active. Consider using a persistent object reference.

35*xxx* **SOMDERROR_SOCKET**
Explanation: A socket error is occurring with a return code equal to xxx.
Programmer Response: Consult the technical reference for the implementor of the socket protocol in use.

4*xxxx* **SOMDERROR_Operating System**
Explanation: An operating system error is occurring with a return code equal to xxxx.
Programmer Response: Consult the technical reference for the operating system being used.

Externalization Service Error Codes

You might encounter the following error codes from the Externalization Service while an application is running.

54000 **somStream_GENERAL**
Explanation: The method is raising a standard exception.
Programmer Response: Refer to the name of the standard exception to determine the cause of the problem. Gather information about the problem and follow your local procedures for resolving problems.

54001 **somStream_SEMAPHORE_CREATE**
Explanation: An error is occurring while creating a semaphore.
Programmer Response: Follow these steps:

1. Ensure that your system is not exceeding the maximum number of semaphores it is allowed to create.
2. Ensure that your system has threading support.

54002 **somStream_SEMAPHORE_REQUEST**
Explanation: An error is occurring while requesting a semaphore.
Programmer Response: Follows these steps:

1. Check the global environment after you create each object to ensure that it creates and initializes properly.
2. Ensure that your system has threading support
3. Check for other threads in the process that might have acquired the semaphore and not released it.

54003 **somStream_SEMAPHORE_RELEASE**
Explanation: An error is occurring while releasing a semaphore.
Programmer Response: Review the error log for any previous errors indicating a failure when requesting a semaphore.

54012 **somStream_METHOD_IS_ABSTRACT**
Explanation: The method is not implemented.
Programmer Response: Do not instantiate objects on abstract classes.

54018 **somStream_ALREADY_STREAMED_PARMS**
Explanation: Parameters being passed to **already_streamed** are not valid.
Programmer Response: Ensure that you pass to the **already_streamed** method:

- a pointer to an object, and
- a pointer to a class object that is a parent of the object.

54019 **somStream_BAD_BUFFER_PARAMETER**
Explanation: The parameter to **set_buffer** has an invalid length or the header is incorrect.
Programmer Response: Ensure that the buffer you are attempting to set was produced by the **_get_buffer** method of the same type of **StreamIO**. Ensure that all of the fields of the octet sequence are set correctly.

54020 **somStream_ICONV_FAILURE**
Explanation: An error is occurring while converting the code page of the stream data.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

54021 **somStream_READ_PASSED_END_OF_STREAM**
Explanation: You are attempting to read past the end of the data in the **StreamIO**.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream. To read the stream twice, you must use the **reset** method.

54026 **somStream_UNABLE_TO_READ_SHORT**
Explanation: The **read_short** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54027 **somStream_UNABLE_TO_READ_LONG**
Explanation: The **read_long** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54028 **somStream_UNABLE_TO_READ_USHORT**
Explanation: The **read_unsigned_short** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54029 **somStream_UNABLE_TO_READ_ULONG**
Explanation: The **read_unsigned_long** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54030 **somStream_UNABLE_TO_READ_STRING**
Explanation: The **read_string** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54031 **somStream_UNABLE_TO_READ_CHAR**
Explanation: The **read_char** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54032 **somStream_UNABLE_TO_READ_FLOAT**
Explanation: The **read_float** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54033 **somStream_UNABLE_TO_READ_DOUBLE**
Explanation: The **read_double** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54034 **somStream_UNABLE_TO_READ_OCTET**
Explanation: The **read_octet** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54035 **somStream_UNABLE_TO_READ_BOOLEAN**
Explanation: The **read_boolean** method is failing.
Programmer Response: Ensure that you are reading the same number and same type of items from the stream as were written to the stream.

54037 **somStream_READ_FROM_EMPTY_STREAM**
Explanation: You are trying to read data from a **StreamIO** that is empty.
Programmer Response: You must put data into the stream, by either writing to it or using the **set_buffer** method, before you attempt to read from it.

Naming Service Error Codes

You might encounter the following error codes from the Naming Service while an application is running.

55001 **SOMNM_AbstractClass**
Explanation: The specified class is defined as an abstract class. No implementation is provided for this class. You cannot invoke methods on an abstract class.
Programmer Response: Use the concrete class shipped with the product.

55002 **SOMNM_RandomFailed**
Explanation: The file name generator is failing to generate a unique file name to be used by the Naming Service, which can happen if the directory pointed to by the **SOMDDIR** environment variable contains a large number of files (that is, the Naming Service has approached its limits).
Programmer Response: Try deleting unnecessary naming contexts. If the problem persists, gather information about the problem and follow your local procedures for resolving problems.

55003 **SOMNM_DestroyFailed**
Explanation: An error is occurring while destroying a naming context.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

55004 **SOMNM_ImplLimit**
Explanation: The implementation is reaching its internal limits.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

55101 **SOMNM_DatabaseError**
Explanation: An error is occurring while accessing a naming database. The **SOMDDIR** environment variable in the **somenv.ini** file specifies the location of these databases.
Programmer Response: Ensure that the drive, path, and directory where the naming database files are located are valid.

55102 **SOMNM_DataBaseOpenError**
Explanation: An error is occurring while opening a naming database. The **SOMDDIR** environment variable in the **somenv.ini** file specifies the location of these databases
Programmer Response: Ensure that the drive, path, and directory where the naming database files are located are valid.

55103 **SOMNM_TypecodeError**
Explanation: An error is occurring while manipulating a property value.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

55104 **SOMNM_DatabaseCreateError**
Explanation: An error is occurring while creating a naming database. The **SOMDDIR** environment variable in the **somenv.ini file** specifies the location of these databases.
Programmer Response: Ensure that the drive, path, and directory where the naming database files are located are valid. Ensure that the directory permissions allow files to be created.

55201 **SOMNM_FilterInternalError**
Explanation: An error is occurring in the Naming Service while processing a search request.
Programmer Response: Try restarting the Naming Server. If that does not work, gather information about the problem and follow your local procedures for resolving problems.

55202 **SOMNM_ConstraintError**
Explanation: An error is occurring in one of the **find** methods while processing the constraint.
Programmer Response: Ensure that the constraint string specified in the search satisfies the grammar.

55203 **SOMNM_ConstraintTooLong**
Explanation: The constraint that is specified for the **find** method is exceeding internal limits.
Programmer Response: Invoke the **find** method using a smaller constraint. Try splitting the query into sub-queries.

55301 **SOMNM_ANYInternalError**
Explanation: An error is occurring while trying to use the **any** data type.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

55401 **SOMNM_NoComponent**
Explanation: The method is attempting to operate on a non-existent name component, which can happen if you try to access a name component that is not defined in the **LName** object.
Programmer Response: Ensure that the specified component was inserted using the **insert_component** method.

55402 **SOMNM_OverFlow**
Explanation: The method is attempting to insert a name component into a **LName** object and is exceeding the maximum number of components in an **LName** object. The maximum number of components allowed is 16.
Programmer Response: Reduce the number of name components in the **LName** object.

55404 **SOMNM_NotSet**
Explanation: The method is attempting to retrieve the id string or the kind string within a name component. The id string or the kind string might not be set, which might be acceptable.
Programmer Response: Do nothing if the calling client is using the Naming Service in such a way that the id string or kind string does not need to be set; otherwise, check your application logic and retry the operation.

55405 **SOMNM_NotFound**
Explanation: The Naming Service cannot resolve the name, which can happen if the name was never bound to the naming context or if the name was unbound from the context.
Programmer Response: If the name is a compound name, ensure that all the sub-contexts in the compound name are accessible from the **NamingContext** object on which the resolve operation was performed.

55407 **SOMNM_AlreadyBound**
Explanation: The **bind** method (or its variations) is attempting to associate an object to a name that was previously bound in the same context.
Programmer Response: Use the **rebind** method (or its variations) to replace an existing binding or use the **unbind** method followed by the **bind** method to achieve the same result.

55408 **SOMNM_NotEmpty**
Explanation: A **destroy** method is being invoked on a naming context that has bindings. All bindings in the naming context must to be removed before the naming context can be destroyed.
Programmer Response: Use the **unbind** method to unbind objects.

55412 **SOMNM_PropertyNotFound**
Explanation: A method call to get the property cannot find the specified property. If the method was a batch operation such as **get_properties**, then the last property in the list that cannot be found results in this exception.
Programmer Response: Do nothing if the calling client is using the Naming Service in such a way that the specified property does not need to be set; otherwise, check your application logic and retry the operation.

55415 **SOMNM_IllegalConstraint**
Explanation: The constraint specified for the **find** methods is not syntactically valid. The BNF grammar for the search constraint is described in **Appendix A, BNF for Naming Constraint Language** in *Programmer's Reference for Abstract Interface Definitions*.
Programmer Response: Correct the syntax of the constraint.

55416 **SOMNM_BindingNotFound**
Explanation: One of the **find** methods cannot find a binding that satisfies the specified search constraint. This might be valid response that requires no further action.
Programmer Response: Ensure that the *distance* parameter that was specified for the search is valid and reasonable.

Security Service Error Codes

You might encounter the following error codes from the Security Service while an application is running.

56001 **SOMSEC_ERRCODE_AuthnFail**
Explanation: The principal making a remote method invocation is failing authentication.
Programmer Response: Follow these steps:

1. Ensure that you are logged on to LAN Services. Alternatively, you can log off LAN Server and run your application without authentication, bearing in mind that a secure server will reject unauthenticated method invocations.
2. Restart your application.

56006 **SOMSEC_ERRCODE_NoRegistry**
Explanation: The Security Server cannot create or open its database. Only one Security Server can be running at a time.
Programmer Response: Ensure that a Security Server is not already running. Also, ensure that you have set the **SOMDDIR** environment variable in your **somenv.ini** file to a directory in which you have write access.

56007 **SOMSEC_ERRCODE_BadRegistry**
Explanation: An error is occurring while opening the Registry database.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

56009 **SOMSEC_ERRCODE_ObjectNotFound**
Explanation: Initialization of Security Services is failing because needed resources cannot be obtained from the Naming Services.
Programmer Response: Ensure that **SOMobjects** is properly configured.

56010 **SOMSEC_ERRCODE_AuthnRequired**
Explanation: The application is making an unauthenticated remote method request to a secure server.
Programmer Response: Log on to LAN Services and restart your application. Alternatively, restart the application server in a non-secure mode using the **regimpl** command.

56011 **SOMSEC_ERRCODE_InvalidSessId**
Explanation: An error is occurring in **SOMobjects**.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

56013 **SOMSEC_ERRCODE_CannotLoadDll**
Explanation: A LAN Services dynamic link library (DLL) cannot be loaded and used by Security Services. The name of the DLL in question is logged in the message that accompanies this error. The Security Services cannot perform authentication without this DLL. Your application will continue running, unauthenticated.
Programmer Response: If your application requires authentication, as the case will be if it needs services from a secure server, you need to stop your application and reinstall LAN Services on your machine before restarting your application.

56014 **SOMSEC_ERRCODE_FunctionMissing**
Explanation: A LAN Services dynamic link library (DLL) is available but is not the appropriate DLL for authentication purposes. Your application will continue running, unauthenticated.
Programmer Response: Upgrade or reinstall LAN Services if you need to use the authentication facility provided by the Security Services.

56015 **SOMSEC_ERRCODE_NoLoginContext**
Explanation: You did not log on to LAN Services before starting your application. Your application will continue running, unauthenticated.
Programmer Response: If you need to use the authentication facility provided by the Security Services, stop the application, log on, and restart the application.

56017 **SOMSEC_ERRCODE_UnknownError**
Explanation: A subsystem used by the Security Service is giving an unrecognized return code.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

56018 **SOMSEC_ERRCODE_IOError**
Explanation: The Security Server is encountering an error while using its database. There can be only one Security Server running at a time.
Programmer Response: Check that a Security Server is not already running. Also, ensure that you have set the **SOMDDIR** environment variable in your **somenv.ini** file to a directory in which you have write access.

56019 **SOMSEC_ERRCODE_NotSecure**
Explanation: The Security Service is attempting to authenticate your method requests to a non-secure server.
Programmer Response: Do nothing if your application expects the server to be non-secure. Otherwise, stop the server and restart it in a secure mode. Restart your application.

56020 **SOMSEC_ERRCODE_BadSequence**
Explanation: An error is occurring in the Security Service.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

56021 **SOMSEC_ERRCODE_UnrecognizedMsgNr**
Explanation: An error is occurring in the Security Service.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

56022 **SOMSEC_ERRCODE_UnrecognizedMsgFormat**
Explanation: An error is occurring in the Security Service.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

Object Services (OS) Server Error Codes

You might encounter the following error codes from the Object Services Server while an application is running.

57000 **SOMOS_Server_Already_Running**
Explanation: The initialization of the persistence database is failing because the OS Server is already running. Only one instance of the OS Server associated with a particular implementation ID or implementation alias can be running in a system at any given time.
Programmer Response: Either start the OS Server using a different implementation alias or terminate the existing OS Server.

57001 **SOMOS_Database_Directory_Not_Accessible**
Explanation: The OS Server cannot access the directory specified by the **SOMDDIR** environment variable in the **somenv.ini** file.
Programmer Response: Verify that the path, drive, and directory are valid and accessible and restart the OS Server process again.

57002 **SOMOS_Unable_To_Open_Master_Database**
Explanation: The OS Server cannot open or create the master database **somosdb.dat**. This database contains the references to the metadata database and the attribute persistence database for each implementation of an OS Server. The **SOMDDIR**

environment variable in the **somenv.ini** file specifies where the master database is located on your system.
Programmer Response: Ensure that the database drive, path, and directory are available and accessible.

57003 **SOMOS_Unable_To_Open_Metadata_Database**
Explanation: The OS Server cannot access the metadata database. This database contains the persistence object information for the OS Server. The **SOMDDIR** environment variable in the **somenv.ini** file specifies where the metadata database is located.
Programmer Response: Ensure that the database drive, path, and directory are available and accessible.

57004 **SOMOS_Unable_To_Open_Attribute_Persist_Database**
Explanation: The OS Server cannot access the attribute persistence database. The **SOMDDIR** environment variable in the **somenv.ini** file specifies where the attribute persistence database is located.
Programmer Response: Ensure that the database drive, path, and directory are available and accessible.

57005 **SOMOS_Unable_To_Find_Attribute_Persist_Database**
Explanation: The OS Server cannot initialize because the attribute persists database cannot be found. The **SOMDDIR** environment variable in the **somenv.ini** file specifies where the attribute persistence database is located.
Programmer Response: Ensure that the database drive, path, and directory are available and accessible.

57006 **SOMOS_Unable_To_Find_Metadata_Database**
Explanation: The OS Server cannot initialize because the metadata database cannot be found. The **SOMDDIR** environment variable in the **somenv.ini** file specifies where the metadata database is located.
Programmer Response: Ensure that the database drive, path, and directory are available and accessible.

57007 **SOMOS_Usage_Error**
Explanation: An invalid command is being entered to start the OS Server. The following parameters are valid: **somossvr** [-**i**] [-**d**] -**a** [**impl_alias** | **impl_uuid**], where -**i** is to initialize the OS Server the first time it is started, -**d** is for debugging purposes, and -**a** is to specify the implementation alias for the OS Server.
Programmer Response: Specify the correct parameters to start the OS Server.

57008 **SOMOS_Unable_To_Find_Implementation_Definition**
Explanation: The OS Server cannot find the implementation definition as it is specified to the OS Server.
Programmer Response: Correct the implementation definition as it is specified to the **somossvr** program. If you start the OS Server manually, ensure that the -**a** option specifies a valid implementation alias or that the implementation ID is valid, which is specified without any parameters.

57009 **SOMOS_Impl_Is_Ready_Failed**
Explanation: The **_impl_is_ready()** call is failing within the OS Server.
Programmer Response: See **DSOM Error Codes** for this function for a further description of possible error conditions.

57010 **SOMOS_SOMD_Init_Failed**
Explanation: The **SOMD_Init()** function is failing within the OS Server.
Programmer Response: See **DSOM Error Codes** for this function for a further description of possible error conditions.

57012 **SOMOS_Server_Exit_Abnormal.**
Explanation: The OS Server process is exiting abnormally, which can be caused by an unrecoverable error.
Programmer Response: Check previous error log messages for this server process for an explanation of the abnormal exit. Correct the problem and restart the OS Server.

57100 **SOMOS_Class_Name_Error**
Explanation: A method call to **make_persistent_ref** cannot find the class name specified.
Programmer Response: Pass a valid class name to this method and retry the operation.

57101 **SOMOS_UUID_Create_Error**
Explanation: The **make_persistent_ref** method cannot create a UUID.
Programmer Response: See **SOM Kernel Error Codes** for a further description on why the call **somCreateUUID()** failed and for possible solutions.

57102 **SOMOS_Not_In_Cache**
Explanation: An attempt to find an object in the internal cache is unsuccessful.
Programmer Response: Ensure that the item is in the persistence database, which makes use of the internal cache. The object reference might not be persistent, if that is the case, ensure that a persistence reference is created for the object with a call to **make_persistent_ref()**.

57103 **SOMOS_Add_To_Cache_Error**
Explanation: The **make_persistent_ref** method is attempting to add the object to the internal cache but the object already exists in the cache. This method has already been performed on the object.
Programmer Response: Do not use the **make_persistent_ref** method on the specified object. Correct your program and retry the operation.

57104 **SOMOS_Add_Object_To_Database_Error**
Explanation: The **make_persistent_ref** method is attempting to add the object to the persistence database but is encountering an error. The **SOMDDIR** environment variable in the **somenv.ini** file specifies where the databases are located.
Programmer Response: Ensure that the drive, path, and directory containing the persistence database are valid and accessible.

57105 **SOMOS_Database_Error**
Explanation: The **delete_ref** method cannot delete the object reference from the persistence database. The environment variable **SOMDDIR** in the **somenv.ini** file specifies where the persistence database is located.
Programmer Response: Ensure that the persistence database is still accessible.

57106 **SOMOS_Add_Item_To_Database_Error**
Explanation: The method is encountering an error while saving metadata to the persistence database. The persistence database might not be available and accessible.
Programmer Response: Ensure that the drive, path, and directory containing the persistence databases are valid and accessible.

57107 **SOMOS_Database_Synchronization_Error**
Explanation: The method is encountering an error while synchronizing the persistence database. The persistence database might not be available and accessible.
Programmer Response: Ensure that the drive, path, and directory containing the persistence databases are valid and accessible.

57108 **SOMOS_Get_Item_From_Database_Error.**
Explanation: The method is encountering an error while retrieving an item from the persistence database. The persistence database might not be available and accessible.
Programmer Response: Ensure that the drive, path, and directory containing the persistence databases are valid and accessible.

57109 **SOMOS_Bad_Reference_Data.**
Explanation: The reference data is not from a **somOS::Server** class, which can happen if the **SOMObjFromRef** method is being used on the **somOS::Server**.
Programmer Response: Correct the reference data and try again.

57111 **SOMOS_Get_Object_From_Database_Error**
Explanation: The **somOS::Server** class cannot retrieve an object from the persistence database. This error can occur in the **SOMObjFromRef** method when the object reference is not found in the persistence database.
Programmer Response: Ensure that the object has a persistent reference by using the **make_persist_ref** method.

57112 **SOMOS_Passivate_Object_Not_Found**
Explanation: The **passivate_all_object** method is being called and not all objects can be found. Only one error message will be logged even though several objects might not be found during this method call.
Programmer Response: Ensure that the object reference is valid and that the specified object can be passivated.

57113 **SOMOS_Internal_Cache_Error.**
Explanation: The method call is encountering an internal cache problem while attempting to find an item in the cache.
Programmer Response: Use the **make_persistent_ref** method call to ensure that the object has a persistent reference.

57114 **SOMOS_Database_Initializing_Error**
Explanation: The **somOS::Server** class is encountering a problem while initializing the persistent database. This problem can occur if the persistence database directory is not valid and accessible.
Programmer Response: Ensure that the **SOMDDIR** environment variable in the **somenv.ini** file that specifies the location of the persistence databases is valid and accessible.

57115 **SOMOS_Database_Open_Error**
Explanation: The **somOS::Server** class is encountering a problem opening the internal persistence databases while initializing the class. This problem can occur if the persistence database directory is not valid or accessible.
Programmer Response: Ensure that the the **SOMDDIR** environment variable in the **somenv.ini** file that specifies the location of the persistence databases is valid and accessible.

57116 **SOMOS_Cache_Setup_Error**
Explanation: The **somOS::Server** class is encountering an error during initialization while setting up the internal cache.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57200 **SOMOS_Locking_Failure**
Explanation: The method is attempting to create a semaphore lock and is failing.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57201 **SOMOS_Cache_Error**
Explanation: The method is attempting to initialize the internal cache and is failing.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57202 **SOMOS_Database_Error**
Explanation: The method is attempting to create the persistence database and is failing.
Programmer Response: Follow these steps:

1. Ensure that the drive, path, and directory are valid and accessible where the persistence database is located.
2. Ensure that the **SOMDDIR** environment variable in the **somenv.ini** file that specifies where the persistence databases are located is valid and accessible.

57203 **SOMOS_Not_In_Database_Error**
Explanation: The specified attribute cannot be found in the database.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57204 **SOMOS_Not_In_IR_Error**
Explanation: The attribute definition cannot be found in the Implementation Repository.
Programmer Response: Ensure that the Implementation Repository path specified by the **SOMIR** environment variable is still valid and accessible and that it contains the required Implementation Repositories for your environment.

57205 **SOMOS_IR_Access_Error**
Explanation: An error is occurring while accessing the Implementation Repository to retrieve an attribute type.
Programmer Response: Ensure that the Implementation Repository path specified by the **SOMIR** environment variable is still valid and accessible and that it contains the required Implementation Repositories for your environment.

57206 **SOMOS_Decode_Error**
Explanation: An error is occurring while decoding an attribute.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57207 **SOMOS_Encode_Error**
Explanation: An error is occurring while encoding an attribute.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57208 **SOMOS_Parent_Error**
Explanation: An error is occurring while calling the parent class of the object.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57209 **SOMOS_Persist_Manager_Error**
Explanation: An error is occurring while initializing the Persistence Manager class.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57210 **SOMOS_No_Persist_Manager_Error**
Explanation: An error is occurring while accessing the Persistence Manager class.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57211 **SOMOS_OS_Server_Error**
Explanation: An error is occurring while calling the **somOS::Server** class.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57212 **SOMOS_Mutex_Initialization_Error**
Explanation: An error is occurring while initializing a semaphore.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57213 **SOMOS_Persistent_Reference_Error**
Explanation: An error is occurring while creating or deleting a persistence reference.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57214 **SOMOS_No_Database_Object_Error**
Explanation: The Persistence Manager class cannot create a database object.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57215 **SOMOS_No_Cache_Object_Error**
Explanation: The Persistence Manager class cannot create a cache object.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57216 **SOMOS_Persist_Manager_Already_Initialized_Error.**
Explanation: An error is occurring while initializing the Persistence Manager because it is already initialized.
Programmer Response: Gather information about the problem and follow your local procedures for resolving problems.

57217 **SOMOS_Persist_Manager_Create_Error**
Explanation: The Persistence Manager is failing to create a new process.
Programmer Response: Ensure that you have enough resources available in your system.

Metaclass Framework Error Codes

You might encounter the following messages from the Metaclass Framework while an application is running.

60019 **Explanation:** An attempt is being made to construct a class with **SOMMSingleInstance Metaclass** as a metaclass constraint, which can occur indirectly because of the construction of a derived metaclass. The initialization of the class is failing because **somInitMIClass** defined by **SOMMSingleInstance** is in conflict with another metaclass that is overriding **somNew** Method. That is, some other metaclass is already claiming the right to return the value for **somNew**.

60029 **Explanation:** An attempt is being made to construct a class with **SOMMSingleInstance** as a metaclass constraint. (This might occur indirectly because of the construction of a derived metaclass). The initialization of the class is failing because **somInitMIClass** defined by **SOMMSingleInstance** is in conflict with another metaclass that is overriding **somFree Method**. That is, some other metaclass is already claiming the right to override **somFree**.

60069 **Explanation:** A **SOMMBeforeAfter** metaclass must override both the **sommBeforeMethod** and **sommAfterMethod**. An attempt is being made to create a **SOMMBeforeAfter** metaclass where only one of the preceding methods is overridden.

60079 **Explanation:** An attempt is being made to subclass a non-subclassable metaclass. Only the SOM kernel is allowed to create subclasses of non-subclassable metaclasses when building a derived metaclass.

60089 **Explanation:** An attempt is being made to create a proxy class with more than two parents. A proxy class can have either two parents (a proxy class and the class of a target object) or one parent (a proxy class: in this case, a special proxy class is being created).

60099 **Explanation:** An attempt is being made to create a proxy class for which the first parent was not a descendant of **SOMMProxyForObject Class**. The first parent of all proxy classes must be a descendant of **SOMMProxyForObject**.

Appendix B. Converting OIDL Files to IDL

This appendix describes how to convert OIDL class descriptions in **.csc** files to IDL class descriptions in **.idl** files.

The conversion process involves two steps:

- Converting **.csc** files to .idl files. This step is largely automatic, and most classes can be converted without intervention.
- Adding extra type information. The difficulty of this step depends largely on how much passthrus are used to define types and constants.

To Convert or not to Convert

There are several reasons why OIDL users should convert to IDL class descriptions. Unlike OIDL, IDL offers SOM users multiple inheritance, exception handling, type checking, and automatic descriptor support. In addition, binaries generated from OIDL class descriptions are significantly larger and run more slowly than binaries generated from IDL class descriptions. If users choose not to convert their OIDL class descriptions to IDL, however, they can continue to use the SOM Compiler to update their classes with a few minor changes in protocol. These protocol changes are:

1. The SOM Compiler no longer generates a **.ph** (private) and **.h** (public) file, only a **.h** file that includes bindings for both private and public methods. To generate a public version of the **.h** file, first generate a **.sc** file by invoking the SOM Compiler on the **.csc** file with the **-ssc** option; then generate a **.h** file from the **.sc** file by invoking the SOM Compiler on the **.sc** file with the **-sh** option.
2. Because **.ph** files are no longer used, **passthru** statements directed toward **.ph** files should be redirected toward **.h** files.
3. **passthru** statements directed toward **.c** files should be removed or redirected toward **.ih** files.
4. Set the environment variables SMADDSTAR=1 and SMNOTC=2:

   For OS/2:

   ```
   SET SMADDSTAR=1
   SET SMNOTC=2
   ```

   For AIX:

   ```
   export SMADDSTAR=1
   export SMNOTC=1
   ```
5. Any methods that return structures should have the modifier **struct** attached to them. For example,

   ```
   UserStruct getUserStruct(), struct;
   ```

Converting .csc Files to .idl Files

The SOM Toolkit supplies a program, **ctoi**, to assist users in converting **.csc** files to **.idl** files. Before running **ctoi**, ensure that the directories containing files to convert have all the necessary **.sc** and **.psc** files already created. The SOM Compiler can be run with the **-ssc** and **-spsc** options to create **.sc** and **.psc** files from a **.csc** file.)

The conversion process requires a list of all the classes used in the files to be converted, so that forward references to classes can be handled correctly. Store this list of class names in some file. The name of this file must be specified to the SOM Compiler by the SMCLASSES environment variable:

For OS/2:

```
SET SMCLASSES=clsfile
```

For AIX:

```
export SMCLASSES=clsfile
```

The following command executes the **ctoi** conversion program:

```
ctoi  [file1 file2 ... ]
```

The **ctoi** program generates a **.idl** file for each specified **.csc** file.

Once you have run **ctoi**, you should be able to install and run your application program as usual. The following situations, however, may require attention:

- Be sure to change any of your installation batch files or makefiles that explicitly mention **.csc**, **.sc** or **.psc** files so that they instead refer to **.idl** files.
- Set the environment variables SMADDSTAR=1 and SMNOTC=2:

  For OS/2:

  ```
  SET SMADDSTAR=1
  SET SMNOTC=2
  ```

  For AIX:

  ```
  export SMADDSTAR=1
  export SMNOTC=1
  ```
- Any methods that return structures should have the modifier *struct* attached to them. For example,

  ```
  UserStruct getUserStruct(), struct;
  ```
- If any of your classes use IDL reserved words as function or variable names, then these names must be changed. Typical cases include string, context and interface.
- IDL does not permit the following notation for a struct type:

  ```
  data:
      struct stat fileStats;
  ```

  Instead, you must add a *typedef* in the IDL interface statement that introduces the data element:

  ```
  interface: filemi {
    typedef struct stat stat;
  ...
  #ifdef __SOMIDL__
     implementation {
      stat fileStats;
      ...
   };
  #endif
  };
  ```

  To have the typedef emitted into the **.h** header file, put the typedef within the interface statement, as shown above. If you don't want the typedef to be emitted in the **.h** header file, then put it outside the interface statement or in a separate file to be **#include**d. Alternatively, if you **#include** a central header file, then the typedef can be put in that header file.

  If you cannot simply add a typedef, due to name conflicts in other standard header files, then add a new type (such as `stat_t`, for the example above) and change your **.idl** files to reflect the new type name.

- The use of unbounded arrays is not allowed in IDL. For example,

  ```
  char *argv[];
  ```

  must be rewritten as:

  ```
  char **argv;
  ```

  or as:

  ```
  #define MAX_SIZE 32
  char *argv[MAX_SIZE];
  ```

- The unsigned char type is not supported by IDL. To effectively use unsigned chars, define the type `uchar_t` as follows:

  ```
  typedef octet uchar_t;
  ```

  The SOM Compiler will map this onto an unsigned char type in the **.h** header file.

- IDL does not permit structures to be passed by value. Instead, your methods must pass a pointer to a structure. Methods can, however, return a structure.

- Forward references are required in IDL. For all classes not in the ancestry of a class that are used in the interface statement for the class, the following statement must precede the class's interface statement:

  ```
  interface <className>;
  ```

- Numeric and string macros that you want to appear in your output files must be mapped onto string constants. For example,

  ```
  #define FILE_NAME_MAX  256
  #define FILE_NAME "hello.c"
  ```

  must be replaced by:

  ```
  const long FILE_NAME_MAX = 256;
  const string FILE_NAME = "hello.c";
  ```

- Public or private instance variables are converted to IDL attributes. However, there are some limitations, as follows: For instance variables that are explicit arrays (such as, `char x[10];` or `short y[20];`) the **ctoi** conversion will result in invalid IDL attributes, because IDL attributes cannot include array declarators. Attributes can be of a type that is an array, such as

  ```
  typedef char myarraytype[10];
  attribute myarraytype myarray;
  ```

  but not an explicit array, as in

  ```
  attribute char myarray[10];       /* not valid */
  ```

  If a **.csc** file contains a public or private instance variable that is an array, such as

  ```
  char myarray[10];
  ```

  the **ctoi** conversion facility will produce the following in the **.idl** file it generates:

  ```
  attribute char[10] myarray;
  ```

  This is invalid IDL; it must be fixed manually before the SOM Compiler will accept the **.idl** file. It is invalid not only because the array declarator is in the wrong place, but also because attributes cannot include array declarators at all. To fix it, introduce a typedef that defines an array type, and make that the type of the attribute, as shown:

  ```
  typedef char myarraytype[10];
  attribute myarraytype myarray;
  ```

  This limitation does not affect internal instance variables, just public and private ones. Internal instance variables are not converted to attributes.

- Most information contained in passthru lines directed to the implementation header (**.ih**) file should be moved to the implementation (**.c**) file. In addition, **passthru** statements directed toward **.c** files must be removed.
- If after running **ctoi**, you discover that you inadvertently omitted a class name from the file that the **SMCLASSES** environment variable refers to, it is best to update the class name file, remove the new **.idl** files, and recreate them using **ctoi**.
- Unlike OIDL, IDL does not include a private modifier for data and methods. Instead, private data and methods are surrounded by `#ifdef __PRIVATE__` and `#endif` directives. For example, to declare a method `foo` as a private method within an IDL specification, the following declaration would appear within the interface statement:

  ```
  #ifdef __PRIVATE__
  void foo();
  #endif
  ```

  To include private data/methods in a compilation of a **.idl** file, the SOM Compiler must be invoked with the **-D__PRIVATE__** option. If any of the data or methods in your **.csc** files are marked as private, then when using the SOM Compiler to generate binding files from the **.idl** files that **ctoi** creates from these **.csc** files, use the **-D__PRIVATE__** option to have the private data or methods included.

Adding Type Information

IDL, unlike OIDL, is strongly typed. This means that the SOM IDL compiler expects types and constants to be declared before they are referenced. If they are not, the SOM Compiler produces warning messages. Converting from OIDL to IDL does not require adding additional typing information (for example, typedefs and constant definitions), because these warning messages can be safely ignored. If this additional typing information is added when converting from OIDL to IDL, however, SOM provides additional functionality not available otherwise. For example, an Interface Repository can be created from a **.idl** file and the IDL specification can by type-checked only if the file declares types and constants before they are referenced.

In IDL, types (including typedefs, structs, unions and enums) are defined in a similar way to C. These types can be emitted into header files if they are defined within the interface statement for the class. Type definitions placed outside the interface statement are not transferred to header files. See **SOM Interface Definition Language** on page 116 for a complete discussion of defining types and constants in IDL.

passthru statements are not generally needed in IDL to define constants or types, although they may still be used to pass **#include** directives to header files.

Appendix C. 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                 | *+
```

Glossary

A

abstract class. A class that 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 always introduces methods that must be overridden in a subclass.

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.

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 method that extracts the arguments from the va_list, invokes the method, and stores its result. Also are registered with class objects when instance methods are defined. Invoked using the somApply function.

B

base class. See parent class.

Basic Object Adapter (BOA). A type of object adapter defined by CORBA to support a wide variety of common object implementations.

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.

BOA (basic object adapter) class. A CORBA interface, which defines generic object-adapter (OA) methods that a server can use to register itself and its objects with an ORB (object request broker).

BOA. See Basic Object Adapter.

C

casted dispatching. A form of method dispatching that uses casted method resolution.

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.

class variable. Instance data found within an object that is a class.

client code. 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.

compound name. In the Naming Service, a name that has multiple components. Name components are IDL structures.

constraint. In the Naming Service, an expression used to describe the characteristics of a bound object being searched for. Constraints are expressed in Constraint Language.

CORBA. The Common Object Request Broker Architecture established by the Object Management Group. The SOM Interface Definition Language used to describe the interface for SOM classes is fully compliant with CORBA standards.

D

data token. A value that identifies a specific instance variable within an object whose class inherits the instance variable derived class See subclass and subclassing.

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.

dispatch method. A method invoked in order to determine the appropriate method procedure to execute.Using dispatch methods facilitates dispatch-function resolution in SOM applications and enables method invocation on remote objects in DSOM applications.

DLL. dynamic link library.

dynamic dispatching. Method dispatching using dispatch-function resolution

Dynamic Invocation Interface (DII). The CORBA-specified interface, that is used to dynamically build requests on remote objects. DSOM applications can also use the somDispatch method for dynamic method calls when the object is remote.

dynamic link library. A piece of code that can be loaded (activated) dynamically. This code is physically separate from its callers. DLLs can be loaded at load time or at run time. Widely used term on OS/2 and other operating systems.

E

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 is transformed and formatted according to a user-defined template.

encapsulation. An object-oriented programming feature whereby the implementation details of a class are hidden from client programs, which are required to know the only interface of a class in order to use the class's methods and attributes.

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.

F

factory. An object that is capable of creating another object.

I

ID. See somId.

Implementation Repository. A database used by DSOM to store the implementation definitions of DSOM servers.

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).

index. In the Naming Service, an index that the user can create on specific properties in a naming context. It improves the performance of searches that involve a property.

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.

in-memory object. An object instantiated in memory. Differs from an object whose state can be stored in a persistent database for which no in-memory object has been instantiated.

instance method. A method valid for an object instance (versus 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. (Or object instance or just object.) A specific object, as distinguished from a class of objects. See also object.

Interface Repository (IR). The database that SOM optionally creates, providing 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.

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.

IR. Interface Repository.

L

location services daemon. A process whose primary purpose is to give DSOM clients the communications information they need to connect with an implementation server.

M

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.

managed object. An object subject to any of the SOMobjects object services.

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.

metastate. The state introduced to an object and used by an object service framework.

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 pointer. A pointer type that identifies one method on a single class. Method pointers are not ensured to be persistent among multiple processes.

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 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.

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.

N

name binding. In the Naming Service, a name-to-object association. Different names can be bound to an object in the same or different naming contexts at the same time.

name. In the Naming Service, an ordered sequence of name components, which are IDL structures composed of id and kind strings. A simple name has a single component.

names library. In the Naming Service, a library of names from that and other services. It allows names to evolve without affecting existing clients. Names are implemented as pseudo-objects, which are converted to and from structures.

naming context. In the Naming Service, an object that contains name-object associations (bindings).

naming scope. See scope.

Naming Service. A service that provides the ability to refer to objects by name. It organizes computing resources so that they easily can be located, identified, and categorized either in context or by explicit characterization.

nonstatic method. A special kind of SOM method.

O

object adapter. Defined by CORBA as being responsible for object reference, activation, and state-related services to an object implementation.

object definition. See class.

object implementation. See implementation.

object instance. See instance and object.

object passivation. The process of deleting the in-memory instantiation of an object, especially an object with persistent state even after being passivated.

object reactivation. The process of re-instantiating an object in-memory, especially when the object exists in persistent form even before being reactivated.

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.

object services base class. The base class for object services mix-in classes.

object services mix-in class. Any mix-in class introduced by an object service that is intended to be mixed-in to a managed object.

Object Services Server. A server that, with the DSOM object adapter, exports and imports object references. Aa a specialization of the DSOM framework, supports SOMobjects Object Services, handling such tasks as metastate and persistent object references.

object services server-object. The Object Services Server specialization (somOS Server) of the default DSOM framework server-object (SOMDServer).

objref. An abbreviation for object reference, specified by CORBA to be a value that unambiguously references an object.

OIDL. The original language used for declaring SOM classes. The acronym stands for Object Interface Definition Language. OIDL is still supported by SOM, but it does not include the ability to specify multiple inheritance classes.

OOP. object-oriented programming.

operation. See method.

ORB. (object request broker). A CORBA term designating the means by which objects tranparently make requests (that is, invoke methods) and receive responses from objects, whether they are local or remote.

overridden method. A method defined by a parent class and reimplemented (redefined or overridden) in the current class.

override. 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.

P

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.

persistent reference. An object reference that can survive the process or thread that created it.

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 method procedure.

process. A series of instructions (a program or part of a program) that a computer executes in a multitasking environment.

pragma. A compiler directive, usually specified in code by #pragma.

property. A name-value pair associated with a name binding. The name can be any CORBA String and the value is a CORBA any.

R

readers and writers. A reader is a process that does not intend to update the object, but wants to watch 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.

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.

S

scope. That portion of a program within which an identifier name has visibility and denotes a unique variable. An IDL source file forms a scope. An identifier can only be defined once within a scope.

server object. An artifact in the DSOM framework to assist in the mapping of object references to in-memory objects, and in-memory objects to object references. The mapping is used in the exportation and importation of object references.

shadowing. A technique that is required when any of the entry classes are subclassed. Shadowing causes instances of the new subclasses to be used as input for building the object graph, without requiring a recompile of emitter framework code.

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).

simple name. In the Naming Service, a name that has a single component. Name components are IDL structures.

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.

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.

SOMDServer. The default implementation of a server-object provided by the DSOM framework.

somId. A pointer to a number that uniquely represents a zero-terminated string. Such pointers are declared as type somId. In SOM, somIds are used to represent method names, class names, and so forth.

SOMOA. The DSOM implementation of a CORBA object adapter.

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.

somOSServiceBase. The module and interface name for the managed object base 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.

state (of an object). The data (attributes, instance variables and their values) associated with an object. See also behavior.

static linkage. Occurs when a program uses data or functions that are defined elsewhere. Simply declaring the existence of external data or functions does not create this linkage, actual usage of external data or functions is required.

static method. Any method you can access 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.

superclass. See parent class.

symbol. Any of a set of names 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.

T

target object. 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.

transient object. In CORBA, an object whose existence is limited by the lifetime of the process or thread that created it. In SOMobjects, more accurately an object with a transient state.

transient reference. An object reference whose existence is limited by the lifetime of the process or thread that created it.

U

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.

W

writers. See readers and writers.

Index

D

E

J

K

L

M

N

O

P

Q

R

Printed in U.S.A.

.