

OS/390



SOMobjects Programmer's Guide

OS/390



SOMobjects Programmer's Guide

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xxi.

Fourth Edition, September 1997

This is a major revision of, and obsoletes, GC28-1859-02.

| This edition applies to Version 2 Release 4 of OS/390 (5647-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+914+432-9405

FAX (Other Countries):

 Your International Access Code +1+914+432-9405

IBMLink (United States customers only): KGNVMC(MHVRCFS)

IBM Mail Exchange: USIB6TC9 at IBMMAIL

Internet e-mail: mhvrcfs@vnet.ibm.com

World Wide Web: <http://www.s390.ibm.com/os390>

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 1997. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	xxi
Examples in This Book	xxi
Programming Interface Information	xxii
Trademarks	xxii
About This Book	xxv
Who Should Use This Book	xxv
How This Book Is Organized	xxvi
Where to Find More Information	xxviii
Related SOMobjects Publications	xxviii
Introductory Object-Oriented Programming Publications	xxviii
CORBA Publications	xxix
Summary of Changes	xxxi
New Information	xxxi
Changed Information	xxxi

Part 1. Object Technology and the MVS Solution

Chapter 1. Introduction to Object-Oriented Technology on OS/390	1-1
Object-oriented (OO) Technology - Why?	1-1
OO Technology - What Is It?	1-1
Understanding Relationships	1-4
Understanding Frameworks	1-5
OO Technology - The Challenges	1-6
OO Technology - The System Object Model is IBM's Solution	1-6
OS/390 Support for OO Technology (SOMobjects)	1-7
Introducing OS/390 SOMobjects	1-7
Languages	1-8
The SOMobjects Compiler	1-9
The SOMobjects Run-Time Library	1-10
Frameworks provided in OS/390 SOMobjects	1-10
Distributed SOMobjects (DSOM)	1-10
Interface Repository Framework	1-10
Emitter Framework	1-11
Metaclass Framework	1-11
Migration Considerations	1-11
Migrating from OS/390 Version 1 Release 3 to OS/390 Version 2 Release 4	1-11
Migrating from Releases Prior to OS/390 Version 1 Release 3 to OS/390 Version 2 Release 4	1-11
Chapter 2. Understanding SOMobjects Programming	2-1
What Is the SOMobjects Programming Environment?	2-1
The Uses and Benefits of the SOMobjects Programming Environment	2-1
Understanding the SOMobjects Programming Environment	2-2
Understanding the SOM Kernel	2-2
SOMObject Class Object	2-2
SOMClass Class Object	2-2

SOMClassMgr Class Object and SOMClassMgrObject	2-3
SOM Kernel Relationships	2-4
Understanding SOMobjects Inheritance	2-4
Techniques for Deriving Subclasses	2-5
Multiple Inheritance	2-5
Understanding SOM Methods, Functions, and Macros	2-9
SOMobjects Methods	2-9
Method Resolution	2-9
SOMobjects Functions and Macros	2-11
Understanding SOMobjects Metaclasses	2-11
Understanding Dynamic Link Libraries (DLLs)	2-11
Initializing the SOMobjects Programming Environment	2-11
Getting Started with SOMobjects	2-12
Understanding the SOMobjects "Big Picture"	2-12
Building Class Libraries	2-12
Using IDL to Build Class Libraries into DLLs	2-12
Step 1: Determine Which Classes/Objects are Needed and Which SOM	2-13
Runtime Options/Services to Use	2-13
Step 2: Create Interface Definition Language (IDL) to Support Your	2-14
Designed Classes	2-14
Step 3: Compile Your IDL Datasets with the SOM Compiler, Creating	2-15
Your Client Header File (Bindings)	2-15
Step 4: Update the Implementation Template File with Code to Make	2-16
Methods Work	2-16
Steps 5 and 6: Compile the Implementation Code with the DLL Option,	2-16
Prelink and Link	2-16
Developing Client Applications	2-17
Step 1: Determine Which Classes/Objects are Needed and Which SOM	2-17
Runtime Option/Mode/Services to Use	2-17
Step 2: Create Your Application Code	2-17
Step 3: Compile Your Client Application Program with Client Header	2-18
Files	2-18
Step 4: Prelink and Link the Object Deck to Create the Client Load	2-18
Module	2-18

Part 2. Building Class Libraries and Developing Client Applications

Chapter 3. SOMobjects Interface Definition Language (IDL)	3-1
What is IDL?	3-1
Common Object Request Broker Architecture (CORBA) Compliant	3-2
Interface versus Implementation	3-2
The Uses and Benefits of IDL	3-3
Understanding IDL Syntax (Keywords, Directives, and Declarations)	3-3
Lexical Rules	3-4
Keywords	3-5
Include Directive	3-5
Type and Constant Declarations	3-6
Integral Types	3-6
Floating Point Types	3-6
Character Type	3-6
Boolean Type	3-7
Octet Type	3-7
Any Type	3-7

Constructed Types	3-7
Template Types (Sequences and Strings)	3-10
Arrays	3-13
Pointers	3-13
Object Types	3-13
Exception Declarations	3-14
“BAD_FLAG” Exception Declaration	3-15
Exception Declaration Within an Interface	3-16
Standard Exceptions Defined by CORBA	3-16
Interface Declarations	3-17
Constant, Type, and Exception Declarations Within the Body of an Interface Declaration	3-18
Attribute Declarations Within an Interface Declaration	3-19
Method Declarations Within an Interface Declaration	3-20
Module Declarations	3-24
Implementation Statements	3-24
Modifier Statements	3-25
Declaring Instance Variables and Staticdata Variables	3-35
Passing Parameters by Copying:	3-36
Passthru Statements	3-37
Introducing non-IDL Data Types or Classes	3-38
Comments within SOMobjects IDL	3-38
Adding attributes with IDL	3-39
Overriding a method using IDL	3-40
Designating ‘Private’ Methods and Attributes	3-40
Defining Multiple Interfaces in an IDL Data Set	3-41
Scoping and Name Resolution	3-41
Name Usage in Client Programs	3-42
Extensions to CORBA IDL Permitted by SOMobjects IDL	3-43
Pointer “*” Types	3-43
Unsigned Types	3-44
Implementation Section	3-44
Comment Processing	3-44
Generated Header Data Sets	3-45
Using IDL	3-45
Building a “Payroll” Class Library with Two Classes	3-45
Step 1: Determine Which Classes Are Needed	3-45
Step 2: Create Interface Definition Language (IDL) Datasets to Support Your Classes	3-46
Chapter 4. SOMobjects Compiler	4-1
What is the SOMobjects Compiler?	4-2
The Uses and Benefits of the SOMobjects Compiler	4-2
Understanding the SOMobjects Compiler	4-3
Generating Bindings	4-3
Tailoring the SOMobjects Environment Variables	4-8
Global Variables	4-8
Local Variables	4-8
Using the SOMobjects Compiler	4-9
Setting Up the SOMobjects Compiler to run from TSO, ISPF, OpenEdition	
Shell, or Batch	4-9
TSO READY	4-9
ISPF Command Line	4-9
ISPF SOM Compiler option	4-9

OpenEdition Shell	4-9
Batch	4-10
Setting Up the SOMobjects Compiler with Environment Variables	4-10
Understanding the Syntax of the SOMobjects Compiler	4-11
Using SOMobjects Compiler Examples	4-15
Continuation of the "Payroll" Example	4-17
The Public Definition Language (PDL) Utility	4-19
PDL Utility Syntax	4-20
PDL Utility Option	4-20
Chapter 5. Building Class Libraries	5-1
Direct-to-SOM (DTS) Build	5-1
Creating SOM-enabled Classes for C++/MVS	5-1
Creating C++/MVS Classes that are Accessible to C or COBOL	5-2
Creating SOM-enabled Classes for COBOL and Making COBOL Classes Accessible to C or C++	5-2
Dynamic Link Library (DLL) Build	5-2
What are DLLs?	5-4
The Uses and Benefits of DLLs	5-4
Understanding DLLs	5-4
DLL Building	5-4
Using DLLs	5-4
Creating a DLL	5-5
Continuation of the "Payroll" Example	5-5
Step 4: Update the Implementation Template with Your Method Code	5-5
Steps 5 and 6: Compile the Implementation Code with the DLL Option, Prelink and Link	5-8
Chapter 6. Developing Client Applications	6-1
Who Should Read This Chapter?	6-2
Understanding Client Programs	6-3
SOMobjects Usage Bindings	6-3
C/C++ Header Dataset Member Names	6-3
An Example Client Program	6-4
Using Classes	6-5
Declaring Object Variables	6-6
Creating Instances of a Class	6-7
For C programmers with Usage Bindings	6-7
For C++ programmers with Usage Bindings	6-10
Invoking Methods on Objects	6-12
Making Typical Method Calls	6-12
Accessing Attributes	6-16
Obtaining a Method's Procedure Pointer	6-17
Method Name or Signature Unknown at Compile Time	6-19
Using Class Objects	6-20
Getting the Class of an Object	6-20
Creating a Class Object	6-20
Using va_list Methods	6-24
Examples of va_list Usage	6-25
Using Name-Lookup Method Resolution	6-29
Compiling, Prelinking, and Linking	6-33
Running the Client Program	6-34
Application Access to DLLs	6-34
Setting Up Your Environment to Run Your Client Program	6-34

Running Your Client Application Program	6-34
Step 2: Create the Application Code	6-35
Steps 3 and 4: Compile the Application with the DLL Compiler Option, Prelink and Link	6-35
Step 5: Go (Run the Client Application Calling the “Payroll” Class)	6-39
Chapter 7. Distributed SOMobjects	7-1
Who Should Read This Chapter	7-1
What is DSOM?	7-2
Other Uses, Benefits, and Features of DSOM	7-2
A Conceptual Overview of Distributed SOMobjects	7-3
Platforms SOMobjects Runs On and Architecture It Complies With	7-4
SOM Subsystem	7-5
Naming Server	7-5
Security Server	7-5
OpenEdition Integrated Socket Support (TCP/IP and SNA)	7-5
DSOM Repository Files (Interface Repository and Implementation Repository)	7-5
Execution Environment Overview (TSO, OpenEdition Shell, APPC, etc.)	7-5
DSOM Application Overview	7-5
DSOM Tutorial	7-7
Application Components	7-7
The Stack Interface	7-7
Changing a Client Program from a Local to a Remote Stack	7-8
Code Differences and Similarities	7-10
Locate and Create Method	7-10
somdCreate Function	7-11
Naming Service	7-11
Finding Existing Objects	7-12
Stack Server Implementation	7-12
Stack Example Run-Time Scenario	7-13
Setting Up the SOMobjects Environment	7-15
Register Your Classes in the Interface Repository	7-15
DLL Considerations	7-15
Programming the Client Application	7-16
Process Flow of a Distributed SOMobjects Application	7-16
Initializing a Client Program	7-18
The ORB Object	7-18
The SOMD_Init Function	7-18
Finding Initial Object References	7-18
Creating Remote Objects	7-19
Finding a SOM Object Factory	7-20
Creating an Object from a Factory	7-22
Using the somdCreate Function	7-24
Making Remote Method Calls	7-24
Remote Object Invocation Methods	7-25
Memory Allocation and Ownership	7-27
Memory-Management Functions	7-29
Destroying Remote Objects	7-31
Exiting a Client Program	7-32
Compiling and Linking Clients	7-32
Programming the Server	7-32
Implementing SOM Classes in DSOM	7-33
Using SOM Class Libraries	7-33

Role of DSOM Generic Server Program (somdsvr)	7-33
Role of SOM Object Adapter	7-34
Role of SOMDServer	7-34
Implementation Constraints	7-34
How DSOM complies with the Common Object Request Broker Architecture (CORBA)	7-35
Mapping OMG CORBA Terminology onto DSOM	7-36
Object Request Broker Run-Time Interfaces	7-36
Object References and Proxy Objects	7-38
Interface Definition Language	7-40
C Language Mapping	7-40
Dynamic Invocation Interface (DII)	7-40
Implementations	7-40
Servers	7-41
Object Adapters	7-41
DSOM Limitations	7-42
DSOM Extensions	7-42
Chapter 8. Non-Distributed SOMobjects Examples	8-1
'Payroll' Example with IDL, SOM Compiled Output, Method Code, and Client Code	8-2
Overview of the Two Major Tasks for the "Payroll" Example	8-2
Basic Steps for Building and Using SOMobjects Classes	8-3
Building SOMobjects Classes	8-3
Step 1. Determine Classes/Objects Needed and SOM Runtime Option/Mode/Services to use	8-4
Your Mission as a Class Library Builder:	8-4
Step 2. Create the Source IDL	8-6
Step 3. Run the SOMobjects Compiler	8-11
Output of the SOMobjects Compiler	8-14
Step 4. Update the Implementation Template	8-16
Implementing SOMobjects Classes	8-16
The Implementation Template	8-17
Extending the Implementation Template	8-21
"Payroll" Class Library Implementation Code (in C)	8-25
Steps 5 and 6: Compile the Implementation Code with the DLL Option, Prelink and Link	8-31
Running Your Client Program	8-32
Step 1: Create the Application Code	8-32
Steps 2 and 3: Compile the Application with the DLL Compiler Option, Prelink and Link	8-35
Step 4: Go (Run the Client Application Calling the "Payroll" Class)	8-35
Building and Running an Application from the TSO Environment	8-36
Setting Up and Running Non-distributed Vehicle in Different OS/390 Environments	8-36
Building and Running a Non-Distributed Application in the Batch Environment	8-37
Building and Running a Non-Distributed Application from the OpenEdition Shell Environment	8-37
Building a Non-Distributed Application from the OpenEdition Shell Environment	8-37
Running a Non-Distributed Application from the OpenEdition Shell Environment	8-39

Building and Running a Non-Distributed Application from the TSO Environment	8-39
Building a Non-Distributed Application from the TSO Environment	8-39
Running a Non-Distributed Application from the TSO Environment	8-40
Summarized Non-Distributed Examples	8-40
Animals	8-40
Text Processing	8-42
Hello	8-44
Chapter 9. Distributed SOMobjects (DSOM) Examples	9-1
Setting Up and Running Distributed SOMobjects (DSOM) Vehicle in Different OS/390 Environments	9-1
Building and Running a Distributed Application in the Batch Environment	9-2
Building and Running a Distributed Application from the OpenEdition Shell Environment	9-2
Building and Running a Distributed Application from the TSO Environment	9-4
Summarized Distributed Examples	9-5
Stack	9-5
SStack	9-9
Vehicle	9-11
Animals	9-14
Phone	9-18
Chapter 10. Language Neutrality with SOMobjects: Examples	10-1
Four Scenarios	10-1
Scenario 1: C Client Program Using DTS C++ Classes	10-2
Building the Classes	10-4
Create the Header Files	10-4
Create C++ DLL	10-8
Developing and Running the Client Program	10-18
Create C Executable	10-18
Scenario 2: COBOL Program Using DTS C++ Classes	10-27
Building the Classes	10-27
Create the Header Files and IR Entries	10-27
Create the C++ Implementation Files	10-28
Developing and Running the Client Program	10-28
Create COBOL Executable	10-28
Scenario 3: COBOL Program Using COBOL Class that Inherits from C++ Classes	10-33
Building the Classes	10-34
Create the Header Files and IR Entries	10-34
Create the C++ Implementation Files	10-34
Developing and Running the Client Program	10-39
Create COBOL Executable	10-39
Scenario 4: C++ Program Using COBOL Class that Inherits C++ Classes	10-45
Building the Classes	10-46
Create the Header Files and IR Entries	10-46
Create the C++ Implementation Files	10-46
Developing and Running the Client Program	10-49
Create C++ Executable	10-50
Chapter 11. SOMobjects Advanced Topics	11-1
Understanding Classes and Hierarchy	11-1
Parent Class vs. Metaclass	11-1

SOM-derived Metaclasses	11-3
Understanding Metaclass Incompatibility	11-4
SOMobjects Derived Metaclasses	11-4
The Four Kinds of SOMobjects Methods	11-6
Static Methods	11-7
Nonstatic Methods	11-7
Dynamic Methods	11-7
Direct-call Procedures	11-7
Initializing and Uninitializing Objects	11-8
Initializer Methods	11-8
Declaring New Initializers in IDL	11-10
Considerations re: 'somInit' Initialization	11-12
Implementing Initializers	11-13
Selecting Non-default Ancestor Initializer Calls	11-14
Using Initializers when Creating New Objects	11-15
Uninitialization	11-16
Using 'somDestruct'	11-16
A Complete Example	11-17
Implementation Code	11-17
Customizing the Initialization of Class Objects	11-24
C/C++ Methods and Functions	11-25
Generating Output	11-25
Getting Information about a Class	11-25
Getting Information about an Object	11-27
Customizing SOM Runtime Features	11-28
Customizing Class Loading and Unloading	11-28
Customizing DLL Loading	11-28
Customizing DLL Unloading	11-29
Method Resolution Advanced Topics	11-30
Name-lookup Resolution	11-30
Dispatch-function Resolution	11-30
Customizing Memory Management	11-31
Clearing memory for objects	11-32
Clearing memory for the Environment	11-32
Customizing SOMobjects Output Processing	11-32
Customizing Error Handling	11-33
Chapter 12. Distributed SOMobjects (DSOM) Advanced Topics	12-1
Who Should Read This Chapter	12-1
TCP/IP Communications and the Well-known Port	12-1
Designing Local/Remote Transparent Programs	12-2
Class Objects	12-2
Object Creation	12-3
Using Factories to Create Objects	12-3
Using Factories While Controlling Memory Allocation	12-4
Gaining Access to Existing Objects	12-6
Proxy versus Object Destruction	12-6
Memory Management of Parameters	12-7
Distribution Related Errors	12-8
Generating and Resolving Object References	12-8
Support for CORBA Specified Interfaces to Local Objects	12-8
Data Types not Supported In Distributed Interfaces	12-9
SOM Objects That Do not Have IDL Interfaces	12-9
Procedure Methods	12-9

Global Variables	12-9
Class Data	12-9
Class Methods	12-10
Direct Instance Data Access	12-10
Passing Objects by Value	12-10
Object Invocation: Synchronous, Oneway, Deferred Synchronous and Asynchronous	12-11
Assignment in DTS C++	12-11
Summary of Local/Remote Guidelines	12-11
Method Classification for Local/Remote Transparency	12-12
Terms Used in Method Classification	12-13
Inquiring about a Remote Object Interface or Implementation	12-18
Working with Object References	12-19
Saving and Restoring References to Objects	12-19
Finding Existing Objects	12-21
Creating Remote Objects Using User-Defined Metaclasses	12-21
Updating the Implementation Repository Dynamically Using the Programmatic Interface	12-22
Advanced Memory-Management Options	12-24
Object-Owned Policy	12-24
somdReleaseResources method and object-owned parameters	12-26
Dual-owned policy	12-26
suppress_inout_free	12-27
Passing Objects by Copying	12-28
Passing Foreign Data Types	12-29
Example	12-30
Generic IDL for impctx	12-30
impctx Modifier with Opaque	12-31
impctx Modifier with Dynamic	12-31
impctx Modifier with Static	12-31
function-name Parameters	12-32
Using #pragma with Foreign Data Types	12-33
Writing Clients that are also Servers	12-33
Programming a Server	12-33
Server Run-Time Objects	12-35
Server Implementation Definition	12-35
SOM Object Adapter (SOMOA)	12-36
Server Object (SOMDServer)	12-37
Server Activation	12-37
Example Server Program	12-38
Initializing a Server Program	12-38
Initializing the DSOM Run-Time Environment	12-39
Initializing the Server's ImplementationDef	12-39
Initializing the SOM Object Adapter	12-39
When Initialization Fails	12-40
Processing Requests	12-40
Exiting a Server Program	12-41
Managing Objects in the Server	12-42
Customizing Factory Creation	12-48
Customizing Method Dispatching	12-49
Identifying the Source of a Request	12-50
Compiling and Linking Servers	12-51
Building Client-Only "Stub" DLLs	12-51
Creating User-Supplied Proxies	12-52

Customizing the Default Base Proxy Class	12-55
Peer vs. Client-Server Processes	12-56
Event-Driven DSOM Programs Using Event Manager (EMan)	12-56
Sample Server Using EMan	12-57
Dynamic Invocation Interface	12-59
The NamedValue Structure	12-60
The NVList Class	12-61
Creating Argument Lists	12-62
Building a Request	12-62
Initiating a Request	12-63
Invoking a Request with Example Code	12-64
Guidelines for Direct-to-SOM DSOM Programmers	12-66
Chapter 13. Collection Classes	13-1
Categories of Collection Classes	13-1
IsSame versus IsEqual Comparisons	13-2
Class Inheritance versus Element Inheritance	13-2
Object-Initializer Methods	13-2
Naming Conventions	13-3
Abstract Classes	13-3
Main Collection Classes	13-4
Hash Table Class: somf_THashTable	13-4
Dictionary Class: somf_TDictionary	13-5
Set Class: somf_TSet	13-5
Deque, Queue and Stack Class: somf_TDeque	13-6
Linked List Class: somf_TPrimitiveLinkedList	13-6
Sorted Sequence Class: somf_TSortedSequence	13-6
Priority Queue Class: somf_TPriorityQueue	13-7
Choosing the Best Class	13-7
Iterator Classes	13-8
Mixin Classes	13-9
Supporting Classes	13-10

Part 3. SOMobjects Frameworks

Chapter 14. Emitter Framework	14-1
What is the Emitter Framework?	14-1
The Uses and Benefits of the Emitter Framework	14-2
Understanding the Emitter Framework	14-3
The Structure of the Emitter Framework	14-3
The Abstract Syntax Graph and the Object Graph Builder	14-3
The entry classes	14-4
The Emitter Class	14-4
The template class and template definitions	14-4
Emitter Framework Classes	14-5
The emitter class (SOMTEmitC)	14-7
The template output class (SOMTTemplateOutputC)	14-10
The entry classes (SOMTEntryC, SOMTClassEntryC, ...)	14-12
SOMTEntryC	14-13
SOMTCommonEntryC	14-14
SOMTClassEntryC	14-14
SOMTBaseClassEntryC	14-15
SOMTMetaClassEntryC	14-15

SOMTModuleEntryC	14-16
SOMTPassthruEntryC	14-16
SOMTTypedefEntryC	14-16
SOMTDataEntryC	14-16
SOMTAttributeEntryC	14-17
SOMTMethodEntryC	14-17
SOMTPParameterEntryC	14-17
SOMTConstEntryC	14-17
SOMTEnumEntryC	14-18
SOMTSequenceEntryC	14-18
SOMTStringEntryC	14-18
SOMTUnionEntryC	14-18
SOMTEnumNameEntryC	14-18
SOMTStructEntryC	14-18
SOMTUserDefinedTypeEntryC	14-18
Using the Emitter Framework	14-19
Writing an Emitter—the Basics	14-19
Running the NEWEMIT Utility	14-19
An Example of NEWEMIT in More Detail	14-22
Step 1	14-22
Step 2	14-22
Steps 3 and 4	14-25
Step 5	14-25
Writing an Emitter—Advanced Topics	14-26
Defining New Symbols	14-26
Customizing Section-emitting Methods	14-29
Changing Section Names	14-29
Shadowing	14-30
Handling Modules	14-31
Error Handling	14-31
Chapter 15. The Interface Repository Framework	15-1
What is the Interface Repository (IR) Framework?	15-1
The Uses and Benefits of the IR Framework	15-3
Needed for Cross Language Support for Non-header Languages	15-3
Required for Systems that Use Dynamic Class Lookup	15-3
Understanding the IR Framework	15-4
Managing Interface Repository files	15-4
The SOM IR file	15-4
Managing IRs via the SOMIR Environment Variable	15-5
Placing ‘Private’ Information in the Interface Repository	15-6
Programming with Interface Repository Objects	15-7
The IR Eleven Classes of Objects	15-7
Methods introduced by IR classes	15-8
A Word About Memory Management	15-10
Using the IR Framework	15-12
Using the SOM Compiler to Build an Interface Repository	15-12
Accessing Objects in the Interface Repository	15-13
Using TypeCode pseudo-objects	15-18
Example of a TypeCode as a Pseudo-Object	15-19
Providing ‘Alignment’ Information	15-21
Using the ‘tk_foreign’ TypeCode	15-22
TypeCode Constants	15-23
Using the IDL Basic Type ‘Any’	15-23

Building an Index for the Interface Repository	15-24
Chapter 16. Event Management Framework	16-1
What is the Event Manager Framework?	16-1
Understanding the Event Manager Framework	16-1
Event Management Basics	16-1
Model of EMan usage	16-2
Event types	16-2
Registration	16-3
Registering for Events	16-4
Troubleshooting Hint	16-5
Unregistering for Events	16-5
An Example Callback Procedure	16-5
Generating Client Events	16-5
Processing Events	16-6
Example of Using a Timer Event	16-6
Interactive Applications	16-7
Event Manager Advanced Topics	16-8
Writing an X or MOTIF application	16-8
Extending EMan	16-8
Using EMan from C++	16-9
Using EMan from Other Languages	16-9
Tips on Using EMan	16-9
Chapter 17. Metaclass Framework	17-1
What is the Metaclass Framework?	17-1
A Note about Metaclass Programming	17-3
The SOMMBeforeAfter metaclass	17-3
SOM IDL for 'Barking' Metaclass	17-5
C implementation for 'Barking' Metaclass	17-5
Composition of Before/After Metaclasses	17-6
Notes and Advantages of 'Before/After' Usage	17-9
The SOMMSingleInstance Metaclass	17-9
The SOMMTraced Metaclass	17-10
SOM IDL for 'TracedDog' Class	17-11
The SOMM_MVS_Secure Metaclass	17-12
An Example	17-14
Security Administration for Protected Methods	17-16
An Example of Inherited Method Protection	17-16
An Example Illustrating the Use of Generic and Discrete Profiles	17-17
Appendix A. Setting up Configuration Files	A-1
About Configuration Files	A-1
Configuration File Variable Settings	A-1
Comparison of Global and Local Configuration File Values	A-2
Configuration File Syntax	A-3
Stanzas and Keyword Variables	A-3
[somc] Stanza (SOMobjects Compiler)	A-3
[somd] Stanza (Distributed SOMobjects)	A-6
[somir] Stanza (Interface Repository)	A-7
[somras] Stanza (Error Log and Trace Facility)	A-8
[somsec] Stanza (Security Service)	A-9
[SOM_POSSOM] Stanza (Persistent Object Service)	A-10

Appendix B. SOM IDL Language Grammar	B-1
Glossary	X-1
Index	X-17

Figures

0-1.	How this book is organized.	xxvi
1-1.	Example of a generic class, a Checking Account class, and a Checking Account object	1-3
1-2.	Example of a class hierarchy	1-4
1-3.	Example of encapsulation	1-5
2-1.	The SOMobjects programming environment provides four primitive objects , three of them class objects.	2-4
2-2.	Multiple Inheritance can Create Naming Conflicts.	2-6
2-3.	Resolution of Multiple-Inheritance Ambiguities.	2-7
2-4.	The SOMobjects "Big Picture" for a C application.	2-13
2-5.	Your application and the runtime environment initialization process.	2-18
3-1.	Steps 1 & 2 of the SOMobjects "Big Picture".	3-1
3-2.	"Payroll" class design.	3-46
3-3.	"Payroll" IDL statements.	3-47
4-1.	Step 3 of the SOMobjects "Big Picture", showing C binding files.	4-1
4-2.	Structure of the SOMobjects Compiler with potential outputs.	4-2
4-3.	SOMobjects Compiler batch job containing an inline procedure.	4-17
4-4.	Output of the SOMobjects Compiler batch job.	4-18
5-1.	Steps for building and accessing DLLs.	5-3
5-2.	"Payroll" implementation code.	5-6
5-3.	"Payroll" JCL to compile the implementation code, prelink and link into a DLL load module.	5-9
6-1.	Steps for building and accessing DLLs.	6-2
6-2.	A C client program that accesses the "Payroll" DLL.	6-4
6-3.	A C++ client program that accesses the "Payroll" DLL.	6-5
6-4.	Creating ten instances of the "employee" class in C.	6-9
6-5.	Creating ten instances of the "employee" class in C++.	6-11
6-6.	Name-Lookup Resolution	6-32
6-7.	Client application code which calls the "Payroll" DLL load module.	6-35
6-8.	JCL to compile, prelink, and link the client application to create the executable file.	6-37
6-9.	"Payroll" REXX exec to set up your online interactive environment.	6-40
7-1.	Overview of OS/390 Distributed SOMobjects	7-4
7-2.	Sample IDL from sommvs.SGOSSMPI.IDL(STACK).	7-7
7-3.	DSOM process flow of a client application invoking methods on SOM objects in other processes.	7-17
7-4.	Construction of a Proxy Class in DSOM	7-38
8-1.	"Payroll" class hierarchy with data attributes and methods of its classes.	8-6
8-2.	"Payroll" class library IDL statements.	8-8
8-3.	JCL to SOM compile "Payroll".	8-12
8-4.	"Payroll" class library C implementation template created with the SOMobjects compiler.	8-15
8-5.	"Payroll" class library implementation code.	8-26
8-6.	Client application code which calls the "Payroll" DLL load module.	8-33
8-7.	"Payroll" REXX exec to set up your online interactive environment.	8-36
10-1.	C program using DTS C++ classes.	10-3
10-2.	C++ pbook header file for Scenario 1.	10-4
10-3.	C++ pentry header file for Scenario 1.	10-5
10-4.	C++ mystring header file for Scenario 1.	10-6

10-5.	JCL to compile the header files for the C++ classes in Scenario 1.	10-7
10-6.	Implementation file pbook for Scenario 1.	10-9
10-7.	Implementation file pentry for Scenario 1.	10-13
10-8.	Implementation file mystring for Scenario 1.	10-15
10-9.	JCL to compile the implementation files for the C++ classes in Scenario 1.	10-17
10-10.	JCL to prelink and link to create the C++ DLL and the corresponding definition side-deck in Scenario 1.	10-18
10-11.	C main program for Scenario 1.	10-19
10-12.	C compile JCL of the main program for Scenario 1.	10-21
10-13.	C prelink and link JCL of the object deck for Scenario 1.	10-22
10-14.	C executable JCL for Scenario 1.	10-23
10-15.	Output of the RUNJOB step for Scenario 1.	10-24
10-16.	JCL to SOM compile the IDL files for Scenario 2.	10-28
10-17.	COBOL main program for Scenario 2.	10-29
10-18.	Compile the COBOL main program, prelink and link the C++ definition side decks and run the program for Scenario 2.	10-32
10-19.	Output of the RUNJOB step for Scenario 2.	10-33
10-20.	COBOL subclass for Scenario 3.	10-35
10-21.	JCL to compile the COBOL subclass for Scenario 3.	10-37
10-22.	JCL to build the COBOL subclass DLL (Cobphone) which includes the C++ classes object decks for Scenario 3.	10-38
10-23.	JCL to SOM compile the COBOL-generated IDL file for Scenario 3.	10-39
10-24.	COBOL main program for Scenario 3.	10-40
10-25.	JCL to compile, prelink and link the COBOL main program with the Cobphone definition side deck and run job for Scenario 3.	10-43
10-26.	Output of the RUNJOB step for Scenario 3.	10-44
10-27.	C++ main program for Scenario 4.	10-48
10-28.	JCL to compile the C++ main program for Scenario 4.	10-50
10-29.	JCL to prelink and link the C++	10-51
10-30.	Output of the RUNJOB step for Scenario 4.	10-53
11-1.	A class has both parent classes and a metaclass	11-2
11-2.	Parent classes and metaclasses each have their own independent inheritance hierarchies	11-3
11-3.	Example of metaclass incompatibility	11-4
11-4.	Example of a derived metaclass	11-5
11-5.	Multiple inheritance requires derived metaclasses	11-6
11-6.	A default initializer ordering of a class's inheritance hierarchy.	11-10
11-7.	Two classes that introduce new initializers.	11-11
11-8.	Implementing initializers.	11-14
11-9.	A C++ example illustrating the implementation and use of initializers and destructors.	11-17
11-10.	C++ Implementation code and a client program for the initialization and destructor classes.	11-18
11-11.	Illustration of how a default ancestor initializer call is replaced with a non-default initializer call.	11-20
11-12.	Illustration of how a default ancestor initializer call is replaced with a non-default initializer call.	11-22
11-13.	Illustrated output of a default ancestor initializer call being replaced with a non-default initializer call.	11-23
12-1.	Relationship of the SOMOA with servers and the DSOM runtime.	12-36
12-2.	Potential Deadlocks Exist Using EMan and DSOM	12-57
13-1.	Format of the somf_THashTable class	13-4
13-2.	Selection chart for the Main Collection Classes	13-8

14-1.	Structure of the SOM Compiler	14-2
14-2.	The Structure of the SOM Emitter Framework	14-3
14-3.	Emitter Framework classes	14-6
14-4.	The Entry Class Hierarchy	14-13
14-5.	The NEWEMIT process.	14-20
15-1.	IR part of the SOMObjects "Big Picture".	15-1
15-2.	Relationship of the IR Framework to the SOMObjects Compiler and the end user.	15-2
15-3.	Interface Repository Framework classes	15-7
15-4.	JCL to dump the IR output.	15-16
15-5.	IR dump output.	15-17
15-6.	TypeCode interfaces defined within IDL.	15-20
17-1.	The Primitive Objects of the SOM Run Time	17-2
17-2.	Metaclass Framework Class Organization	17-3
17-3.	A Hierarchy of Metaclasses	17-4
17-4.	Example for Composition of Before/After Metaclasses	17-7
17-5.	Three Techniques for Composing Before/After Metaclasses	17-8
17-6.	Three Techniques for Creating a "FierceBarkingDog"	17-8
17-7.	All Methods That Are Invoked on "Collie" Are Traced	17-11
17-8.	The SOMM_MVS_Secure Metaclass	17-13
17-9.	Three Techniques for Creating a Secure Class	17-14
17-10.	Inherited Method Protection	17-17

Notices

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

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

IBM 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:

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
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

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

Examples in This Book

The examples in this book are samples only, created by IBM Corporation. These sample examples are not part of any standard or IBM product and are provided to you solely for the purpose of assisting you in the development of your applications. The examples are provided "as is". IBM makes no warranties, express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, regarding the function or performance of these examples. IBM shall not be liable for any damages arising out of your use of the sample examples, even if they have been advised of the possibility of such damages.

These sample examples can be freely distributed, copied, altered, and incorporated into other software, provided that it bears the above disclaimer intact.

All examples that were shipped with the product had a high level qualifier (*hlq*) of *SOMMVS*. *SOMMVS* may have been changed when these datasets were copied into your OS/390 system by your system administrator. Any other time you see *hlq*

mentioned in this book, it means that the dataset has a user-definable high level qualifier.

Note: Before you can begin running any of these examples, the SOMobjects environment needs to be configured on your system. Consult with your system administrator to ensure that the SOMobjects environment has been configured. For more information on how to configure the SOMobjects environment, see the *OS/390 SOMobjects Configuration and Administration Guide*.

Programming Interface Information

This publication is intended to help the customer use SOMobjects to build object-oriented class libraries. This publication documents General-use Programming Interface and Associated Guidance Information provided by SOMobjects.

General-use programming interfaces allow the customer to write programs that obtain the services of SOMobjects.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AD/Cycle
AIX
C++/MVS
C/MVS
C/370
CICS
IBM
Language Environment
OpenEdition
OS/2
OS/390
RACF
RMF
RS/6000
SP
SOMobjects
System Object Model

C-bus is a trademark of Corollary, Inc.

Microsoft, Windows, and the Windows 95 logo are trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

X/Open is a trademark of X/Open Company Limited.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

The term "CORBA" used throughout this publication refers to the Common Object Request Broker Architecture standards promulgated by the Object Management Group, Inc.

The term "ANSI C" used throughout this publication refers to American National Standard X3.159-1989.

About This Book

This book describes the OS/390 SOMobjects product, to be called SOMobjects throughout the rest of this book. The SOMobjects product is based on the SOMobjects Developer Toolkit and this book explains how programmers using IBM C/C++ for MVS Version 3 Release 1 (or later), or IBM COBOL for OS/390 and VM Version 2 Release 1 (or later) can:

- Use SOMobjects to build object-oriented (OO) class libraries
- Write application programs using class libraries that have been implemented using SOMobjects.

Note that throughout the rest of this book, any reference to C or C++ means IBM C/C++ for MVS/ESA Version 3 Release 1 (or later) and any reference to COBOL means IBM COBOL for OS/390 and VM Version 2 Release 1 (or later). Any reference to Language Environment for MVS & VM means Language Environment for MVS & VM Version 1 Release 7 or above. The support for C/C++ in Language Environment for MVS & VM is also available as the MVS C/C++ Language Support feature on MVS/ESA SP Version 5 Release 2.2 at Language Environment for MVS/VM 1.7 level or above.

In addition to this book, refer to

- *OS/390 SOMobjects Programmer's Reference, Volume 1*
- *OS/390 SOMobjects Programmer's Reference, Volume 2*
- *OS/390 SOMobjects Programmer's Reference, Volume 3*

for more specific information about the classes, methods, functions, and macros supplied with SOMobjects.

Who Should Use This Book

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

- Use SOMobjects to build object-oriented (OO) class libraries
- Note:** Most of the examples in this book are in C (some are in C++ and some in COBOL). They use the Interface Definition Language (IDL) and the SOMobjects Compiler to create implementation templates. If you plan to create SOM classes and objects using the direct-to-SOM (DTS) support that is part of some OO programming languages, such as C++ or COBOL, you should refer to their respective documentation.
- Write application programs using class libraries that have been implemented using SOMobjects.

You should also have some knowledge of running jobs on MVS, such as using job control language (JCL), OS/390 OpenEdition, Time Sharing Option (TSO*), interactive system productivity facility (ISPF) or restructured extended executor language (REXX).

This book uses the common terminology of object-oriented programming. A number of important terms are everyday English words that take on specialized meanings. These terms appear in the Glossary at the back of this book. You

might consult the Glossary if the unusual significance attached to an otherwise ordinary word puzzles you.

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

How This Book Is Organized

Figure 0-1 shows how this book is organized.

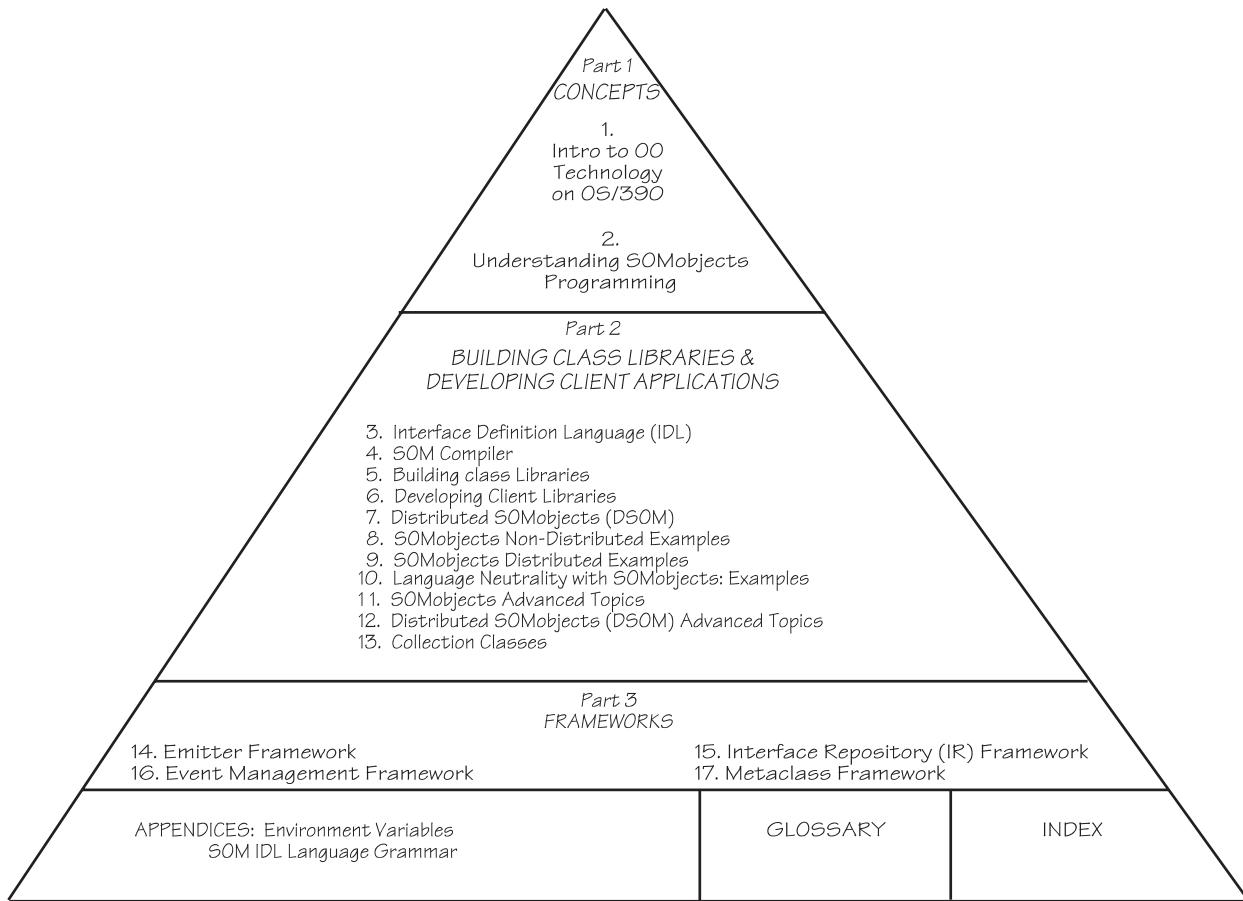


Figure 0-1. How this book is organized.

This book consists of three parts.

- **Part 1** consists of two chapters:
 - Chapter 1 describes object-oriented programming basics, along with its values and challenges. It then describes how the OS/390 solution answers those challenges.
 - Chapter 2 discusses the SOMobjects programming environment and gives a “Big Picture” overview of “Building Class Libraries” and “Developing Client Applications”.
- **Part 2** details the user tasks of “Building Class Libraries” and “Developing Client Applications” and consists of eight chapters:

- Chapter 3 gives the syntax of the SOMobjects interface definition language (IDL) and it introduces a “Payroll” example.
- Chapter 4 provides directions for running the SOMobjects compiler and shows how to compile the “Payroll” IDL created in chapter 3.
- Chapter 5 describes how to build a class library in SOMobjects using DLLs, and Direct-to-SOM (DTS). A “Payroll” DLL is created at the end of this chapter.
- Chapter 6 gives examples of how a client application can access a class library and uses “Payroll” as the DLL to access.
- Chapter 7 contains information about Distributed SOMobjects (or DSOM) which 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 or platforms.
- Chapter 8 is the SOMobjects non-distributed examples chapter and discusses how classes are built by class library builders and accessed by clients. It expands upon the “Payroll” example and also presents non-distributed examples in multiple OS/390 environments.
- Chapter 9 is the SOMobjects distributed examples chapter and discusses how classes are built by class library builders and accessed by clients. It also gives distributed examples in multiple OS/390 environments.
- Chapter 10 demonstrates language neutrality with SOMobjects examples. The languages discussed are C, C++ and COBOL.
- Chapter 11 contains advanced topics of SOMobjects that you may be interested in as you become more familiar with the programming concepts of SOMobjects.
- Chapter 12 contains advanced topics of distributed SOMobjects (DSOM) that you may be interested in as you become more familiar with the programming concepts of DSOM.
- Chapter 13 contains information on how to use SOMobjects Collection Classes. This chapter should be used in conjunction with *OS/390 SOMobjects Programmer's Reference, Volume 3*.
- **Part 3** contains four chapters on the frameworks that SOMobjects provides in creating support classes and databases for object-oriented programming with SOMobjects:
 - Chapter 14 contains information on the Emitter Framework.
 - Chapter 15 contains information on the Interface Repository (IR)
 - Chapter 16 contains information on the Event Manager Framework.
 - Chapter 17 contains information on the Metaclass Framework.

The **appendices**:

- Provide environment variables for the application programmer.
- Provide the grammar for SOMobjects IDL

The **glossary** provides brief definitions of terminology related to SOMobjects.

The **index** enables the reader to locate specific information quickly.

Where to Find More Information

Where necessary, this book references information in other books, using shortened versions of the book title. For complete titles and order numbers of the books for all related products that are part of OS/390, see *OS/390 Information Roadmap*, GC28-1727.

Related SOMobjects Publications

The OS/390 SOMobjects publications library includes the following:

- *OS/390 SOMobjects: Getting Started*
- *OS/390 SOMobjects Configuration and Administration Guide*
- *OS/390 SOMobjects Programmer's Guide*
- *OS/390 SOMobjects Object Services*
- *OS/390 SOMobjects Programmer's Reference, Volume 1*
- *OS/390 SOMobjects Programmer's Reference, Volume 2*
- *OS/390 SOMobjects Programmer's Reference, Volume 3*
- *OS/390 SOMobjects Messages, Codes, and Diagnosis*

OS/390 SOMobjects books are also available in softcopy on the OS/390 Collection (SK2T-6700) CD-ROM. IBM provides one copy of the CD-ROM automatically with the basic material for OS/390. You can order additional copies for a fee.

Introductory Object-Oriented Programming Publications

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

- Orfali (Robert), Harkey (Dan), Edwards (Jeri), *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc., ISBN 0-471-12993-3
- Taylor, David, *Object Oriented Technology: A Manager's Guide*, Addison-Wesley 1990, ISBN 0-201-56358-4.
- Booch, G, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings 1994, ISBN 0-8053-5340-2.
- Budd, T, *An Introduction to Object-Oriented Programming*, Addison-Wesley 1991, ISBN 0-201-54709-0.
- Cox, B, and Novobilski, A, *Object-Oriented Programming, An Evolutionary Approach*, 2nd Edition, Addison-Wesley 1991, ISBN 0-201-54834-8.
- Lau, Christina, *Object-Oriented Programming Using SOM and DSOM*, Van Nostrand Reinhold. 1994 ISBN 0-442-01948-3.
- Cattell, R.G., *Object Data Management, Object-Oriented and Extended Relational Database Systems*, Addison-Wesley Publishing Company, Inc. 1994 ISBN 0-201-54748-1.

CORBA Publications

- *The Common Object Request Broker: Architecture and Specification*, published by the Object Management Group and x/Open.

Summary of Changes

Summary of Changes for GC28-1859-03 OS/390 Release 3

This book contains information previously presented in *OS/390 SOMobjects User's Guide*, GC28-1859-02, which supports Version 1 Release 3.

This book includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

New Information

Function:

- “Building an Index for the Interface Repository” on page 15-24 describes how to build indexes for the Interface Repository.
- Chapter 13, “Collection Classes” on page 13-1 describes how to use the Collection Classes in SOMobjects. More information on these classes can be found in *OS/390 SOMobjects Programmer's Reference, Volume 3*.

Changed Information

- “Summarized Non-Distributed Examples” on page 8-40 has been reorganized into a more user-friendly, step-by-step “How to run the summarized non-distributed SOMobjects examples section”.
- “Summarized Distributed Examples” on page 9-5 has been reorganized into a more user-friendly, step-by-step “How to run the summarized distributed SOMobjects (DSOM) examples section”.
- Appendix A, “Setting up Configuration Files” on page A-1 has been modified to include changed or additional information on environment variables for SOMobjects.

The following changes appear only in the online version of this publication. A pair of vertical dots (:) in the left margin indicates changes to the text and illustrations.

This revision reflects the deletion, addition, or modification of information to support miscellaneous maintenance items.

Part 1. Object Technology and the MVS Solution

Chapter 1. Introduction to Object-Oriented Technology on OS/390

This chapter introduces the basic concepts and benefits of object-oriented technology and the OS/390 offerings related to this technology and consists of the following topics:

- “Object-oriented (OO) Technology - Why?”
- “OO Technology - What Is It?”
- “OO Technology - The Challenges” on page 1-6
- “OO Technology - The System Object Model is IBM's Solution” on page 1-6
- “Introducing OS/390 SOMobjects” on page 1-7
- “Migration Considerations” on page 1-11

Object-oriented (OO) Technology - Why?

Current technology in application development may not provide the flexibility, productivity, and quality you need to gain market advantage.

OO technology can address these application development issues. Object-oriented technology shifts the emphasis from traditional procedural approaches to the recognition of objects that *model your business*.

Object-oriented technology:

- Enables you to build and maintain custom applications that are portable and interoperable in a heterogeneous, multi-vendor environment
- Helps you respond to quickly changing business needs
- Allows code reuse, which lowers the overall cost of computing, improves programmer quality and productivity, and reduces development time and maintenance costs
- Increases application code quality through building programs out of existing, tested components
- Gives the ability to quickly enhance or extend applications.

OO Technology - What Is It?

If you are already familiar with object-oriented technology concepts and terminology, you may want to skip this section and proceed to “OO Technology - The System Object Model is IBM's Solution” on page 1-6.

Object-oriented technology takes its name from the fact that the central paradigm of programming has shifted from “procedures” to “objects”.

Let's begin with several definitions of object-oriented terms:

classes define the implementation of an object. Some like to think of a class as a “cookie cutter” of objects.

objects are instances of classes. They are discrete units of data and code that represent real world things. They are the “cookie” created by the “cookie cutter”.

data (also called attributes) is the content of the class or object.

- Data within a class is called a *variable*. Variables are like containers that at first have no value. For example, *color* is a container which has no real value until you put a color into it.
- Data within an object is called a *value*. Values are the contents of the container. For example, *blue* could be the value of the variable *color*.

methods are the code (or procedures) associated with an object that define the behavior associated with that object. Objects communicate by issuing messages (calls) to one another.

messages are the signals sent from one object to another that request the receiving object to use one of its defined methods. This is how a programming task is accomplished. The message (sometimes called a *call* in OO terms) can also tell the object to change its data.

One of the great values of OO technology is that objects typically represent real world things that model the actual entities a business deals with. With OO design, a programmer has much the same view of the problem or situation as the business end user. For example, a programmer for a financial institution, such as a bank, would code objects such as a checking account, a customer order, or a financial report, that have methods that implement the behavior of the real world objects.

In Figure 1-1 on page 1-3, the Checking Account class contains:

- *Data*: Name, Account#, Balance
- *Methods*: Open_Account, Close_Account, Withdraw, Deposit

The checking account object contains:

- *Data*: John Smith (Name), 012345 (Account #), \$275.00 (Balance)
- *Methods*: Open_Account, Close_Account, Withdraw, Deposit

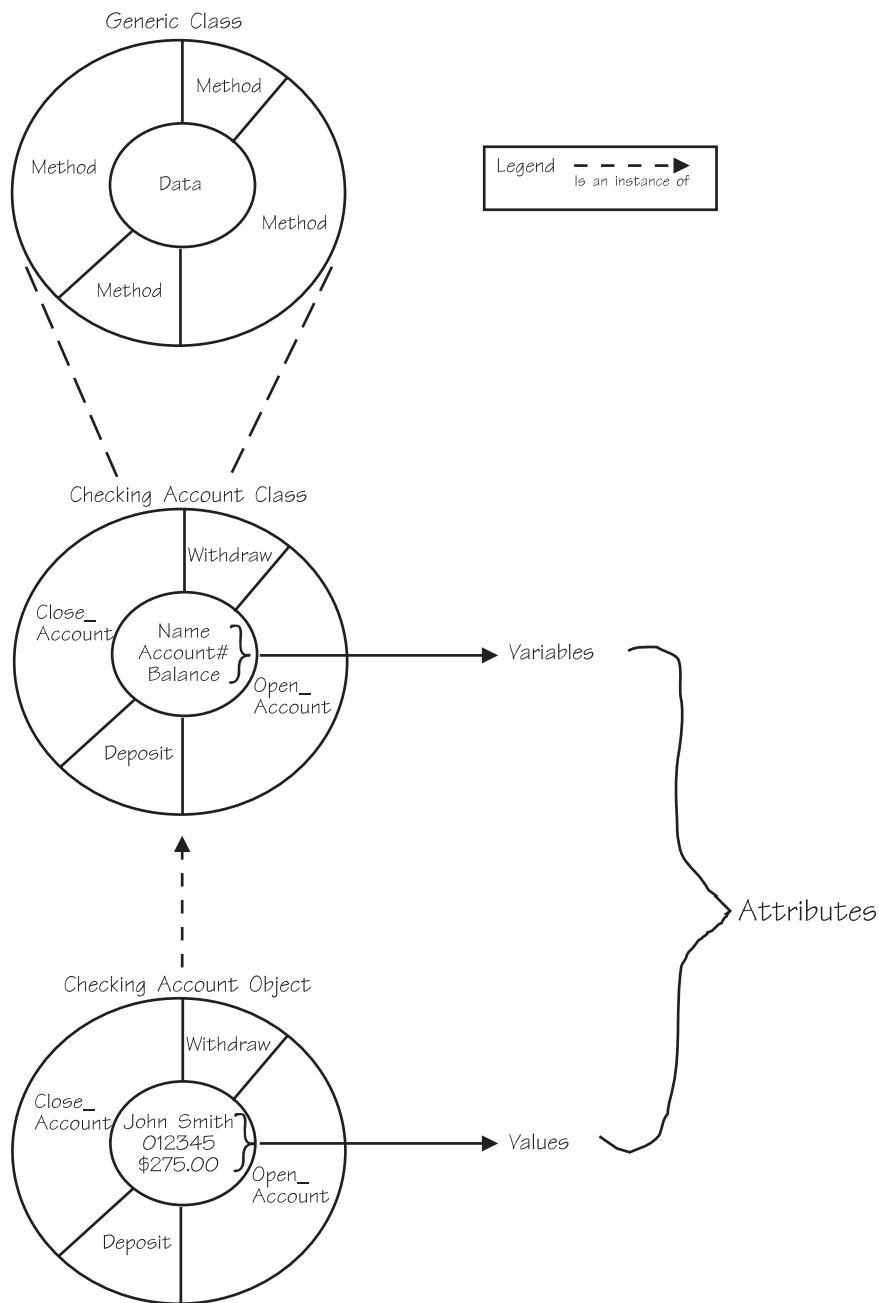


Figure 1-1. Example of a generic class, a Checking Account class, and a Checking Account object

Typically there are many objects of the same type; many checking accounts, many customer orders, or many financial reports. Objects of the same type are grouped in a class (for example, a Checking Account class).

The Checking Account class is used to create individual instances of checking account objects. The class defines the number of variables and their type. The instance contains the variable and its current value. Each instance will contain the same set of variables, but not necessarily the same values (for example, John

Smith's checking account object includes actual data values of his Name, Account #, and Balance). Programmers work with classes of objects to develop applications.

Understanding Relationships

Classes are organized in a hierarchy. When using SOMobjects, every class has a *parent* class (sometimes called a *base class*) except the root of the hierarchy.

A code sharing mechanism, called *inheritance*, allows reuse of methods in the definition of subclasses. A *child* class (sometimes called *subclass*) inherits the methods and data variables from the parent class. In Figure 1-2, both the Checking Account and Loan Account classes inherit the Open_Account and Close_Account methods from the (parent) Account class.

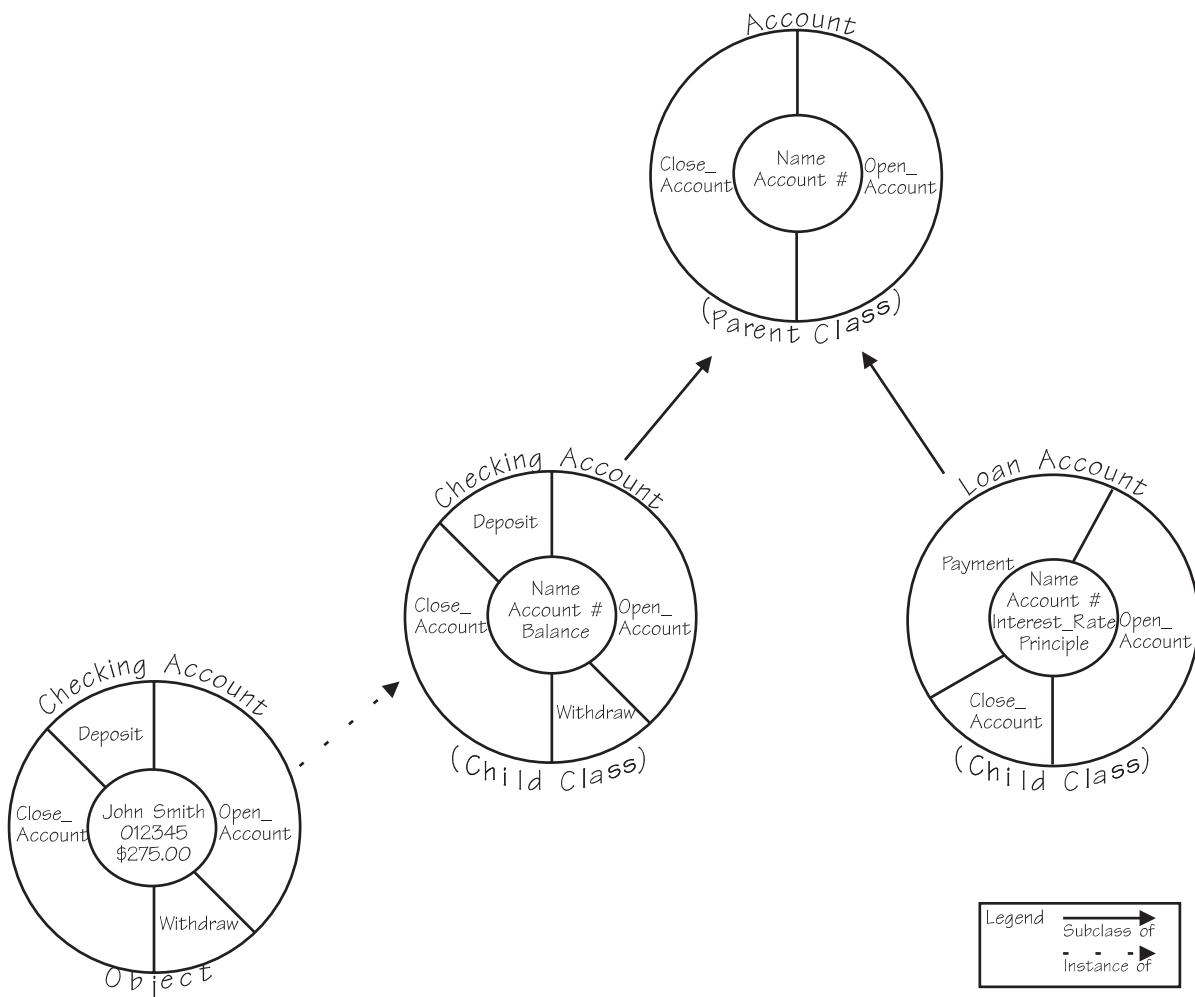


Figure 1-2. Example of a class hierarchy. *Account* is the parent class and has two subclasses: *CheckingAccount* and *LoanAccount*

A subclass can add new methods and data variables (for example, the `Payment` method is a new method associated only with the `Loan Account` class).

Polymorphism is the capability of allowing the same message to be sent to objects of different classes and having each of those objects respond in their own (possibly unique) way.

Encapsulation offers programmers a means to hide internal information; the internal information can only be accessed through the methods defined for that object. For example, the Account class programmer can hide the implementation of the *Name* attribute by requiring other programs to access the name only through Account's methods. Because of this, other programs will not be affected in the future if the implementation of the *Name* attribute changes. See Figure 1-3 for an example of how balance data is hidden and can only be accessed by a defined method.

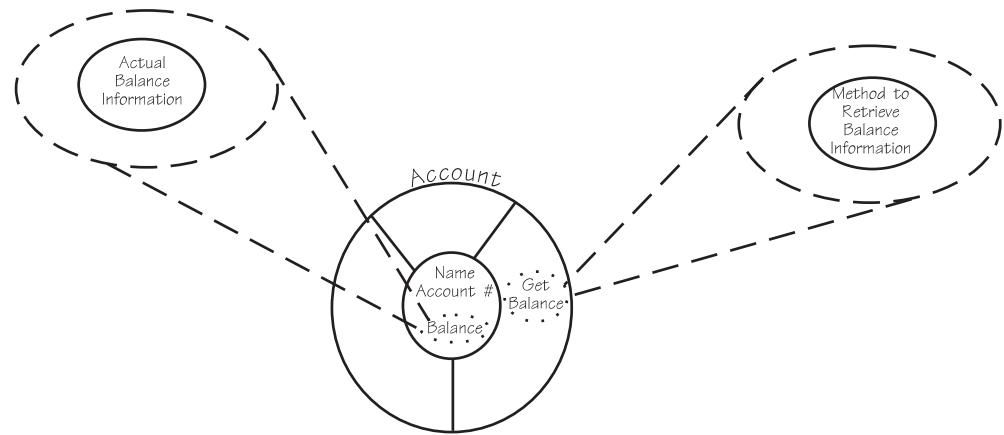


Figure 1-3. Example of encapsulation

Encapsulation also allows programmers to change an object with little or no effect on the users of the object (for example, you could add a method *Compute_Interest* to the Account class without affecting existing users of the class).

Understanding Frameworks

Libraries that contain classes relevant to a particular domain (for example, the domain of communications or graphics or transportation) are called *frameworks*.

Frameworks are pre-built class libraries built specifically to support descendant classes and act as off-the-shelf models for building applications to solve specific business problems. They should have clearly defined classes from which new classes should inherit. They should also clearly document what the descendant classes should redefine to operate correctly within that framework (within the set of behaviors and expectations coded into the ancestor framework classes).

In practice, a framework consists of a mixture of concrete classes (off-the-shelf components) and abstract classes, which must be further refined to support a particular application or subsystem. This scaffolding provides an environment that allows developers to focus on their domain expertise.

If this task is performed easily enough, and if the inherited functionality is rich enough, then the framework is likely to be heavily used.

OO Technology - The Challenges

While object-oriented technology holds great promise, it also has its challenges:

- Language barriers

Although several OO languages are available (one example is C++), programs coded in one language cannot be linked to (or operate with) programs of another language. Classes can not be shared between two different languages. This limits the ability to reuse classes.

Various object languages combine and store classes differently. For example, C++ must be compiled and linked in order to combine its classes. Other languages use different processes and tools to combine their classes.

- Unnecessary recompiling

Application programs need to be recompiled to share classes that have had changes made to them. There is no facility to do this automatically without a recompile.

- Incompatible models

Several OO models exist for objects and they interact differently. These models are provided through the languages and toolkits provided with them. Because the models differ, programs developed in one model may be limited in scope.

- Protecting source code

Vendors who provide class libraries generally must provide them in source-code form. This means that there is no current way to protect source code from other vendors. It also means the class users are burdened with cumbersome details.

OO Technology - The System Object Model is IBM's Solution

The System Object Model (**SOM**) provides a mechanism for defining classes and class libraries, as well as object management services.

SOMobjects addresses the challenges of OO technology because it is:

- Language independent

SOMobjects provides the capability for applications written in one language to use objects developed in other languages.

- Recompiling is not needed

SOMobjects also allows class libraries to have interfaces (methods) added without requiring the applications using these classes to be recompiled.

SOMobjects dynamically binds applications to objects during runtime.

Note: You can delete or modify private interfaces that were not available for client use. But any public or existing interface (one that has been established) must be maintained until you rebase your code, which requires the client to recompile.

- Interface Definition Language (IDL)

SOMobjects supports the Interface Definition Language (IDL) defined by the Object Management Group's (OMG) CORBA specification. IDL provides a

standardized way to define object interfaces, enabling class libraries to easily interact.

- Code is in executable form only

SOMobjects allows you to ship the implementation piece of your classes in executable form only by separating the class interface from its implementation.

Source code is protected and doesn't burden other class vendors with details they do not have to learn or understand.

OS/390 Support for OO Technology (SOMobjects)

Numerous vendors provide object-oriented technology through advanced desktop solutions that take on the challenge of addressing application development issues. However, there is also a need to address application development issues in your current OS/390 environment so that you can continue to maintain, build, and enhance your traditional business applications. SOMobjects addresses that need. By leveraging existing applications with the ability to develop new ones, you can get the most out of your investment.

With SOMobjects you can begin to shift your application programming focus to the object-oriented approach and in so doing, take advantage of object-oriented features to address your large scale business problems.

SOMobjects gives you all of the benefits of SOM plus the strengths of OS/390.

OS/390's strengths are:

- Processing large commercial applications and huge transaction processing loads, as well as processing in batch mode.
- Providing industrial strength integrity, security, and availability of data.

Introducing OS/390 SOMobjects

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

- With SOMobjects, 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.

SOMobjects extends the advantages of object-oriented programming to programmers who use non-object-oriented programming languages.

SOMobjects accommodates changes in implementation details without breaking the binary interface to a class library or requiring recompiling client programs. If changes to a SOMobjects class do not require source code changes in client programs, then those client programs do not need to be recompiled. SOMobjects 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.

The rest of this section consists of the following topics:

- “Languages”
- “The SOMobjects Compiler” on page 1-9
- “The SOMobjects Run-Time Library” on page 1-10
- “Frameworks provided in OS/390 SOMobjects” on page 1-10

Languages

The SOMobjects Runtime Library supports the use of several different programming languages in client programs and class implementations.

This support enables the user of a SOMobjects class and the class library builder of the SOMobjects class to not have to use the same programming language, and neither is required to use an object-oriented language. The independence of client language and implementation language also extends to subclassing: a SOMobjects class can be derived from other SOMobjects classes, and the child may or may not be implemented in the same language as the parent class(es).

The following are the languages that work with SOMobjects:

- C/C++ for OS/390
- IBM COBOL for OS/390 and VM Version 2 Release 1

For examples of language neutrality with SOMobjects using C, C++ and COBOL, see Chapter 10, “Language Neutrality with SOMobjects: Examples” on page 10-1.

SOMobjects 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 SOMobjects class use the same programming language. SOMobjects is said to be language-neutral for the following reasons:

- All SOMobjects interactions consist of standard procedure calls. The C linkage convention is used.
- The form of the SOMobjects Application Programming Interface (API) can vary widely from language to language, due to SOMobjects bindings. Bindings are a set of macros and procedure calls that make implementing and using SOMobjects classes more convenient by tailoring the interface to a particular programming language.
- SOMobjects supports several mechanisms for method resolution that can be readily mapped into the semantics of a wide range of object-oriented programming languages. Thus, SOMobjects class libraries can be shared across object-oriented languages with differing object models. A SOMobjects 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 SOMobjects 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.
- Interfaces to SOMobjects classes are described in CORBA's Interface Definition Language (IDL). SOMobjects supports all CORBA-defined data types.
- The SOMobjects bindings for the C language are compatible with the C bindings prescribed by CORBA.
- All information about the interface to a SOMobjects class is available at run time through a CORBA-defined Interface Repository.

SOMobjects does not replace existing object-oriented languages; it complements them so that application programs written in different programming languages can share common SOMobjects class libraries. For example, SOM can be used with C++ to:

- Provide upwardly compatible class libraries, so that when a new version of a SOMobjects class is released, client code need not 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 SOMobjects classes implemented in C++.
- Allow C++ programs to use SOMobjects classes implemented using other languages.
- Allow other language users to implement SOMobjects classes derived from SOMobjects classes implemented in C++.
- Allow C++ programmers to implement SOMobjects classes derived from SOM classes implemented using other languages.
- Allow encapsulation (implementation hiding) so that SOMobjects 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 SOMobjects 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 SOMobjects Compiler

SOMobjects contains the SOMobjects Compiler to build classes in which interface and implementation are decoupled. The SOMobjects Compiler reads the IDL definition of a class interface and generates:

- an implementation skeleton for the class
- bindings for class implementers
- bindings for client programs.

Bindings are language-specific macros and procedures that make implementing and using SOMobjects classes more convenient. These bindings offer a convenient interface to SOMobjects 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 SOMobjects objects as C++ objects, so that C++ programs can invoke methods on SOMobjects objects in the same way they invoke methods on C++ objects. In addition, SOMobjects objects receive full C++ typechecking, just as C++ objects do. The SOMobjects 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 SOMobjects bindings.

The SOMobjects Run-Time Library

The SOMobjects 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 SOMobjects 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 SOMobjects 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 SOMobjects 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 OS/390 SOMobjects

In addition to the SOMobjects Compiler and the SOM run-time library, SOMobjects provides a set of frameworks (class libraries) that can be used in developing object-oriented applications. These include Distributed SOMobjects (DSOM), the Interface Repository Framework, the Emitter Framework and the Metaclass Framework.

Distributed SOMobjects (DSOM)

Distributed SOMobjects (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 it were local. DSOM supports distribution of objects among processes within a workstation, and it also supports objects between/among address spaces in an OS/390 system. See Chapter 7, "Distributed SOMobjects" on page 7-1 for more information.

Interface Repository Framework

The Interface Repository is a database, optionally created and maintained by the SOMobjects 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. See Chapter 15, “The Interface Repository Framework” on page 15-1 for more information.

Emitter Framework

The Emitter Framework is a collection of SOMobjects classes that you write as your own emitters. Emitter is a general term used to describe a back-end output component of the SOMobjects Compiler. Each emitter takes input information about an interface, generated by the SOMobjects Compiler as it processes an IDL specification, and produces output organized in a different format. SOMobjects 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++. See Chapter 14, “Emitter Framework” on page 14-1 for more on the Emitter Framework.

Metaclass Framework

The Metaclass Framework is a collection of SOMobjects metaclasses that provide functions used by SOMobjects class designers to modify the default semantics of method invocation and object creation. These metaclasses are described in Chapter 17, “Metaclass Framework” on page 17-1.

Migration Considerations

There are two migration paths to consider when migrating to OS/390 Version 2 Release 4:

1. Migrating from OS/390 Version 1 Release 3 to OS/390 Version 2 Release 4
2. Migrating from releases prior to OS/390 Version 1 Release 3 to OS/390 Version 2 Release 4.

Migrating from OS/390 Version 1 Release 3 to OS/390 Version 2 Release 4

There are no migration issues when migrating from OS/390 Version 1 Release 3 to OS/390 Version 2 Release 4.

Migrating from Releases Prior to OS/390 Version 1 Release 3 to OS/390 Version 2 Release 4

The following migration issues must be considered when migrating from releases prior to OS/390 Version 1 Release 3 to OS/390 Version 2 Release 4:

1. SOMPROF/SOMLPROF

The SOMPROF/SOMLPROF profile settings have moved to a SOMENV specification. The change is in the file contents as well as how you (the user) specify the data set. Specifically, SOMENV may be specified as a SOMENV DD allocation or as a program environment variable. The content is from a list of variable settings to multiple “stanzas”, each of which contains its own list of variable settings. These stanzas are SOM-framework related. For more information on the stanzas and the variables associated with them, see Appendix A, “Setting up Configuration Files” on page A-1. For those users of OS/390 Version 1 Release 2 SOMobjects, you will need to create/update a configuration file. For more information on the configuration file, see *OS/390 SOMobjects Configuration and Administration Guide*.

2. Static linking support

You no longer will be able to statically link the SOM Kernel. In particular, OO COBOL users must use IBM COBOL for OS/390 and VM Version 2 Release 1 and access SOM kernel functions through COBOL's DLL support.

3. va_list

- In previous releases, a va_list was an array of two pointers.
- In this release, you can control the structure of a va_list via:

```
#define_VARARG_EXT_
```

- This is an enhancement to LE 1.7.
- If used in a module, va_list will be defined as a pointer.
- If used, make sure the #define is before includes of stdio.h and/or stdarg.h.
- If not used, va_list will continue to be defined as a block of two pointers.

4. exception_type and completion_status

In previous releases, *exception_type* and *completion_status* were each one byte in size. In this release, the size of these structures changed to four bytes. This change has been made in order to comply with the CORBA specification.

This change primarily impacts programs accessing the SOM *environment* structure to detect error situations. Field *_major* in this structure is of type *exception_type*, which is now four bytes long.

C and C++ programs accessing variables of type *exception_type*, such as *_major*, must be recompiled with the new level of SOM header files in order to function properly on this release of SOMobjects.

OO COBOL programs that define the SOM *environment* structure explicitly and check the "major" field, must be recoded. For example, if your COBOL program contains the following *environment* structure:

```
01 Environment-structure-old.  
 02 major pic X.  
    88 NO-EXCEPTION      value X"00".  
    88 USER-EXCEPTION   value X"01".  
    88 SYSTEM-EXCEPTION value X"02".  
 02 pic x(15).
```

The revised coding for OS/390 V1R3 SOMobjects would be:

```
01 Environment-structure.  
 02 major pic 9(9) binary.  
    88 NO-EXCEPTION      value 0.  
    88 USER-EXCEPTION   value 1.  
    88 SYSTEM-EXCEPTION value 2.  
 02 pic x(12).
```

For more information on using *exception_type* and *completion_status* structures, see *OS/390 SOMobjects Messages, Codes, and Diagnosis*.

Chapter 2. Understanding SOMobjects Programming

Before designing and building classes or developing and running an application, you should consider planning for the SOMobjects programming environment that will support your application.

This chapter addresses the following topics:

- “What Is the SOMobjects Programming Environment?”
- “The Uses and Benefits of the SOMobjects Programming Environment”
- “Understanding the SOMobjects Programming Environment” on page 2-2
- “Initializing the SOMobjects Programming Environment” on page 2-11
- “Understanding the SOMobjects “Big Picture”” on page 2-12

What Is the SOMobjects Programming Environment?

The *SOMobjects programming environment* refers to the environment that manages your objects and your SOM-enabled application as it runs, including the SOMobjects class library (the SOM kernel) and the SOM-enabled frameworks that you may use to augment your application.

The SOMobjects programming library provides a set of functions used primarily for creating objects and invoking methods on them. When an application program is run, the data structures and objects that are created, maintained, and used by the functions in the SOMobjects programming library make up the *SOMobjects programming environment*.

The Uses and Benefits of the SOMobjects Programming Environment

A distinguishing characteristic of the SOMobjects programming environment is that SOMobjects classes are represented by objects, called *class objects*. By contrast, object-oriented languages such as C/C++ for OS/390 treat classes strictly as compile-time structures that have no properties at SOMobjects execution time. In SOMobjects, each class has a corresponding object. This provides three advantages:

- Your application program can access information about a class at SOMobjects execution time

The information about the class can include the relationships the class has with other classes, the methods the class supports, the size of each instance of the class, and so on.

- You will not need to recompile your application as often

Because much of the information about a class is established at SOMobjects execution time (rather than at compile time), recompiles are only required when changes are made to the base IDL of the class.

- You can adapt the techniques for subclassing and inheritance

Because class objects are instances of metaclasses you define in SOMobjects, you can build object-oriented solutions to problems that generally cannot be easily addressed within an object-oriented programming (OOP) context.

- Many of the traditional OS/390 execution environments will support object-oriented applications and their use of the SOMobjects programming environment. In particular, SOMobjects applications will be supported in the CICS, batch, TSO, OpenEdition Shell, APPC and started task execution environments.

Understanding the SOMobjects Programming Environment

This section contains common conceptual and reference information that class designers and builders, application programmers, and application users need to consider before coding classes and applications. This section contains information about:

- “Understanding the SOM Kernel”
- “Understanding SOMobjects Inheritance” on page 2-4
- “Understanding SOM Methods, Functions, and Macros” on page 2-9
- “Understanding SOMobjects Metaclasses” on page 2-11
- “Understanding Dynamic Link Libraries (DLLs)” on page 2-11

Understanding the SOM Kernel

When the SOMobjects programming environment is initialized, four primitive SOMobjects 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 so forth.

SOMObject Class Object

SOMObject is the primitive class for all SOMobjects classes. It defines the essential behavior common to all SOMobjects objects. All user-defined SOMobjects classes are derived, directly or indirectly, from this class. That is, every SOMobjects class is a subclass of SOMObject or of some other class derived from SOMObject. SOMObject has no instance variables, thus objects that inherit from SOMObject incur no size increase. They do inherit a suite of methods that provide the behavior of all SOMobjects objects.

SOMClass Class Object

Because SOMobjects classes are SOMobjects execution time objects, and since all SOMobjects execution time objects are instances of some class, it follows that a SOMobjects class object must also be an instance of some class. The class of a class is called a *metaclass*. Therefore, 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* 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 operations of a supervisory nature. Also, class methods facilitate inheritance of instance methods from parent classes.

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

SOMClass is the primitive class for all SOMobjects metaclasses. That is, all SOMobjects metaclasses must be subclasses of SOMClass or of some metaclass derived from SOMClass. SOMClass defines the essential behavior common to all SOM class objects. In particular, SOMClass provides:

- 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.

SOMClass is a subclass (or child) of SOMObject. Therefore, SOMobjects class objects can also perform the same set of basic instance methods common to all SOMobjects objects; this is what allows SOMobjects classes to be real objects in the SOMobjects programming environment. SOMClass also has the unique distinction of being its own metaclass (that is, SOMClass defines its own class methods).

A user-defined class can designate as its metaclass either SOMClass or another user-written metaclass descended from SOMClass. If a metaclass is not explicitly specified, SOMobjects determines one automatically.

SOMClassMgr Class Object and SOMClassMgrObject

The third primitive SOMobjects class is SOMClassMgr. A single instance of the SOMClassMgr class is created automatically during SOMobjects initialization. This instance is referred to as the SOMClassMgrObject, because it is pointed to by the global variable SOMClassMgrObject. The object SOMClassMgrObject has responsibility for:

- Maintaining a *registry* (a SOMobjects execution time directory) of all SOMobjects classes that exist within the current process
- Assisting in the dynamic loading and unloading of class libraries.

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, **somFindClassMethod**, 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.

SOM Kernel Relationships

Relationships among the four primitive SOMobjects execution time objects are illustrated in Figure 2-1. Again, the primitive classes supplied with SOMobjects are SOMObject, SOMClass, and SOMClassMgr. During SOMobjects initialization, the latter class generates an instance called SOMClassMgrObject. The left-hand side of Figure 2-1 shows parent-class relationships between the built-in SOMobjects 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, of SOMObject, and of SOMClassMgr, which are all class objects at runtime. SOMClassMgr is the class of SOMClassMgrObject.

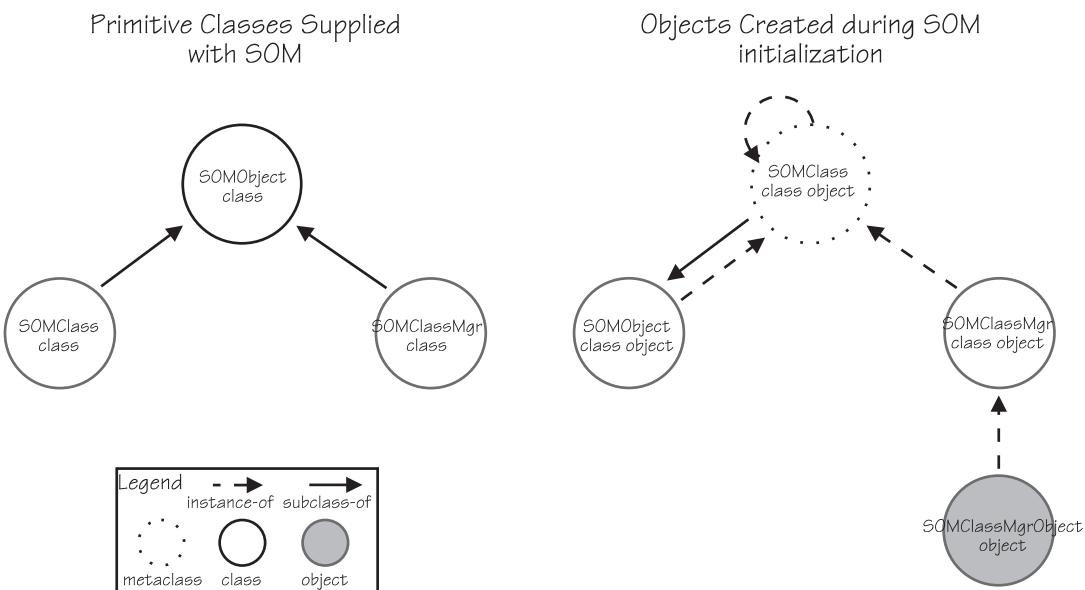


Figure 2-1. The SOMobjects programming environment provides four primitive objects , three of them class objects.

For more details on the functions that are included in the four primitive objects of the SOMobjects programming environment, see *OS/390 SOMobjects Programmer's Reference, Volume 1*.

Understanding SOMobjects 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.

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. See Figure 2-2 on page 2-6 for an example.

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 2-2 on page 2-6. (They are implemented using two different method-table entries, each of which is associated with its introducing class.)

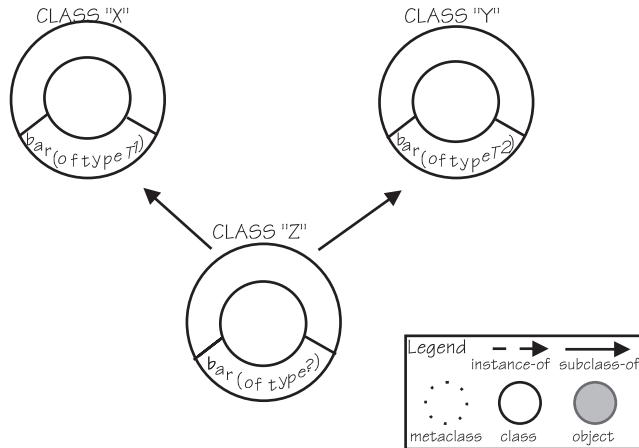


Figure 2-2. Multiple Inheritance can Create Naming Conflicts.

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 “Using Name-Lookup Method Resolution” on page 6-29), 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 2-3 on page 2-7. 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.

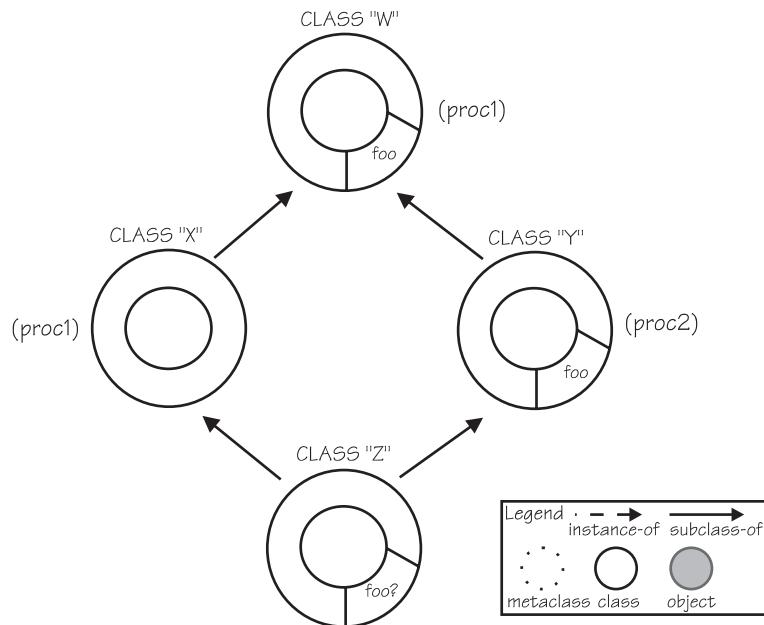


Figure 2-3. Resolution of Multiple-Inheritance Ambiguities.

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 left-most 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 2-3, 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 3-25.

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 sub-class
- 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.

Understanding SOM Methods, Functions, and Macros

To build SOMobjects classes, you need to understand which methods, functions, and macros SOMobjects provides, what the purpose is for each, and how they can help you.

The three SOM base classes, SOMObject, SOMClass, and SOMClassMgr provide methods needed by the System Object Model. SOMClass, as the primitive metaclass, provides methods for

- Instance creation
- Class initialization/termination
- Access to information about the class
- Testing class's version, ancestors, and supported methods
- Methods for locating static and dynamic methods.

SOMClassMgr acts as a SOMobjects execution time-registry for all SOM class objects that are active and assists with dynamic loading and unloading of class libraries. It provides methods for locating and loading classes from files using information in the interface repository.

SOMObject, as the primitive class, provides a suite of methods needed by all SOM objects for:

- Initialization/termination of objects
- Accessing information about an object
- Methods required for dispatch method resolution.

The following sections describe SOMobjects methods (and how they are resolved), functions, and macros.

SOMobjects Methods

You can run SOMobjects methods dynamically.

Dynamic methods are added to a class object at SOMobjects execution time. This gives the most flexibility for defining a class's interface.

SOMobjects methods perform a variety of class-related tasks, including:

- Getting information about a class
- Getting information about an object
- Output processing
- Class loading
- Some debugging.

For information on specific methods SOMobjects provides, refer to “[Implementing SOMobjects Classes](#)” on page 8-16.

Method Resolution

Method resolution is the step of determining which procedure to execute in response to a method invocation. For example, consider this scenario:

- Class “Dog” introduces a method “bark”, and
- A subclass of “Dog”, called “BigDog”, overrides “bark”, and

- A client program creates an instance of either “Dog” or “BigDog” (depending on some SOMobjects execution time criteria) and invokes method “bark” on that instance.

Method resolution is the process of determining, at run time, which method procedure to execute in response to the method invocation (either the method procedure for “bark” defined by “Dog”, or the method procedure for “bark” defined by “BigDog”). This determination depends on whether the receiver of the method (the object on which it is invoked) is an instance of “Dog” or “BigDog” (or perhaps depending on some other criteria).

SOMobjects allows class implementers and client programs considerable flexibility in deciding how SOMobjects performs method resolution. In particular, SOMobjects supports three mechanisms for method resolution, described in order of increased flexibility and increased computational cost:

- Offset resolution
- Name-lookup resolution
- Dispatch-function resolution.

The following section describes *offset resolution*, which is the more commonly used method resolution.

Offset Resolution: When using SOMobjects's C and C++ language bindings, offset resolution is the default way of resolving methods, because it is the fastest (nearly as fast as an ordinary procedure call). For those familiar with C++, it is roughly equivalent to the C++ “virtual function” concept.

Although offset resolution is the fastest technique for method resolution, it is also the most constrained. Specifically, using offset resolution requires these constraints:

- The name of the method to be invoked must be known at compile time
- The name of the class that introduces the method must be known at compile time (although not necessarily by you)
- The method to be invoked must be part of the introducing class's interface.

To perform offset method resolution, SOMobjects first obtains a method token from a global data structure associated with the class that introduced the method. This data structure is called the ClassData structure. It includes a method token for each method the class introduces. The method token is then used as an “index” into the receiving objects, i.e. receiver's method table, to access the appropriate method procedure. Because it is known at compile time which class introduces the method and where in that class's ClassData structure the method's token is stored, offset resolution is quite efficient.

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 SOMobjects, 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. See “Name-lookup Resolution” on page 11-30 for more information. Still more flexibility can be obtained by using

dispatch-function resolution. See “Dispatch-function Resolution” on page 11-30 for more information.

SOMobjects Functions and Macros

SOMobjects also provides functions and macros that can be used for a variety of purposes. Functions are generally used when you are setting up your application environment, and do not normally affect object management. Macros are essentially for the same purpose, but for different tasks.

Function tasks include:

- Class loading
- Storage allocation
- Object, class, and method naming (for manipulation purposes)
- Customization (for example, memory management)
- Exception and error handling
- Generating output

Macro tasks include:

- Defining your local environment
- Debugging

For more information, refer to *OS/390 SOMobjects Programmer's Reference, Volume 1*.

Understanding SOMobjects Metaclasses

The SOMobjects classes have slightly different relationships than regular objects classes, especially with regards to metaclasses and hierarchy. If you want more detail on these relationships, see “Understanding Classes and Hierarchy” on page 11-1.

Understanding Dynamic Link Libraries (DLLs)

SOMobjects builds class libraries dynamically .

Dynamic Link Libraries (DLLs) are a collection of functions and variables packaged in a separate load module that can be dynamically accessed from an application program.

This book goes into more detail on building dynamic link libraries (DLLs). To see how to build a class library dynamically, see Chapter 5, “Building Class Libraries” on page 5-1.

Initializing the SOMobjects Programming Environment

In addition to creating the four primitive SOMobjects objects, initialization of the SOMobjects execution time environment also involves initializing global variables to hold data structures that maintain the state of the environment. Other functions in the SOMobjects execution time library rely on these global variables. For more information on this topic, see “Tailoring the SOMobjects Environment Variables” on page 4-8 and Appendix A, “Setting up Configuration Files” on page A-1.

For application programs written in C or C++ that use the language-specific bindings provided by SOMobjects, the SOMobjects programming environment is

automatically initialized the first time any object is created. If you are using other languages, you must initialize the SOMobjects execution time environment explicitly by calling the `somEnvironmentNew` function (provided by the SOMobjects execution time library) or invoke `SOM_MainProgram` macro before using any other SOMobjects functions or methods.

Getting Started with SOMobjects

To get a better understanding of how to get started with SOMobjects in both a non-distributed and distributed environment. See *OS/390 SOMobjects: Getting Started*, which describes how to set up your environment and run a simple “Automobile” example in non-distributed SOMobjects and in distributed SOMobjects (DSOM).

The remainder of this chapter contains a more in depth discussion of the topics discussed in *OS/390 SOMobjects: Getting Started*.

Understanding the SOMobjects “Big Picture”

With the help of the concepts discussed previously, it is now possible to explain the two major tasks associated with the SOMobjects “Big Picture”:

- Building class libraries
- Developing client applications that use the classes in class libraries.

Building Class Libraries

The first major task associated with the SOMobjects “Big Picture” is to build a class library. This book will cover how to build a dynamic link library (DLL) and then, how to access that DLL. There are other approaches to building class libraries in SOMobjects. They are:

- Direct-to-SOM (DTS) Build
- Object Cobol (using IBM COBOL for OS/390 and VM Version 2 Release 1)

For more information on these approaches and DLL builds, refer to Chapter 5, “Building Class Libraries” on page 5-1 and to Chapter 8, “Non-Distributed SOMobjects Examples” on page 8-1.

Dynamic Link Libraries (DLLs) are a collection of functions and variables that can be dynamically accessed from an external program or application. The rest of this section on “Building Class Libraries” will describe the process of creating a DLL build using IDL.

Using IDL to Build Class Libraries into DLLs

Figure 2-4 on page 2-13 shows the relationship of the SOMobjects components with the two major tasks that SOMobjects helps you accomplish in building DLLs by using IDL.

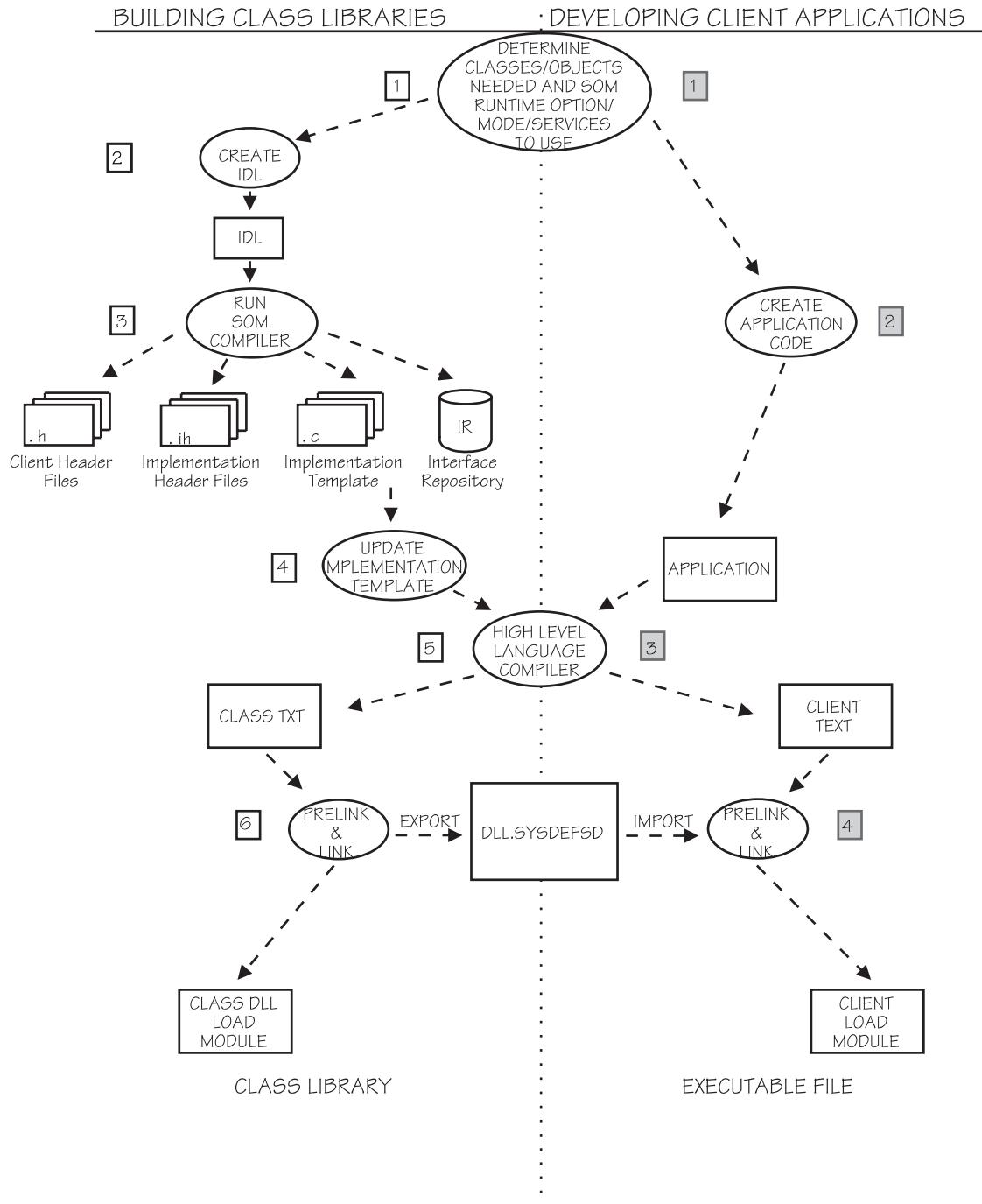


Figure 2-4. The SOMobjects “Big Picture” for a C application.

Building DLL class libraries with IDL includes the steps (tasks) on the left hand side of Figure 2-4. They include:

Step 1: Determine Which Classes/Objects are Needed and Which SOM Runtime Options/Services to Use

After determining which classes/objects are needed for your client application, you must then decide which runtime option/mode/services to use. SOMobjects includes a *runtime library*. The runtime library provides, among other things, a set of classes, methods, and procedures used to create objects and invoke methods on them. Not all classes need to be written from scratch. SOMobjects, and C++

provide some class libraries which can be used. Moreover, *SOM's runtime environment* allows applications to access information about classes dynamically (at runtime) and to use special SOMobjects services. (For additional information on Runtime, refer to Chapter 2, "Understanding SOMobjects Programming" on page 2-1.)

There are various services that are provided with SOMobjects. They include the:

- Emitter Framework
- Interface Repository (IR) Framework.

Emitter Framework: An emitter is a set of classes that is responsible for generating output forms. The Emitter Framework is a set of support classes and a general emitter class that can be subclassed to produce a specific emitter. For more information on emitters and how the Emitter Framework can be used, see Chapter 14, "Emitter Framework" on page 14-1.

The Interface Repository (IR): 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 IR contains objects that correspond to the major elements in IDL descriptions. The SOM IR framework is a set of classes that provide methods that executing programs can use to access these objects and discover everything known about the programming interfaces of SOM classes.

For more information on the IR, see Chapter 15, "The Interface Repository Framework" on page 15-1.

Step 2: Create Interface Definition Language (IDL) to Support Your Designed Classes

As previously noted, SOMobjects classes are designed to be language neutral. That is, SOMobjects classes can be implemented in one programming language and used in programs of another language.

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

Basic Components of IDL: IDL is made up of three basic components:

- *Interface declarations (or statements)*

An interface definition is specified within the interface declaration (or interface statement) of the IDL data set, which includes:

- the interface name (also known as class name) and the name(s) of the class's parent classes, and
- the names of the class's attributes and the signatures of its new methods. (Note that the complete set of available methods also includes all inherited methods.)

- *Method signatures*

Method signatures include the method name, and the type and order of its arguments, as well as the type of its return value (if any).

- *Attributes*

Attributes are IDL declarations that define instance variables for which "set" and "get" methods are automatically defined for use by the application program. (By contrast, instance variables that are not attributes are hidden from the user.)

For more information on IDL and how to use it, see Chapter 3, "SOMobjects Interface Definition Language (IDL)" on page 3-1.

Step 3: Compile Your IDL Datasets with the SOM Compiler, Creating Your Client Header File (Bindings)

Once the IDL source data set is complete, the SOM Compiler is used to analyze the IDL data set and create the *implementation template*, within which the class implementation will be defined. The implementation template contains *stub procedures* for each method of the class. It is up to the class implementor to fill in each stub with the body of the procedure. The template can be generated in different languages. The language that is generated is specified as a compile option.

.c is an example of a low level qualifier of a C implementation template file.

Before issuing the SOM Compiler command, the class library builder can set an environment variable that determines which *emitters* (output-generating programs) the SOM Compiler will call to create language specific *bindings*.

Bindings are language-specific macros and procedures that make implementing and using SOMobjects classes more convenient. These bindings offer a convenient interface to SOMobjects that is tailored to a particular programming language. For instance, C programmers can invoke methods written in C++ in the same way they make ordinary procedure calls in C.

In addition to the implementation template itself, the bindings include two language-specific header data sets that will be *#included* in the implementation template and in the client programs. The header data sets define many useful SOMobjects macros, functions, and procedures that can be invoked from the data sets that include the header data sets.

The following are the low level qualifiers of C binding files:

- .h (client header file)
- .ih (class implementation header file)

(Note that we are showing an example of C binding files. Binding files are language specific, so each language using the SOM Compiler will have different binding files.)

For more information on the SOM Compiler, see Chapter 4, “SOMobjects Compiler” on page 4-1.

Step 4: Update the Implementation Template File with Code to Make Methods Work

You now need to take the incomplete method procedures and write the code to make them work in the programming language of choice.

Steps 5 and 6: Compile the Implementation Code with the DLL Option, Prelink and Link

Compile the implementation code to create a standard object deck (or text file).

In the “Big Picture” example, the implementation code would be compiled using the C/C++ for OS/390 compiler with the DLL compile option.

After the method procedures have gone through the language compiler, the text file needs to be prelinked, linked and made into a dynamic link library (DLL) load module.

Note: The class programmer should refer to the appropriate documentation for information about creating DLLs. Each language will contain descriptions about what options to specify on the language compiler invocation for creating DLLs.

Developing Client Applications

The application (or client) program(s) use the objects and methods of the newly implemented class. A programmer could write an application program using a class implemented entirely by someone else. Before testing or running your client program, you should ensure that your client header files (or usage bindings) have been SOM-compiled (generated) 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).

Developing client applications is comprised of all those steps (tasks) on the right hand side of Figure 2-4 on page 2-13. They include:

Step 1: Determine which classes are needed and which SOM runtime option/services to use

Step 2: Create your application code

Step 3: Compile your client application program

Step 4: Prelink and link the object deck to create the client load module

Step 1: Determine Which Classes/Objects are Needed and Which SOM Runtime Option/Mode/Services to Use

Your first task is determining which classes (with associated objects, methods and data) are needed for your client application that can be used from the class libraries.

After determining which classes/objects are needed for you, you must then decide which runtime option/mode/services to use.

Runtime options include:

- Customizing memory management
- Processing character output
- Invoking methods (via language bindings or not)
- Using method resolution techniques.

Runtime services consist of the macros and functions that provide:

- Exception handling
- Error handling

See Chapter 2, “Understanding SOMobjects Programming” on page 2-1 and Chapter 11, “SOMobjects Advanced Topics” on page 11-1 for more information on runtime.

See *OS/390 SOMobjects Messages, Codes, and Diagnosis* for more information on exception and error handling.

Step 2: Create Your Application Code

This is the business logic that you code (including SOMobjects classes, methods and attributes that you import as is or *override*).

Step 3: Compile Your Client Application Program with Client Header Files

This step creates a standard object deck (or client text).

- For a COBOL client, ensure that the Interface Repository (IR) is populated. This can be done either by the class library builder or the client programmer by using the SOM Compiler option *-u -sir* or equivalently *-usir*.

Note: When the COBOL client compiles with the COBOL compiler, the client needs to have the SOM environment variable reference the appropriate IR file which points to the class definitions.

Refer to the appropriate C++ and COBOL publications for more information.

Step 4: Prelink and Link the Object Deck to Create the Client Load Module

After the method procedures have gone through the language compiler, the text file needs to be prelinked, linked and made into your client load module.

At this point, you can run your application. Figure 2-5 illustrates initialization of the SOMobjects runtime environment for your application. Your application will also call the class DLL you need and may use the Interface Repository. The user DLL will be accessed at this point and the IR may also be accessed.

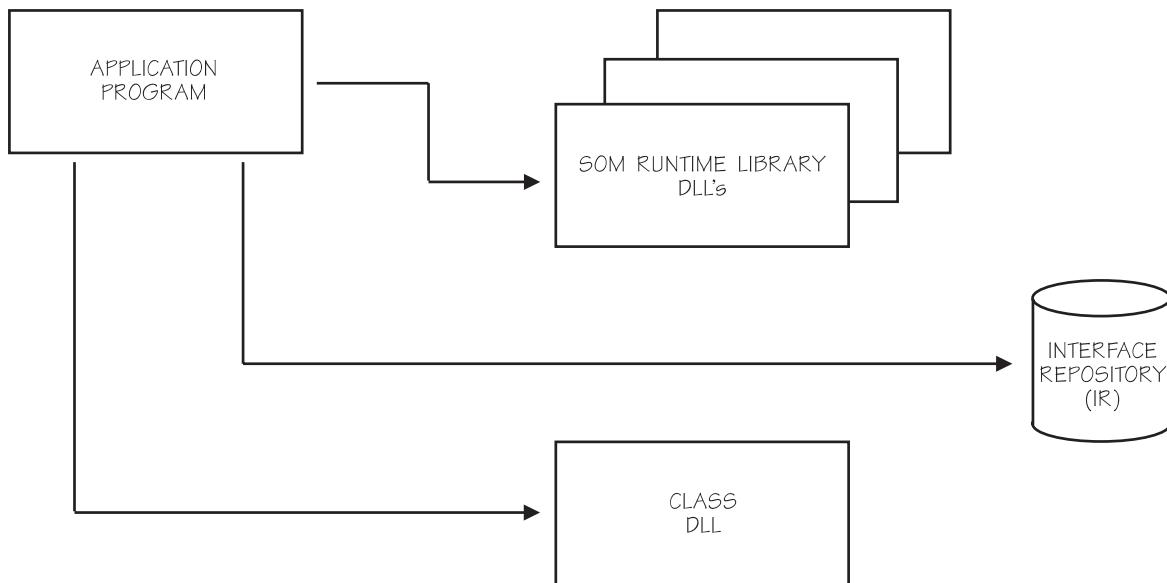


Figure 2-5. Your application and the runtime environment initialization process.

The SOMobjects runtime environment is automatically initialized the first time any SOM object is created. Programmers can initialize the runtime environment explicitly by calling the *somEnvironmentNew* function or invoke *SOM_MainProgram* macro (provided by the SOMobjects runtime library) before using any other SOMobjects functions or methods.

For more information on how to develop client applications using SOMobjects, see Chapter 6, “Developing Client Applications” on page 6-1.

Part 2. Building Class Libraries and Developing Client Applications

Chapter 3. SOMobjects Interface Definition Language (IDL)

As a class library builder, your next step after determining which classes are needed for your client application, and which SOMobjects runtime options/modes/services to use, is to create Interface Definition Language (IDL) datasets to support your classes.

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.

This chapter addresses the following topics:

- What is IDL?
- The uses and benefits of IDL
- Understanding IDL syntax (keywords, directives, and declarations)
- Using IDL (by building an example payroll application with two simple classes)

Figure 3-1 shows the part of the SOMobjects "Big Picture" process that this chapter describes in detail.

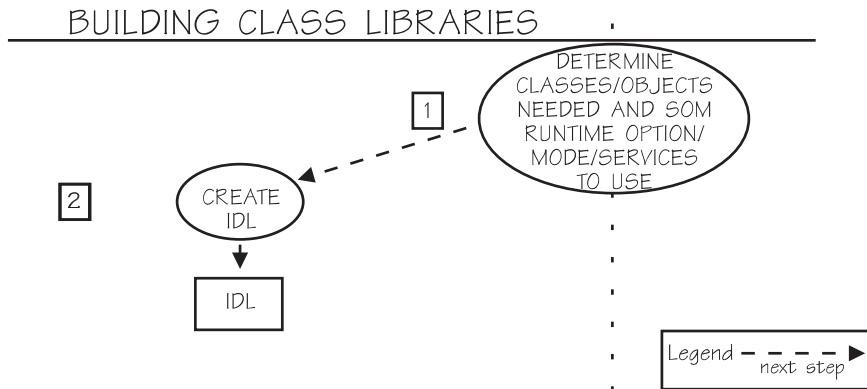


Figure 3-1. Steps 1 & 2 of the SOMobjects "Big Picture".

What is IDL?

IDL is a formal language used to describe object interfaces. Because, in SOMobjects, objects are implemented as instances of classes, an IDL object interface definition specifies for a class of objects what methods and attributes are available. Methods have return types and parameter types. We often speak of an IDL specification for a class (as opposed to simply an object interface).

Common Object Request Broker Architecture (CORBA) Compliant

IDL complies with CORBA's standard for IDL; it also adds constructs specific to SOMobjects. (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 SOMobjects IDL is given in Appendix B, "SOM IDL Language Grammar." This section describes the syntax and semantics of SOMobjects IDL using the following conventions:

Constants (words to be used literally, such as keywords) appear in **this font**.

User-supplied elements appear in *this font*.

{ } 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.

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 data set (or **.idl** data set)
- Method procedures written in the implementor's language of choice: the implementation file.

The interface definition data set has a **.idl** qualifier, 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** qualifier, and implementations written in C++ have a **.cxx** qualifier.

To assist users in implementing SOM classes, SOMobjects 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.

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 Uses and Benefits of IDL

IDL provides the following benefits for class implementers:

- IDL describes the interface for a class of objects

IDL allows class implementers to describe the interface for a class of objects which allows other applications to access this class of objects. The interface contains the names of the methods (return types and parameter types) and attributes it supports. After using the SOMobjects compiler on the IDL statements, the class implementers can then implement methods in their preferred programming language (where supported).

- IDL provides for compatible development platforms

SOMobjects applications are IDL-compatible with SOM applications written using the SOMobjects Developer Toolkit (on OS/2, AIX, or Windows). The IDL can be shipped to MVS for compilation. This provides a level of portability for SOM applications.

Understanding IDL Syntax (Keywords, Directives, and Declarations)

This section describes the lexical rules and syntax of SOMobjects's Interface Definition Language (IDL) and will give you a better understanding of how to code your IDL statements to build a class library. Once you understand the rules and syntax of IDL, you can move on to "Using IDL" on page 3-45 to create a simple "Payroll" application using IDL statements. The following topics need to be understood in order to use the IDL in creating class libraries:

- "Lexical Rules" on page 3-4
- "Keywords" on page 3-5
- "Include Directive" on page 3-5
- "Type and Constant Declarations" on page 3-6
- "Exception Declarations" on page 3-14
- "Interface Declarations" on page 3-17
 - Constant, type and exception declarations
 - Attribute declarations
 - Method declarations
- "Module Declarations" on page 3-24
- "Implementation Statements" on page 3-24
- "Comments within SOMobjects IDL" on page 3-38
- "Adding attributes with IDL" on page 3-39
- "Overriding a method using IDL" on page 3-40
- "Designating 'Private' Methods and Attributes" on page 3-40

- “Defining Multiple Interfaces in an IDL Data Set” on page 3-41
- “Scoping and Name Resolution” on page 3-41
- “Extensions to CORBA IDL Permitted by SOMobjects IDL” on page 3-43

This list looks as if IDL involves understanding many different and complicated topics before you can use it. But in general, an IDL specification may need no more than:

- A *#include* directive to tell the SOMobjects Compiler where to find the IDL for each of the class's parent classes and the class's metaclass
- An *interface* keyword to show where it is inheriting from
- An *attribute* keyword for each class attribute to be defined
- A method declaration defining the method's return type and input parameters for each class method to be defined.
- An *implementation statement* to specify information about how the class will be implemented.

Each of the above topics are discussed in greater detail in the following sections:

Lexical Rules

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

- White space is ignored except as token delimiters.
- C and C++ comment styles are supported.

For example, the following two comment styles are supported by C and by C++:

```
/* ... */
// ...
```

Whereas, the double hyphen:

```
--
```

form of a comment is not available to users of C/C++ for OS/390.

- IDL supports standard C/C++ preprocessing, including macro substitution, conditional compilation, and source data set inclusion.
- Identifiers (user-defined names for methods, attributes, instance variable and so forth) can be arbitrarily long sequences of alphanumeric and underscore characters, provided the first character is alphabetic.
- Identifiers must be spelled consistently with respect to case throughout a specification.
- Identifiers that differ only in case yield a compilation error.
- There is a single name space for identifiers (thus, using the same identifier for a constant and a class name within the same naming scope, for example, yields a compilation error).
- Integer, floating point, character, and string literals are defined just as in C and C++.

Keywords

The terms listed in Table 3-1 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.

Table 3-1. *Keywords for SOMobjects IDL*

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

Note: A typical IDL specification for a single class, residing in a single IDL data set, may have directives and declarations (they are optional). (Also see the later section, “Defining Multiple Interfaces in an IDL Data Set” on page 3-41.) The order is unimportant, except that names must be declared (or forward referenced) before they are referenced.

Include Directive

The IDL specification for a class normally contains **#include** statements that tell the SOMobjects compiler where to find the interface definitions (the IDL data sets) for:

- Each of the class's parent classes, and
- The class's metaclass (if one is specified)

The **#include** statements must appear in the above order. For example, if class “C” has parents “foo” and “bar” and metaclass “meta”, then data set “myclasses.IDL(c)” must begin with the following **#include** statements:

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

As in C and C++, if a data set name is enclosed in angle brackets (< >), the search for the data set will begin in system-specific locations. If the data set name appears in double quotation marks (“ ”), the search for the data set 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 SOMobjects bindings):

- Integral types
- Floating point types
- Character type
- Boolean type
- Octet type
- Any type
- Constructed types
 - struct
 - union
 - enum
- Template types
 - Sequences
 - Strings
- Arrays
- Pointers
- Object types

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 format supplied by S/390 single-precision floating-point numbers; double represents the S/390 double-precision floating-point numbers.

Character Type

IDL supports a **char** type, which represents an 8-bit quantity. 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 SOMobjects 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. See Chapter 15, “The Interface Repository Framework” on page 15-1 for more information about Typecode.

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 as 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 incorrect 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:

Valid IDL struct and enum definitions: The following IDL **struct** and **enum** definitions are valid:

```
struct myStruct {
    long x;
    double y;
};

/* defines type name myStruct*/

enum employee_types { hourly, salary, commission }; /* defines type name emptypes */
```

Not valid IDL struct and enum definitions: By contrast, the following definitions are *not* valid:

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

typedef enum employee_types { hourly, salary, commission} employee_types;
/* NOT VALID */
```

Valid IDL struct and enum definitions translated by the SOMobjects Compiler:

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

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

typedef unsigned long Payroll_employee_types; /* C/C+ bindings for IDL enum */
#define Payroll_hourly 1UL
#define Payroll_salary 2UL
#define Payroll_commission 3UL
```

Note: To see the entire IDL of the “Payroll” example, see Figure 8-2 on page 8-8.

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 *employee_types* enum, above, were defined within the interface statement for class *Payroll*, then the enumeration names would be referenced as *Payroll_hourly*, *Payroll_salary*, and *Payroll_commission*.

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

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

That specification would generate the following three types:

```

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

```

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

```

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

```

In the C/C++ bindings of the preceding fragment, you can refer to “M_long_t” as “long_t”, to “M_N_short_t” as “short_t”, and to “M_N_I_char_t” as “char_t”.

However, these shorter forms are available *only* when their interpretation is not ambiguous. Thus, in the first example the shorthand for “M_N_I_long_t” would not be allowed, since it clashes with “M_long_t” and “M_N_long_t”. If these shorter forms are not required, they can be ignored by setting #define SOM_DONT_USE_SHORT_NAMES before including the public header data sets, or by using the SOMobjects Compiler option -mnoabbrev so that they are not generated in the header data sets.

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

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

```

union identifier switch ( switch-type )
{ case+ }

```

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

```

case-label+ type-spec declarator ;

```

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

```
case const-expr:  
default:
```

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

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

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

is mapped onto the following C struct:

```
typedef struct Payroll_Foo {  
    long _d;  
    union {  
        long x;  
        float y;  
        char z;  
    } _u;  
} Payroll_Foo;
```

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

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

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

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 runtime).

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

```
unsigned long _maximum;
unsigned long _length;
simple-type *_buffer;
```

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

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

results in the following C **struct**:

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

and an instance of this type is declared as follows:

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

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

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

C and C++ programs using SOMobjects'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;
```

Strings: 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 SOMobjects's C and C++ bindings, **strings** are mapped onto zero-byte terminated character arrays. The length of the string is encoded by the position of the zero-byte. For example, the following IDL declaration:

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

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

```
typedef char *foo;
```

A variable of this type is then declared as follows:

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

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

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

Arrays

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

```
identifier {[ positive-integer-const ] }+
```

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

Pointers

Although the CORBA standard for IDL does not include them, SOMobjects 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.

Within SOMobjects's C++ bindings, the pointer is made explicit, and the use of an interface name as a type refers to a class instance itself, rather than a pointer to a class instance. For example, to declare a variable “myobj” that is a pointer to an instance of class “Foo” in an IDL specification and in a C program, the following declaration is required:

```
Foo myobj;
```

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

```
Foo *myobj;
```

If a C programmer uses the SOMobjects 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 because it more closely resembles the bindings for C++, thus making it easier to change to the C++ bindings at a later date.

Note: The C and C++ binding emitters should *not* be run in the same SOMobjects Compiler command. For example, do not issue the following command from a TSO READY:

```
sc -V -sh:xh 'dataset_stem.idl(member)'
```

where

dataset_stem represents the data set name qualifiers that precede the IDL qualifier;
.idl must appear as the last qualifier before the member name.

member represents the name of the member of the partitioned data set that actually contains the source IDL code.

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

```
sc -V -sh 'dataset_stem.idl(member)'  
sc -V -sxh 'dataset_stem.idl(member)'
```

For more information on SOMobjects Compiler commands and options, see Chapter 4, “SOMobjects Compiler” on page 4-1. For more information on the naming conventions of IDL datasets, see “General Application Development Considerations” on page 8-4 and “Step 1. Determine Classes/Objects Needed and SOM Runtime Option/Mode/Services to use” on page 8-4.

Exception Declarations

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

```
exception identifier { member* };
```

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

```
type-spec declarators ;
```

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

If an **exception** is returned as the outcome of a method, the exception “identifier” indicates which exception occurred. The values of the members of the exception provide additional information specific to the exception. The topic “Method Declarations Within an Interface Declaration” on page 3-20 describes how to indicate that a particular method may raise a particular exception, and the section on “Handling Exceptions” in *OS/390 SOMobjects Messages, Codes, and Diagnosis* describes how exceptions are handled.

There are two types of exceptions:

- BAD_FLAG exception declaration
- Exception declaration within an interface

“BAD_FLAG” Exception Declaration

Following is an example declaration of a “BAD_FLAG” exception:

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

The SOMobjects 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.

Exception Declaration Within an Interface

An exception declaration within an interface “Payroll”, such as this:

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

would map onto:

```
#define ex_Payroll_LOCAL_EXCEPTION "::Payroll::LOCAL_EXCEPTION"  
typedef struct Payroll_LOCAL_EXCEPTION {  
    long ErrCode;  
} Payroll_LOCAL_EXCEPTION;  
#define ex_LOCAL_EXCEPTION ex_Payroll_LOCAL_EXCEPTION
```

Standard Exceptions Defined by CORBA

In addition to user-defined exceptions, there are several predefined exceptions for system runtime errors. The standard exceptions as prescribed by CORBA are shown in Table 3-2 on page 3-17. These exceptions correspond to standard runtime 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, **SOMMVS.SGOSH.H(SOMCORBA)** (included by **SOMMVS.SGOSH.H(SOM)**) defines a generic **StExcep** typedef which can be used instead of the specific typedefs:

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

The standard exceptions shown in Table 3-2 are defined in an IDL module called **STEXCEP**, in the data set called **SOMMVS.SGOSIDL.IDL(stexcep)**, and the C definitions can be found in **SOMMVS.SGOSH.H(stexcep)..**

Table 3-2. Standard Exceptions Defined by CORBA

```
module StExcep {

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

enum completion_status { YES, NO, MAYBE };

enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};

exception UNKNOWN
exception BAD_PARAM
exception NO_MEMORY
exception IMP_LIMIT
exception COMM_FAILURE
exception INV_OBJREF
exception NO_PERMISSION
exception INTERNAL
exception MARSHAL
exception INITIALIZE
exception NO_IMPLEMENT
exception BAD_TYPECODE
exception BAD_OPERATION
exception NO_RESOURCES
exception NO_RESPONSE
exception PERSIST_STORE
exception BAD_INV_ORDER
exception TRANSIENT

exception FREE_MEM
exception INV_IDENT
exception INV_FLAG
exception INTF_REPOS

exception CONTEXT
exception OBJ_ADAPTER
exception DATA_CONVERSION
exception TRANSACTIONREQUIRED
exception TRANSACTIONROLLEDBACK
exception INVALIDTRANSACTION
exception WRONGTRANSACTION

};

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
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
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
exception TRANSACTIONREQUIRED    ex_body; // operation requires transaction
exception TRANSACTIONROLLEDBACK   ex_body; // current transaction has rolled
exception INVALIDTRANSACTION     ex_body; // transaction invalid or invalid state
exception WRONGTRANSACTION      ex_body; // reply received for wrong transaction
};
```

The standard exceptions shown in Table 3-2 are defined in an IDL module called **StExcep**, in the data set called **SOMMVS.SGOSIDL.IDL(STEXCEP)** and the C definitions can be found in **SOMMVS.SGOSH.H(STEXCEP)**.

Interface Declarations

The IDL specification for a class of objects must contain a declaration of the **interface** these objects will support. Because, in SOMobjects, 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 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 inter-

face indicates new methods, 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 implementers 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 data set). A parent class cannot be named more than once in the **interface** statement header.

Note: In general, an “**interface** class-name” header must precede any subsequent forward declarations or references to the “class-name”. For more discussion of multiple **interface** statements, refer to the later topic “Defining Multiple Interfaces in an IDL Data Set” on page 3-41.

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++.

The following declarations can be found within an **interface** declaration:

- Constant, Type, and Exception declarations
- Attribute declarations
- Method declarations.

Constant, Type, and Exception Declarations Within the Body of an Interface Declaration

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 SOMobjects Compiler to the bindings 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 Tong_t;
#pragma somemittypes off
```

Note: Includes of other IDL files inside of somemittypes on/off to get global types should not be done as only types in the source IDL file will be emitted.

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 “Overriding a method using IDL” on page 3-40). To refer to a **constant**, **type**, or **exception** “name” defined by a parent class and redefined by “class-name,” use the “parent-name::name” syntax as before.

Note: A name reference such as MyParentClass::myType required in IDL syntax is equivalent to MyParentClass_myType in C/C++. For a full discussion of name recognition in SOMobjects, see “Scoping and Name Resolution” on page 3-41.

Attribute Declarations Within an Interface Declaration

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

Attributes are declared as follows:

```
[ readonly ] attribute
type-spec declarators ;
```

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

```

interface Employee: Payroll, SOMObject
{
    void hire();

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

```

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

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

```

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();

```

Attributes are inherited from ancestor classes (direct and indirect base classes). An inherited **attribute** name cannot be redefined to be a different type, nor can it be overridden.

Method Declarations Within an Interface Declaration

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

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.

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 techniques, as in Name-Lookup method resolution).

The following subtopics describe the remaining syntax of a method declaration:

- Oneway keyword
- Parameter list

- Examples of valid method declarations in SOMobjects IDL
- Raises expression
- Context expression.

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**). A method must not modify an **in** parameter. If a method raises an exception, the values of the return result and the values of the **out** and **inout** parameters (if any) are undefined. When an unbounded **string** or **sequence** is passed as an **inout** parameter, the returned value must be no longer than the input value.

For example, let's use the analogy of C/C++ for OS/390 *passing parameters by value* and *passing parameters by reference*.

Passing parameters by value: A program calls a function and passes an integer. When the function is entered, it gets a copy of the integer and changes its value. These changes are not seen by the calling program. Its integer variable remains unaffected by the function call. This is what IN does.

Passing parameters by reference: The same program calls a different function, passing a pointer to the integer. The function uses the pointer to update the integer's value. Since the caller and function are both pointing to the same integer now, the caller sees all the updates that the called function made to the integer. This is what OUT and INOUT do.

The way to tell whether a C/C++ function is using a parameter passed by value or by reference is to see if the parameter is a pointer.

Examples of Valid Method Declarations in SOMobjects IDL: The following are examples of valid method declarations in SOMobjects IDL:

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

Classes derived from SOMObject can declare methods that take a variable number of arguments through a final parameter of type **va_list**. The **va_list** must use the parameter name “ap”, as in the following example:

```
void MyMethod(in short numArgs, in va_list ap);
```

For **in** parameters of type **array**, C and C++ clients must pass the address of the first element of the array, rather than the array itself. For **in** parameters of a **struct** or **union** type, C/C++ clients must pass the address of a variable of that type, rather than the variable itself.

Note: If you are going to build **va_list** manually and pass it to SOM methods, you will have to do a **#define _VARARG_EXT_** before the include of **stdio.h** or **stdarg.h**.

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 SOMobjects’s C and C++ bindings. Thus, the stub procedure that the SOMobjects 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**” (see “Template Types (Sequences and Strings)” on page 3-10) member of the struct that represents the **sequence**, or for the “**_value**” member (see “Any Type” on page 3-7) of the struct that represents the **any** type.

Note: The foregoing description also applies for the `_get_<attributeName>` method associated with an attribute of type string, sequence, any, or array. Hence, the attribute should be specified with a "noget" modifier to override automatic implementation of the attribute's "get" method. Then, needed memory can be allocated by the developer's "get" method implementation and subsequently deallocated by the caller. (The "noget" modifier is described under the topic "Modifier statements" later in this section.)

Raises Expression: The optional **raises** expression ("raises-expr") in a method declaration indicates which exceptions the method may raise. (IDL exceptions are implemented by simply passing back error information after a method call, as opposed to the "catch/throw" model where an exception is implemented by a long jump or signal.) A **raises** expression is specified as follows:

raises (identifier1, identifier2, ...)

where each "identifier" is the name of a previously defined **exception**. In addition to the exceptions listed in the **raises** expression, a method may also signal any of the standard exceptions. Standard exceptions, however, should not appear in a **raises** expression. If no **raises** expression is given, then a method can raise only the standard exceptions. (See the earlier topic "Exception Declarations" on page 3-14 for information on defining exceptions and for the list of standard exceptions. See the section on "Handling Exceptions" in *OS/390 SOMobjects Messages, Codes, and Diagnosis* for information on 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` or `set_one_value` method of the Context class to set the context properties, or use the `get_default_context` method. The client program then passes the Context object in the method invocation. Note that the CORBA standard also allows properties in addition to those in the context expression to be

passed in the Context object. See Chapter 4, “SOMobjects Compiler” on page 4-1 for a discussion of how clients associate values with **context** identifiers.) A description of the **Context** class and its methods is contained in *OS/390 SOMobjects Programmer’s Reference, Volume 1*.

Naming consideration: Method level performance management by the OS/390 Work Load Manager (WLM) recognizes only the first 123 characters of the combination **module::interface**, and the first 123 characters of the method name. When defining a class, you should avoid naming collisions within the first 123 characters so that your interfaces can be uniquely identified to WLM, in the event method level performance management is being used. For more information, see the WLM chapter in *OS/390 SOMobjects Configuration and Administration Guide*.

Module Declarations

If multiple interfaces are defined in the same IDL data set, and the classes are not a class-metaclass pair, they can be grouped into modules. by using the following syntax:

```
module module-name { definition+ };
```

where each “definition” is a **type** declaration, **constant** declaration, **exception** declaration, **interface** statement, or nested **module** statement. Modules are used to scope identifiers. For more information on how modules are used to define multiple interfaces and to scope identifiers, see “Defining Multiple Interfaces in an IDL Data Set” on page 3-41 and “Scoping and Name Resolution” on page 3-41.

Implementation Statements

A SOMobjects 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 SOMobjects IDL (and is not part of the CORBA standard), the **implementation** statement should be preceded by an “#ifdef __SOMIDL__” directive and followed by an “#endif” directive.

The syntax for the implementation statement is as follows:

```
#ifdef __SOMIDL__
implementation
{
    implementation*
};
#endif
```

where each “implementation” can be a

- **Modifier** statement,
- **Passthru** statement, or a
- Declarator of an **instance variable**, terminated by a semicolon.

These constructs are described below. An **interface** statement may *not* contain multiple **implementation** statements.

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 **unqualified modifier** 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 SOMobjects 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;  
d11name = "food11";
```

A **qualified modifier** is associated with a qualifier (by connecting them with a colon). The qualified modifier has the syntax:

```
qualifier : modifier  
qualifier : modifier = value  
#pragma modifier qualifier : modifier  
#pragma modifier qualifier : modifier = value
```

where “qualifier” is the identifier of an IDL definition or is user defined. If the “qualifier” is an IDL definition introduced in the current interface, module, or global scope, then the modifier is attached to that definition. Otherwise, if the qualifier is user defined, the modifier is attached to the interface it occurs in. If a user-defined modifier is defined outside of an interface body (by using **#pragma modifier**), then it is ignored.

Example of an IDL Dataset with “modifier” & “qualified modifier”: Notice that qualified modifiers can be defined with the “qualifier” and “modifier[=value]” in either order. Also observe that additional modifiers can be included by separating them with commas.

Note: Some of these examples show that **somInit** and **somUninit** are overridden. These methods now execute under the overall control of the **somDefaultInit** method and the **somDestruct** method. When you, in your program, use the **somDefaultInit** method and the **somDestruct** method instead of the somInit and somUninit methods, then you must override them as well. See “Multiple Inheritance” on page 2-5 for more information.

```

#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 {
            #ifdef _SOMIDL_
            implementation {
                somInit : override;
            };
            #endif
        };
        interface I : SOMObject {
            void op ();
            #pragma modifier op : persistent;

            typedef char char_t;
            #ifdef _SOMIDL_
            implementation {
                releaseorder : op;
                metaclass = M_I;
                callstyle = oidl;
                mymod : a, b;
                mymod : c, d;
                e : mymod;
                f : mymod;
                op : persistent;
            };
            #endif
        };
    };
}

```

From the preceding IDL data set, we associate modifiers with the following definitions:

```

TypeDef "::newInt"           1 modifier: nonportable
InterfaceDef "::M::N::M_I"   1 modifier: override = somInit
InterfaceDef "::M::N::I"     9 modifiers: metaclass = M_I,
                             releaseorder = op
                             callstyle = oidl
                             mymod = a,b,c,d,e,f
                             a = mymod
                             b = mymod
                             c = mymod
                             d = mymod
                             e = mymod
                             f = mymod
OperationDef "::M::N::I::op" 1 modifier: persistent

```

Notice, how the modifiers for the user-defined qualifier "mymod":

```
mymod : a, b;  
mymod : c, d;  
e      : mymod;  
f      : mymod;
```

map onto:

```
mymod = a,b,c,d,e,f  
a      = mymod  
b      = mymod  
c      = mymod  
d      = mymod  
e      = mymod  
f      = mymod
```

This enables users to look up the modifiers with “mymod”, either by looking for “mymod” or by using each individual value that uses “mymod”. These user-defined modifiers are available for Emitter writers and can be located in the Interface Repository (see Chapter 15, “The Interface Repository Framework” on page 15-1).

SOMobjects Compiler Unqualified Modifiers: Unqualified modifiers include the SOMobjects Compiler-defined identifiers **abstract**, **abstractparents**, **baseproxyclass**, **callstyle**, **classinit**, **directinitclass**, **dllname**, **factory**, **filestem**, **functionprefix**, **majorversion**, **metaclass**, **memory_management**, **minorversion**, **somallocate**, and **somdeallocate** as described below:

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** is unspecified.

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 SOMobjects's C/C++ for OS/390 bindings will not include the CORBA-specified (*Environment *ev*) and (*context *ctx*) parameters.

classinit=procedure Specifies a user-written procedure that will be executed to initialize the class object when it is created. If the **classinit** modifier is specified in the IDL data set for a class, the SOMobjects compiler will provide a template for the procedure in the implementation data set it generates. The class implementer can then fill in the body of this procedure template.

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 more information, see "Initializer Methods" on page 11-8 and "Implementing Initializers" on page 11-13.

dllname=member or HFS filename Specifies the name of the *dynamic link library (DLL)* that will contain the class's implementation. The DLL name may reside in the MVS file system or the hierarchical file system (HFS). If the DLL is to be in an MVS partitioned dataset, then the **dllname** modifier is assigned the name of the member. If the DLL is to reside in an HFS file, then a file name should be assigned to the **dll** modifier. If the filename contains special characters (e.g. periods, backslashes), then the filename should be surrounded by double quotes. The filename specified can be either a full pathname, or an unqualified (or partially qualified) filename. The *name* specified in the **DLLNAME** field is written to the IR via the SOMobjects Compiler. At runtime, when a DLL is called to be loaded (via **somFindClass** or **somFindClssInFile**), the **SOMIR** environment variable is used to determine if an IR exists. If the **dllname** modifier for the class is found in the IR, it is retrieved and used as the name of the DLL to be loaded. To load a DLL at runtime be sure to invoke the application so that the PDS containing the member or the HFS file can be searched. For TSO or batch users, this may mean allocating the PDS to ISPLLIB, STEPLIB, or to a JOBLIB. For applications that run in the OpenEdition Shell, the path to the directory containing the HFS file should be one of the paths assigned to the **LIBPATH** environment variable.

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.

filestem=member or HFS filesystem Specifies how the SOM Compiler will construct file names for the files it generates. The file name that the emitters will use to create the emitted file name is derived from the input file. The default is for the **filestem** modifier value to be interpreted as a member

name (*my.h(<member>)*, *my.xc(<member>)*, etc.) if the input .idl file is an MVS PDS.

For example, the IDL is:

```
dataset_stem.idl(foo)
```

and the interface specifies the filestem modifier as

```
filestem=foobar
```

When the *h* and *ih* emitters are called by the SOMobjects Compiler, the following datasets will be created if they don't already exist:

```
dataset_stem.h(foobar)  
dataset_stem.ih(foobar)
```

If the input file is an HFS file, then the default is for the filestem modifier value to be treated as an HFS filestem (*<filestem>.h*; *<filestem>.xc*, etc.).

functionprefix=prefix Directs the SOMobjects Compiler to construct method-procedure names by prefixing method names with "prefix". For example, "functionprefix = xx;" within an **implementation** statement would result in a procedure name of xxfoo for method foo. The default for this attribute is the empty string. If an interface is defined in a module, then the default function prefix is the fully scoped interface name. *Tip:* Using a function prefix with the same name as the class makes it easier to remember method-procedure names when debugging.

When multiple classes are specified in the same IDL data set, function prefixes can be used to avoid name conflicts in the implementation data set. For example, if one class introduces a method and another class in the same data set overrides it, then the implementation data set for the classes will contain two method procedures of the same name (unless function prefixes are defined for one of the classes), resulting in a name collision at compile time.

majorversion=number Specifies the major version number of the current class definition. The major version number of a class definition usually changes only when a significant enhancement or incompatible change is made to the class. The "number" must be a positive integer less than $2^{32}-1$. If a non-zero major version number is specified, SOMobjects will verify that any code that purports to implement the class has the same major version number. The default major version number is zero.

metaclass=class Specifies the class's metaclass. The specified metaclass (or one automatically derived from it at runtime) will be used to create the class object for the class. If a **metaclass** is specified, its IDL data set (if separate) must be included in the **include** section of the class's IDL data set. If no metaclass is specified, the metaclass will be defined automatically.

minorversion=number Specifies the minor version number of the current class definition. The minor version number of a class definition changes whenever minor enhancements or fixes are made to a class. Class implementers 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, SOMobjects will verify that any code that purports to implement the class has the same or a higher minor version number. The default minor version number is zero.

somallocate=procedure Specifies a user-written procedure that will be executed to allocate memory for class instances when the somAllocate class method is invoked.

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

Example of unqualified interface modifiers: The following example illustrates the specification of unqualified interface modifiers:

```
implementation
{
    filestem = payroll;
    functionprefix = pay;
    majorversion = 1;
    minorversion = 2;
    classinit = payrollInit;
    metaclass = M_Payroll;
};
```

SOMobjects 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 SOMobjects Compiler-defined identifiers used as qualified modifiers, along with the IDL component to which it applies. Descriptions of all qualified modifiers are then given in alphabetical order. Recall that qualified modifiers are defined using the syntax *qualifier*. *modifier[=value]*.

For classes:

releaseorder

For attributes:

impldef_prompts, indirect, nodata, noget, and noset modifier

For methods:

**caller_owns_parameters, caller_owns_result, const,
maybe_by_value_parameters, maybe_by_value_result, init, method,
migrate, myplan namelookup, nocall, noenv, nonstatic, nooverride,
noself, offset, override, pass_by_copy_parameters,
pass_by_copy_result, procedure, reintroduce, and select.**

For variables:

staticdata

For types:

impctx, length, pointer, struct

The following paragraphs describe each qualified modifier.

caller_owns_parameters = "p1, p2, ..., pn"

Specifies the names of the method's parameters whose ownership is retained by (in the case of "in" parameters) or transferred to (for "inout" or "out" parameters) the caller. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense for parameters whose IDL type is a data item that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any), or a struct or union.

For parameters whose type is an object, ownership applies to the object reference rather than to the object (that is, the caller should invoke **release** on the parameter, rather than **somFree**).

caller_owns_result

Specifies that ownership of the return result of the method is transferred to the caller, and that the caller is responsible for freeing the memory. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense when the method's return type is a data type that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any). For methods that return an object, ownership applies to the object reference rather than to the object (that is, the caller should invoke **release** on the result, rather than **somFree**).

const

Indicates that implementations of the related method should not modify their target argument. SOM provides no way to verify or guarantee that implementations do not modify the targets of such methods, and the information provided by this modifier is not currently of importance to any of the SOMobjects emitters. However, the information may prove useful in the future. For example, since modifiers are available in the Interface Repository, there may be future uses of this information by DSOM.

impctx

Supports types that cannot be fully defined using IDL.

impldef_prompts

Indicates that the DSOM **regimpl/pregimpl** 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**.

indirect

Directs the SOMobjects Compiler to generate "get" and "set" methods for the attribute that take and return a pointer to the attribute's value, rather than the attribute value itself. For example, if an attribute *x* of type float is declared to be an indirect attribute, then the "_get_x" method will return a pointer to a float, and the input to the "_set_x" method must be a pointer to a float. (This modifier is provided for OIDL compatibility only.)

init

Indicates that a method is an initializer method. For information concerning the use of this modifier, see "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM."

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.

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.

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.

method or procedure

Indicates whether or not the method can be overridden. The **method** modifier indicates that the method can be overridden by subclasses. The **procedure** modifier indicates that the method cannot be overridden and that none of the normal method resolution mechanisms will be used to invoke it; it will be called directly. The default modifier is **method**.

migrate = ancestor

Indicates that a method originally introduced by this interface has been moved upward to a specified *ancestor* interface. When this is done, the method introduction must be removed from this interface (because the method is now inherited). However, the original **releaseorder** entry for the method should be retained, and **migrate** should be used to assure that clients compiled based on the original interface will not require recompilation. The ancestor interface is specified using a C-scoped interface name. For example, "Module_InterfaceName", not "Module::InterfaceName". See the later topic "Name Usage in Client Programs" for an explanation of C-scoped names.

namelookup

See "**offset** or **namelookup**."

nocall

Specifies that the related method should not be invoked on an instance of this class even though it is supported by the interface.

nodata

Directs the SOMobjects Compiler *not* to define an instance variable corresponding to the attribute. For example, a "time" attribute would not require an instance variable to maintain its value, because the value can be obtained from the operating system. The "get" and "set" methods for "nodata" attributes must be defined by the class implementer; stub method procedures for them are automatically generated in the implementation template for the class by the SOMobjects compiler.

noenv

Indicates that a direct-call procedure does not receive an environment as an argument.

noget

Directs the SOMobjects Compiler *not* to automatically generate a "get" method procedure for the attribute in the IH/XIH binding for the class. Instead, the "get" method must be implemented by the class implementer. A stub method procedure for the "get" method is automatically generated in the implementation template for the class by the SOMobjects compiler, to be filled in by the implementer.

nonstatic

See "**method or procedure**."

nooverride

Indicates that the method should not be overridden by subclasses. The SOMobjects 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 SOMObjects Compiler *not* to automatically generate a “set” method procedure for the attribute in the IH/XIH binding for the class.

Instead, the “set” method must be implemented by the class implementer. A stub method procedure for the “set” method is automatically generated in the implementation template for the class by the SOMObjects compiler.

Note: The “set” method procedure that the SOMObjects Compiler generates by default for an attribute in the H/XH binding (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 implementer should specify the **noset** attribute modifier and implement the attribute's “set” method manually, rather than having SOMObjects implement the “set” method automatically.

offset or namelookup

Indicates whether the SOMObjects 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. 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.

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 12-28 for more information.

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 12-28 for more information.

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 12-29.

procedure

See “**method** or **nonstatic** or **procedure**.”

reintroduce

Indicates that this interface will “hide” a method introduced by some ancestor interface, and will replace it with another implementation. Methods introduced as direct-call procedures or nonstatic methods can be reintroduced. However,

static methods (the default implementation category for SOM methods) cannot be reintroduced.

releaseorder: *a, b, c, ...*

Specifies the order in which the SOMobjects Compiler will place the class's methods in the data structures it builds to represent the class. Maintaining a consistent release order for a class allows the implementation of a class to change without requiring client programs to be recompiled.

The release order should contain every method name introduced by the class (private and nonprivate), but should not include any inherited methods, even if they are overridden. by the class. The "get" and "set" methods defined automatically for each new attribute (named "`_get_<attributeName>`" and "`_set_<attributeName>`") should also be included in the release order list. The order of the names on the list is unimportant except that once a name is on the list and the class has client programs, it should not be reordered or removed, even if the method is no longer supported by the class, or the client programs will require recompilation. New methods should be added only to the end of the list. If a method named on the list is to be moved up in the class hierarchy, its name should remain on the current list, but it should also be added to the release order list for the class that will now introduce it.

If not explicitly specified, the release order will be determined by the SOMobjects Compiler, and a warning will be issued for each missing method. If new methods or attributes are subsequently added to the class, the default release order might change; programs using the class would then require recompilation. Thus, it is advisable to explicitly give a release order.

select = parent

Used in conjunction with the **override** modifier, this indicates that an inherited static method will use the implementation inherited from the indicated *parent* class. The parent is specified using the C-scoped name. For example, "Module_InterfaceName", not "Module::InterfaceName". See the later topic "Name Usage in Client Programs" for an explanation of C-scoped names.

staticdata

Indicates that the declared variable is not stored within objects, but, instead, that the ClassData structure for the implementing class will contain a pointer to the staticdata variable. This is similar in concept to C++ static data members. The staticdata variable must also be included in the **releaseorder**. The class implementor has responsibility for loading the ClassData pointer during class initialization. This can be facilitated by writing a special class initialization function and indicating its name using the **classinit** unqualified modifier. Note: attributes can be declared as staticdata. This is an important implementation technique that allows classes to introduce attributes whose backing storage is not inherited by subclasses.

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 12-29.

Release Order Within PRIVATE Sections of IDL: There is a special condition that needs to be discussed with regard to releaseorder at this point. It involves having methods within PRIVATE sections of the IDL and having a client application using the public version of that IDL. If the releaseorder entries for the private

methods aren't included in the public IDL (at least as placeholders, if nothing else), the client application will experience erratic behavior from the instances of the class defined in that public IDL. The following is an example of adding a placeholder in the release order to allow a method to be PRIVATE,

```
#ifdef __PRIVATE__
    releaseorder: print_paycheck;
#else
    releaseorder: dummy1;
#endif /* __PRIVATE__ */
```

where dummy1 is the placeholder for the PRIVATE method print_paycheck.

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

```
implementation
{
    releaseorder : op1, op3, op2;
    op1 : persistent;
    somInit : override;
    mymod : a, b;
};
```

Declaring Instance Variables and Staticdata Variables

Declarators are used within the body of an **implementation** statement 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**. 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 vari-

ables 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, 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** data set 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** data sets that the SOM Compiler can #include with your new **.idl** data set, 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** data set.
- 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** data set
 - for the existing user-defined class
 - for the real TYPEDEF.

In the following example, the generic SOM type **somToken** is used in the **.idl** data set 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>"
        "";
    };
};
};
```

Comments within SOMobjects IDL

SOMobjects IDL supports both C and C++ comment styles. The characters “//” start a line comment, which finishes at the end of the current line. The characters “/*” start a block comment that finishes with the “*/”. Block comments can not be nested. The two comment styles can be used interchangeably.

Comments in a SOMobjects IDL specification must be strictly associated with particular syntactic elements, so that the SOMobjects Compiler can put them at the

appropriate place in the header and implementation data sets 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 8, “Non-Distributed SOMobjects Examples” on page 8-1.

Because comments appearing in a SOMobjects IDL specification are transferred to the data sets that the SOMobjects Compiler generates, and because these data sets are often used as input to a programming language compiler, it is best within the body of comments to avoid using characters that are not generally allowed in comments of most programming languages. For example, the C language does not allow “/*” to occur within a comment, so its use is to be avoided, even when using C++ style comments in the IDL data set.

SOMobjects IDL also supports throw-away comments. They may appear anywhere in an IDL specification, because they are ignored by the SOMobjects Compiler and are not transferred to any data set it generates. Throw-away comments start with the string “//#” and end at the end of the line. Throw-away comments can be used to “comment out” portions of an IDL specification.

To disable comment processing (that is, to prevent the SOMobjects Compiler from transferring comments from the IDL specification to the bindings it generates), use the **-c** option of the **sc** command when running the SOMobjects compiler (see Chapter 4, “SOMobjects Compiler” on page 4-1). When comment processing is disabled, comment placement is not restricted and comments can appear anywhere in the IDL specification.

Adding attributes with IDL

Declaring an attribute causes *get* and *set* methods to be automatically defined. For example, specifying:

```
attribute string msg;
```

causes SOM to generate the following two methods:

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

Thus, for convenience, an attribute can be used (rather than an instance variable) in order to use the automatically defined “get” and “set” methods without having to write their method procedures.

Overriding a method using IDL

To override the calc_pay method in “Payroll”, additional information must be provided in **sommvs.sgoosmpi.idl(payroll)** in the form of an **implementation** statement, which gives extra information about the class, its methods and attributes, and any instance variables.

In the expanded “Payroll” example IDL in Figure 3-3 on page 3-47, the “implementation” statements for classes salary_employee and hourly_employee introduce the **override** keyword for method calc_pay. The **override** keyword is a method *modifier*.

Here, calc_pay introduces a *modifier* for the (inherited) calc_pay method in the classes salary_employee and hourly_employee. Modifiers are like C/C ++ #pragma commands and give specific implementation details to the compiler. This example uses only one modifier, “override”. Because of the “override” modifier, when calc_pay is invoked on an instance of class salary_employee or hourly_employee, their implementation of calc_pay (in the implementation template) will be called, instead of the implementation inherited from the parent class, *employee*.

The “#ifdef __SOMIDL__” and “#endif” are standard C and C++ preprocessor commands that cause the “implementation” statement to be read only when using the SOMobjects IDL compiler (and not some other IDL compiler).

Designating ‘Private’ Methods and Attributes

To designate methods or attributes within an IDL specification as “private,” the declaration of the method or attribute must be surrounded with the preprocessor commands **#ifdef __PRIVATE__** (with two leading underscores and two following underscores) and **#endif**. For example, to declare a method “foo” as a private method, the following declaration would appear within the interface statement:

```
#ifdef __PRIVATE__
void foo();
#endif
```

Any number of methods and attributes can be designated as private, either within a single **#ifdef** or in separate ones.

When compiling an IDL data set, the SOMobjects Compiler normally recognizes only public (nonprivate) methods and attributes, as that is generally all that is needed. To generate header data sets for client programs that do need to access private methods and attributes, the **-p** option should be included when running the

SOMobjects Compiler. The resulting H or XH header data set will then include bindings for private, as well as public, methods and attributes. The **-p** option is described in the topic "Understanding the Syntax of the SOMobjects Compiler" on page 4-11.

SOMobjects also provides a Public Definition Language (PDL) facility that can be used with the SOMobjects Compiler to generate a copy of an IDL data set which has the portions designated as private removed. The next main section of this chapter describes how to invoke the SOMobjects compiler and the various emitters.

Defining Multiple Interfaces in an IDL Data Set

A single .idl file can define *multiple interfaces*. This allows, for example, a class and its metaclass to be defined in the same file. When a file defines two (or more) interfaces that reference one another, forward declarations can be used to declare the name of an interface before it is defined. This is done as follows:

```
interface class-name ;
```

The actual definition of the **interface** for "class-name" must appear later in the same .idl file.

If multiple interfaces are defined in the same .idl file, and the classes are not a class-metaclass pair, they can be grouped into modules, by using the following syntax:

```
module module-name { definition+ };
```

where each "definition" is a **type** declaration, **constant** declaration, **exception** declaration, **interface** statement, or nested **module** statement. Modules are used to scope identifiers (see below).

Alternatively, multiple interfaces can be defined in a single .idl file without using a module to group the interfaces. Whether or not a module is used for grouping multiple interfaces, the languages bindings produced from the .idl file will include support for all of the defined interfaces.

Note: When multiple interfaces are defined in a single .idl file and a **module** statement is not used for grouping these interfaces, it is necessary to use the **functionprefix** modifier to assure that different names exist for functions that provide different implementations for a method. In general, it is a good idea to always use the **functionprefix** modifier, but in this case it is essential.

Scoping and Name Resolution

An IDL data set 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.

Using unqualified names: 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.

Using qualified names: Qualified names are of the form:

scoped-name::identifier

For example, method name “meth” defined within interface “Test” of module “M1” would have the fully qualified name “M1::Test::meth.”

A qualified name is resolved by first resolving the “scoped-name” to a particular scope, then locating the definition of “identifier” within that scope. Enclosing scopes are not searched.

Qualified names of the form:

::identifier

are resolved by locating the definition of “identifier” within the smallest enclosing module.

Constructed Names in an IDL Specification: Every name defined in an IDL specification is given a global name, constructed as follows:

- Before the SOMobjects Compiler scans an IDL data set, 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. 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, 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;
    . . .
}

```

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 SOMobjects'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 in IDL identifiers.

Extensions to CORBA IDL Permitted by SOMobjects IDL

The following topics describe several SOM-unique extensions to the CORBA IDL specification that are permitted by SOMobjects IDL. These constructs can be used in an IDL data set without generating a SOMobjects Compiler error.

If you want to verify that an IDL data set contains only standard CORBA specifications, the SOMobjects Compiler option **-mcorba** turns off each of these extensions and produces compiler errors wherever non-CORBA specifications are used. (The SOMobjects Compiler command and options are described in the topic “Understanding the Syntax of the SOMobjects Compiler” on page 4-11.)

SOMobjects extensions of the standard CORBA IDL syntax are:

- Pointer ‘*’ types
- Unsigned types
- Implementation section
- Comment processing
- Generated header data sets.

Pointer ‘*’ Types

In addition to the base CORBA types, SOMobjects IDL permits the use of pointer types (‘*’). As well as increasing the range of base types available to the SOMobjects 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;  
    ...  
};
```

SOMobjects 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

SOMobjects IDL permits the syntax “*unsigned type*”, where *type* is a previously declared type mapping onto “short” or “long”. (Note that CORBA permits only “*unsigned short*” and “*unsigned long*”.)

Implementation Section

SOMobjects IDL permits an **implementation** section in an IDL **interface** specification to allow the addition of instance variables, method overrides, metaclass information, passthru information, and “pragma-like” information, called **modifiers**, for the emitters. See the topic “Implementation Statements” on page 3-24.

Comment Processing

The SOMobjects 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 data sets, C template data sets, documentation data sets, 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 data sets containing comments placed anywhere, you can use the compiler option **-c** or use “throw-away” comments throughout the IDL data set (that is, comments preceded by `//#`); as a result, no comments will be included in the output data sets.

Generated Header Data Sets

CORBA expects one header data set as output from an IDL file. However, SOMobjects IDL permits use of a class modifier, modifier, **filestem**, and a compile option **-d** that can be used to specify a different name for the generated header. (See “Understanding the Syntax of the SOMobjects Compiler” on page 4-11.)

Using IDL

Now that you understand what IDL is, let's look at an example “Payroll” application with two classes using IDL.

Building a “Payroll” Class Library with Two Classes

In this example, we'll be doing the following steps in building a class library:

- **Step 1:** Determine which classes (with associated objects, methods and data) are needed for your class library and which SOM runtime option/mode/services to use
- **Step 2:** Create Interface Definition language (IDL) datasets to support your classes

Step 1: Determine Which Classes Are Needed

Scenario: You have been asked by a human resources client to build a class library that is needed for a payroll application. After consulting with the client and analyzing the needs of the application, you've identified two reusable classes that can help simplify the payroll application's design. As part of the analysis you've identified the methods and attributes to be included in the interfaces of the two classes. The two classes that were identified are:

- **person**
- **employee**.

Note: To see the complete “Payroll” class library with all of its classes, attributes and methods, refer to “Payroll’ Example with IDL, SOM Compiled Output, Method Code, and Client Code” on page 8-2.

It's important to remember that SOMObject class is the root class of all classes and must be used in the creation of these new classes.

The **person** class has attributes of *name*, *address*, and *home_phone*.

The **employee** class has attributes of *work_phone*, *health_benefits* and *salary*. Its methods are *calc_pay*, *hire*, *fire*, and *promote*.

Figure 3-2 on page 3-46 shows the design of the “Payroll” class (**SOMObject**, **Person** and **Employee**).

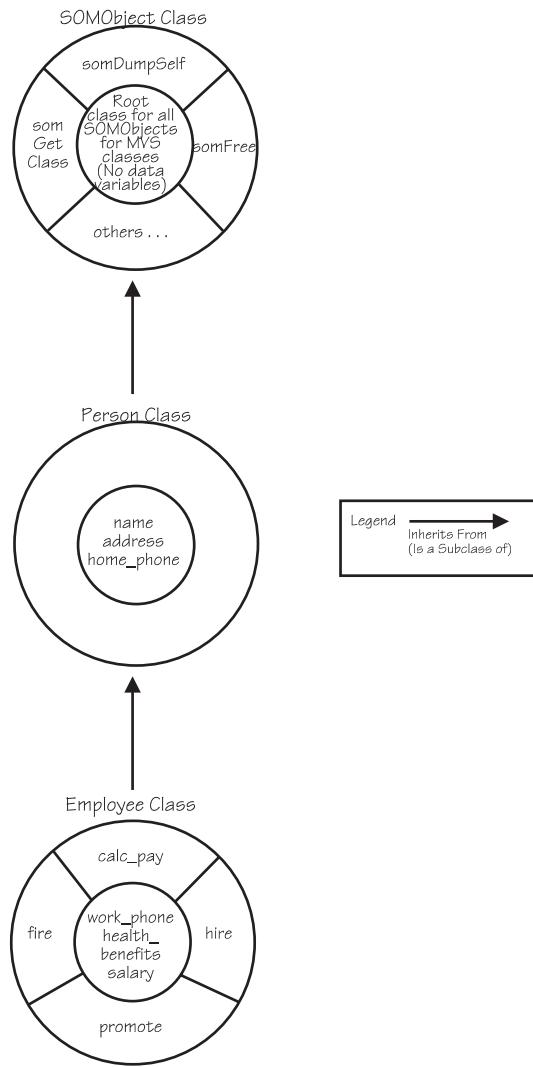


Figure 3-2. "Payroll" class design.

Step 2: Create Interface Definition Language (IDL) Datasets to Support Your Classes

You need to allocate a partitioned dataset before writing the IDL for this class. The name we'll be giving the dataset is `sommvs..SGOSSMPI.IDL(PAYROLLS)`.

Figure 3-3 on page 3-47 shows the IDL statements that define the Payroll class library, its classes and their associated methods and attributes. Note that the design done in Figure 3-2 has a one-to-one mapping with the class interfaces defined in Figure 3-3. IDL Keywords have been highlighted. The code for this example can be found in `sommvs.SGOSSMPI.IDL(PAYROLLS)`.

```

#include <somobj.idl>

module payroll {

    //# **** Person Class ****
interface person : SOMObject {
    //# Attributes
    attribute string name;
    attribute string address;
    attribute string home_phone;
#endif _SOMIDL_
    //# SOM Extentions to IDL
    implementation {
        releaseorder: _get_name, _set_name,
                      _get_address, _set_address,
                      _get_home_phone, _set_home_phone;
        somDestruct: override;
    };
#endif /* _SOMIDL__ */
}; /* end interface person */

    //# **** Employee Class ****
interface employee : person {
    //# Attributes
    attribute string work_phone;
    attribute int health_benefits;
    attribute float salary;
    //# Methods
    void calc_pay();
    void hire();
    void fire();
    void promote();
#endif _SOMIDL_
    //# SOM Extentions to IDL
    implementation {
        functionprefix = base_;
        releaseorder: _get_work_phone, _set_work_phone,
                      _get_health_benefits, _set_health_benefits,
                      _get_salary, _set_salary,
                      calc_pay,
                      hire, fire, promote;
    };
#endif /* _SOMIDL__ */
}; /* end interface employee */
}; /* end module payroll */

```

Figure 3-3. “Payroll” IDL statements.

Now that you've seen how a “Payroll” class library can be defined with IDL, let's move on to the SOMobjects Compiler and learn how to generate the implementation template and language bindings.

Chapter 4. SOMobjects Compiler

Your next step as a class library builder, after creating Interface Definition Language (IDL) datasets to support your classes, is to run these IDL datasets through the SOMobjects Compiler. This will produce the implementation templates, which you use to build your method code.

This chapter addresses the following topics:

- What is the SOMobjects Compiler?
- The uses and benefits of the SOMobjects Compiler
- Understanding the SOMobjects Compiler
 - Generating bindings
 - Tailoring the SOMobjects profile
- Using the SOMobjects Compiler
 - Setting up the SOMobjects Compiler to run from TSO, ISPF or batch
 - Setting up the SOMobjects Compiler with environment variables
 - Understanding the syntax of the SOMobjects Compiler
 - Using SOMobjects Compiler examples (including compiling the “Payroll” example created with IDL statements in “Building a “Payroll” Class Library with Two Classes” on page 3-45).
- The PDL Utility

Step 3 of Figure 4-1 shows the part of the SOMobjects “Big Picture” process that will be used in compiling the “Payroll” example.

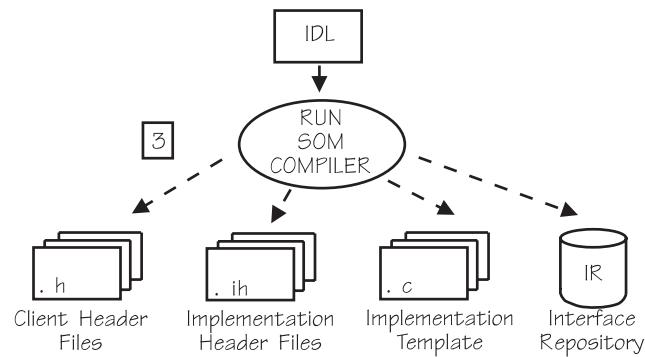


Figure 4-1. Step 3 of the SOMobjects “Big Picture”, showing C binding files.

What is the SOMobjects Compiler?

The SOMobjects Compiler translates the IDL definition of a SOMobjects class into a set of “binding data sets” (commonly referred to as “bindings”) appropriate for the language that will implement the class’s methods and the language(s) that will use the class. These bindings are necessary for programmers to implement and use SOM classes. The SOMobjects Compiler currently produces bindings for the C and C++ languages.

The Uses and Benefits of the SOMobjects Compiler

The SOMobjects Compiler provides the following benefits for class implementers:

- Creates language neutral binding files

Bindings are language-specific macros and procedures that make implementing and using SOMobjects classes more convenient. These bindings offer an interface to SOMobjects that is tailored to a particular programming language. For instance, C programmers can invoke methods in the same way they make ordinary procedure calls, likewise the C++ bindings “wrap” SOMobjects objects as C++ objects, so that C++ programmers can invoke methods on SOMobjects objects in the same way they invoke methods on C++ objects.

- It optionally allows, with the help of emitters, other output types, such as:
 - Class browsers
 - A text documenting the IDL interface
 - Interface Repository (IR) files

See Figure 4-2 for a view of the structure of the SOMobjects compiler and some of its potential outputs.

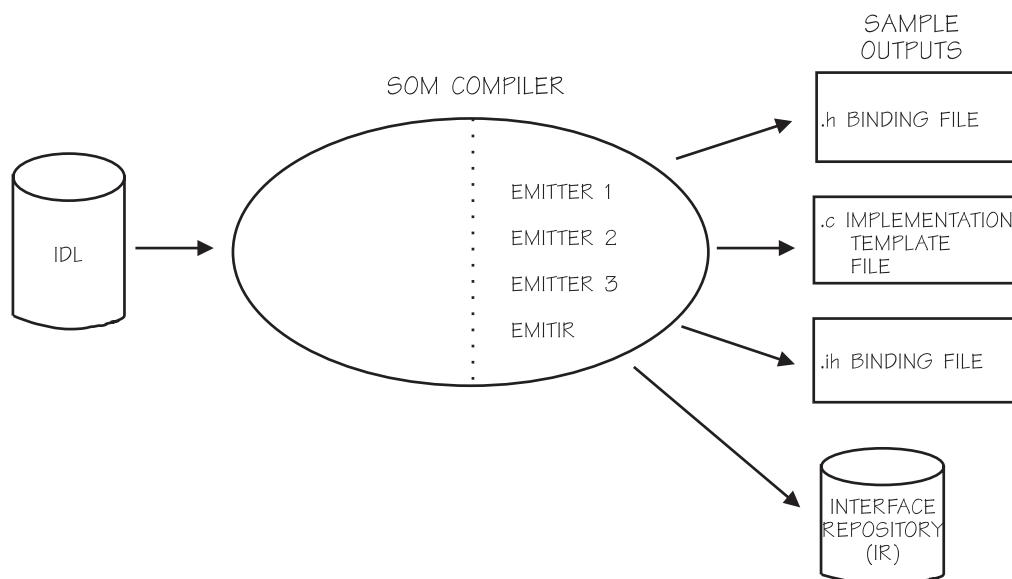


Figure 4-2. Structure of the SOMobjects Compiler with potential outputs.

Understanding the SOMobjects Compiler

Understanding the SOMobjects Compiler involves the following topics:

- Generating bindings
- Tailoring the SOMobjects profile

Generating Bindings

The SOMobjects Compiler operates in two phases:

- A precompile phase, in which a precompiler analyzes the IDL input.
- An emission phase, in which one or more emitter programs produce output files.

Each output file is generated by a separate emitter program. Setting the SMEMIT environment variable determines which emitters will be run by default. It can be overridden when the -s flag is used on the compiler command line to specify the emitters to be used. See “Understanding the Syntax of the SOMobjects Compiler” on page 4-11 for more information on the -s flag.

If changes to definitions in the IDL data set later become necessary, the SOMobjects Compiler should be rerun to update the current implementation template. For more information on generating updates, see “Running Incremental Updates of the Implementation Template” on page 8-23.

Note: In the discussion below, the emitted file name is by default based on the source IDL data set. For files emitted to the MVS file system, the *dataset_stem* and *member* are used from the input file name. For files emitted to the HFS, the *filestem* from the input file is used. The member name or file stem used for emitted output can be changed with the **filestem** modifier in “SOMobjects Compiler Unqualified Modifiers” on page 3-27. Also, the **-d** compiler option can be used to change the *dataset_stem* or path of the emitted output.

Emitters the SOMobjects Compiler generates: The following table provides an example of a source IDL data set and the output data sets generated when using different emitters.

Table 4-1. Example of Inputs and Outputs of the SOMobjects Compiler

Language	Emitter Used	Source IDL Data Set Name	SOM Compiler -s Options	Output Data Set Name	Type of Output
C	c	sgossmpl.idl(payroll)	c	sgossmpl.c(payroll)	implementation template
	ih		ih	sgossmpl.ih(payroll)	bindings for implementers
	h		h	sgossmpl.h(payroll)	bindings for client program
C++	xc	sgossmpl.idl(payroll)	xc	sgossmpl.cxx(payroll)	implementation template
	xih		xih	sgossmpl.xih(payroll)	bindings for implementers
	xh		xh	sgossmpl.xh(payroll)	bindings for client program

Note: The dataset name will default to the member name of the source IDL data set **only if** the dataset name is not modified in the IDL; in this example, the dataset name defaults to the member name.

The SOMobjects Compiler generates the following:

- C language bindings
- C++ language bindings

- DTS C++ language bindings (.hh)
- Public Definition Language (PDL) files
- Interface Repository (IR) files

The emitters for the C language produce the following bindings:

dataset_stem.c(member) or 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 bindings can be generated from the IDL file as needed, although the C source is not compilable without the h and ih bindings. If you're running the C emitter, you will probably run the h and ih emitters also.) This template 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 implementer has supplied the code for the method procedures, running the C emitter again will update the implementation data set to reflect changes made to the class definition (in the IDL file). These updates include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. Existing code within method procedures is not disturbed, however.

The C file contains an **#include** directive for the IH file, described below.

The contents of the C source template are controlled by the Emitter Framework file *sommvs.SGOSEFW*. This data set can be customized to change the template produced. For detailed information on changing the template, see Chapter 14, "Emitter Framework" on page 14-1.

dataset_stem.h(member) or 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 SOMobjects's generic C bindings, **som.h**.

dataset_stem.ih(member) or filestem.ih
(produced by the **IH** emitter)

This is the header data set to be included in the implementation data set (the data set that implements the class's methods—the C data set). 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 c template file.
- a **<className>NewClass** procedure for constructing the class object at runtime, and
- any IDL types and constants defined in the IDL interface.

The emitters for the C++ language produce the following bindings:

dataset_stem.cxx(member) or *filestem.C*
(produced by the **XC** emitter)

This is a template for a C++ source program that implements a class's methods. This will become the primary source data set for the class. (The other bindings can be generated from the IDL file as needed.) This template 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 implementer has supplied the code for the method procedures, running the **xc** emitter again will update this file to reflect changes made to the class definition (in the IDL file). These updates include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. Existing code within method procedures is not disturbed, however.

The C++ implementation data set contains an **#include** directive for the XIH data set, described below.

The contents of the C++ source template is controlled by the Emitter Framework file *sommvs.SGOSEFW*. This file can be customized to change the template produced. For detailed information on changing the template, see Chapter 14, "Emitter Framework" on page 14-1.

dataset_stem.xh(member) or *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**.

dataset_stem.xih(member) or *filestem.xih*
(produced by the **XIH** emitter)

This is the header data set to be included in the implementation data set (the data set that implements the class's methods). It contains the implementation bindings for the class, including:

- a **struct** defining the class's instance variables,
- macros for accessing instance variables,
- macros for invoking parent methods the class overrides,
- the **<className>GetData** macro
- a **<className>NewClass** procedure for constructing the class object at runtime, and
- any IDL types and constants defined in the IDL interface.

Other files the SOMobjects Compiler generates:

dataset_stem.hh(member) or *filestem.hh*
(produced by the **hh** emitter)

This file is a DirectToSOM C++ header file that describes a SOMobjects class in a way appropriate for

DTS C++. When running this emitter, you must include the **noqualifytypes** command-line modifier for the **-m** option of the SOMobjects Compiler command **sc** or **somc**.

dataset_stem.pdl(member) or **filestem.pdl**
(produced by the **PDL** emitter)

This data set is the same as the IDL data set from which it is produced except that all items within the IDL data set that are marked as “private” are removed. (An item is marked as private by surrounding it with “#ifdef __PRIVATE__” and “#endif” directives.) The PDL emitter can be used to generate a “public” version of an IDL data set. There is also a PDL Utility, which performs the same function, but can be invoked independently of the SOMobjects Compiler. For information on the PDL Utility, see “The Public Definition Language (PDL) Utility” on page 4-19.

dataset_stem.c(filestem/dll_filestemi)
(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 output source file is named based on the value of the **dllname** modifier in the implementation section of the **.idl** data set. Otherwise, the output source file is the value of the **idl** data set'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 3-25. and the **-m** option,

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:

sommvs.SGOSEFW(imod)

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 15, “The Interface Repository Framework” on page 15-1 for a discussion on the Interface Repository and how to use this emitter.

Naming conventions to keep in mind when naming your classes: The C/C++ bindings generated by the SOMobjects Compiler have the following limitation: If two classes named "ClassName" and "ClassNameC" are defined, the bindings for these two classes may clash if the attributes and/or methods of these classes have the same name.

This is because the binding files use macros to create short names for things like the attribute *get/set* methods and the objects methods.

For example, if class *ClassName* has an attribute *address* and class *ClassNameC* has an attribute *address*, both class' .h files will have a define for *_get_address* and *_set_address*. If both .h files are included in a client program, only the last one's macros will be valid.

But that's only a problem if you insist on using the short names. SOM supports invoking these methods by explicitly specifying the class name as the method name prefix, thereby avoiding the conflict. For example, you can use *ClassName_get_address* and *ClassNameC_get_address* to access the get accessor method for the *address* attribute of the desired class. Class implementers should keep this naming convention in mind when naming their classes.

Writing your own emitters: SOMobjects users can extend the SOMobjects Compiler to generate additional files by writing their own emitters. To assist users in extending the SOMobjects Compiler, SOMobjects provides an Emitter Framework which is a collection of classes and methods useful for writing object-oriented emitters that the SOMobjects Compiler can invoke. For more information, see Chapter 14, "Emitter Framework" on page 14-1.

Porting SOMobjects classes to another platform: The header data sets (bindings) that the SOMobjects Compiler generates will work only on the platform (operating system) on which they were generated.

If you are porting SOM classes (not SOMobjects classes) from another platform to MVS, you must regenerate the language bindings with the SOMobjects Compiler. If you are porting SOMobjects classes to another platform, then you must regenerate the language bindings with the SOM Compiler for that platform.

If the SOM class was implemented in C, you may need to use the SOMobjects Compiler to update the include statement for the .ih binding file. If you plan to use the SOMobjects Compiler from TSO or batch and the class implementation file is in an MVS PDS, then the SOMobjects compiler should be used to change any #include <name.ih> statements to #include <DD:IH(name)>. You can also use the SOMobjects Compiler when class development is moved from TSO or batch and MVS datasets to the OpenEdition Shell and the hierarchical file system (HFS). The SOMobjects Compiler will change any #include <DD:IH(name)> statements in a class implementation to #include <name.ih> when the class implementation is in an HFS file.

Note: For IBM COBOL for OS/390 and VM Version 2 Release 1 DTS clients, the IR must be populated. Therefore, either the class library builder or the client must ensure that the -u -sir, or the -usir option is used.

Tailoring the SOMobjects Environment Variables

Setting up the SOMobjects environment variables is similar to setting up environment variables on the AIX, OS/2 and DOS platforms.

Tailoring the SOMobjects profile consists of setting up and using the following types of variables:

- Global variables
- Local variables

Global Variables

SOMobjects provides variables that can be used system-wide for applications. These global variables are established as part of the configuration of the SOM subsystem, as documented in the *OS/390 SOMobjects Configuration and Administration Guide*.

Normally your installation will establish these global environment variable settings.

Local Variables

Local environment variables provide the means to:

- Create additional variables
- Override existing (global) variables
- Customize a group of variables for a specific application.

SOMobjects provides a sample environment variable file that can be used for applications.

The file (shipped as *sommvs.SGOSPROF(GOSENV)*) contains:

```
[somk]
  SMLANG=ENUS
[somc]
  SMINCLUDE=//'SOMMVS.SGOSIDL';//'SOMMVS.SGOSEFW';
[somir]
  SOMIR=//'SOMMVS.SGOSIR';//'IBMUSER.SAMPLES.SOMIR';
```

Using Local Variables: To use local variables, you must allocate a local profile to the DDNAME SOMENV. From TSO, for example, use the following command:

```
ALLOCATE DDNAME(SOMENV) DATASET('dataset name') SHR
```

Specify your own dataset name for the dataset name on the ALLOCATE command if you are using your own local variable file.

SOMobjects provides a local file which you can use or you can substitute your own. To use the SOMobjects file, specify 'SOMMVS.SGOSPROF(GOSENV)' for the dataset name on the ALLOCATE command.

For more information on environment variables, see Appendix A, "Setting up Configuration Files" on page A-1.

Using the SOMobjects Compiler

The SOMobjects Compiler can be set up and run in various ways. This section will describe:

- Setting Up the SOMobjects Compiler to run from TSO, ISPF, OpenEdition shell, or batch
- Setting Up the SOMobjects Compiler with environment variables
- Understanding the syntax of the SOMobjects Compiler
- Using SOMobjects Compiler examples (including compiling the “Payroll” example created with IDL statements in “Building a “Payroll” Class Library with Two Classes” on page 3-45).

Setting Up the SOMobjects Compiler to run from TSO, ISPF, OpenEdition Shell, or Batch

The SOMobjects Compiler can be run in various ways. It can be run from:

- TSO READY
- ISPF command line
- ISPF SOM Compiler option
- OpenEdition Shell
- Batch

TSO READY

To setup the SOM Compiler to run in TSO you can run the REXX exec found in Figure 8-7 on page 8-36. TSO examples can be found in “Building and Running a Non-Distributed Application from the TSO Environment” on page 8-39 and “Building and Running a Distributed Application from the TSO Environment” on page 9-4.

ISPF Command Line

To setup the SOM Compiler to run with the ISPF command line, you can run the REXX exec found in Figure 8-7 on page 8-36. Examples can be found in “Building and Running a Non-Distributed Application from the TSO Environment” on page 8-39 and “Building and Running a Distributed Application from the TSO Environment” on page 9-4.

ISPF SOM Compiler option

You may choose to use the ISPF panel interface to run the SOMobjects compiler. Contact your system administrator to see whether this option is available and if so, how to invoke the panel interface. For more information on setting up this ISPF option, see *OS/390 SOMobjects Configuration and Administration Guide*.

OpenEdition Shell

For information on how to setup and use the SOM compiler in the OS/390 OpenEdition environment, see

- *OS/390 SOMobjects Configuration and Administration Guide* for configuration information and
- “Building and Running a Non-Distributed Application from the OpenEdition Shell Environment” on page 8-37 and “Building and Running a Distributed Application from the OpenEdition Shell Environment” on page 9-2 for examples on how to run the SOM compiler in the OS/390 OpenEdition environment.

Batch

For information on how to setup and use the SOM compiler in the batch environment, see

- *OS/390 SOMobjects Configuration and Administration Guide* for configuration information and
- *OS/390 SOMobjects: Getting Started* for examples on how to run the SOM compiler in the batch environment.

Setting Up the SOMobjects Compiler with Environment Variables

To execute the SOMobjects compiler on one or more files that contain IDL specifications for one or more classes, use the following SOMobjects compiler command:

`sc [-options] file`

where *file* specifies one IDL file (or data set).

Available *-options* for the command are detailed in the next topic.

The operation of the SOMobjects compiler (whether it produces C bindings or C++ bindings, for example) can also be controlled by using environment variables that can be set in the environment variable data set.

Environment variables that affect the SOM compiler can be set for any *-m* options of the SOM compiler command.

Additional environment variables for other stanzas may be set to further direct the SOM compiler. For example, you may need to specify your own Interface Repository; you may do so by specifying the following environment variable in the [somir] stanza. The applicable environment variables, which would be listed in the [somc] stanza of the environment variable data set, are as follows:

SMEMIT

Determines which output files the SOMobjects compiler produces. Its value consists of a list of items separated by colons. Each item designates an emitter to execute.

The following line should be added to the global profile for installation wide changes, or to the local profile for user specific changes. For example, the statement:

`SMEMIT=c:h:ih`

indicates the C, H and IH emitters will be run. This directs the SOMobjects compiler to produce the C implementation template (*dataset_stem.c(payroll)*), and the C language bindings (*dataset_stem.h(payroll)* and *dataset_stem.ih(payroll)*) from the *dataset_stem.idl(payroll)* IDL.

By comparison:

`SMEMIT=xc:xh:xih`

indicates the XC, XH and XIH emitters will be run. This directs the SOMobjects compiler to produce the C++ implementation template (*dataset_stem.cxx(payroll)*), and the C++ language bindings (*dataset_stem.xh(payroll)* and *dataset_stem.xih(payroll)*) from the *dataset_stem.idl(payroll)* IDL.

By default, all output file's member names are the same as the IDL file's member name, unless changed in the IDL by using the *filestem* modifier. If the SMEMIT environment variable is not set, then a default value of H:IH: is assumed.

SMINCLUDE

Specifies where the SOMobjects compiler should look for IDL members #included by the IDL data set being compiled. The SMINCLUDE statement must start with the high level qualifier used at installation.

Note: The high level qualifier indicates where the SOMobjects compiler should look for your system's SOM-specific IDL data sets (*sommvs.SGOSIDL.IDL(SOMCLS)* and *sommvs.SGOSIDL.IDL(SOMOBJ)*).

The following line should be added to the global profile for installation wide changes, or to the local profile for user specific changes. For example, if SOMMVS is the high level qualifier, SMINCLUDE is set as:

```
SMINCLUDE='//SOMMVS.SGOSIDL';//'SOMMVS.SGOSEFW';
```

The high level qualifier may be different at your installation.

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 supersede and override the SMADDSTAR setting, however.

The following environment variable would be listed in the [somir] stanza of the environment variable data set, as follows:

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 SOMobjects compiler, the ir emitter also updates an existing Interface Repository.

Understanding the Syntax of the SOMobjects Compiler

The syntax of the **sc** command for running the SOMobjects compiler is:

```
sc [options] file
```

The “file” specified in the **sc** command denotes one file containing the IDL class definitions to be compiled. This file is the IDL data set. The names of the datasets emitted by the SOMobjects compiler are derived from the IDL dataset name unless the *filestem* modifier is used in the IDL or the *-d* option is used (see “Modifier Statements” on page 3-25).

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: The options are:

-C n Sets the maximum allowable size for a single comment in the .idl file (default 49152). This is only needed for very large single comments.

- c Turns off comment processing. This allows comments to appear anywhere within an IDL specification (rather than in restricted places) and causes comments not to be transferred to the output files that the SOMobjects compiler produces.
 - D *name[=def]* Defines *name* as in a #define directive. The default *def* is 1. This option is the same as the -D option for the C compiler. Note: This option can be used to define __PRIVATE__ so that the SOMobjects 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.
 - d *dataset_stem* Specifies the stem to be used by the SOMobjects compiler to form the name of the emitted files.
For example: If your IDL dataset was named "SOMMVS.SGOSSMPI.IDL(PAYROLL)" and you specified the compiler flag:
-d MY.PDS
your output c implementation template would be called:
<userid>MY.PDS.C(PAYROLL).
 - h or -? Displays a brief description of all SOM compiler options.
Note: The SOMobjects compiler ignores any options that appear after the -h option.
 - I *pds* When looking for #included files, look first in *pds*, then in the standard directories (same as the C compiler -I option).
 - i *filename* Specifies the name of the input file (IDL). Use this option to override the built-in assumptions about the input file. If the input file is an MVS PDS, the -i flag overrides the assumption that the input file ends with a .idl extension. For MVS input filenames, the -i flag allows the filename extension to be kept as part of the emitted file name.
 - m *name[=value]* Adds a global modifier.
Note: All command-line -m modifier options can be specified in the environment by changing them to UPPERCASE and prepending "SM" to them. For example, if you want to always set the options and "-maddstar", add the corresponding variables to your local profile as follows:

The currently supported global modifiers are:
-m*name[=value]* Adds a global modifier. (See also the following Note about the -m to an environment variable.)

All command-line -m modifieroptions 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
- Currently supported -m*name[=value]* modifier options:

addcmt	This option directs the ir emitter to emit IDL comments into the 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. The IR browser uses these modifiers to show the information associated with a class. For example, assume an operation is defined in an IDL file as follows:
	<pre>void testMethod(in string arg); // This is a comment.</pre>
	Specifying addcmt creates an OperationDef Class object in the Interface Repository with a comment modifier equal to the string <code>This is a comment.</code>
addprefixes	Adds ‘functionprefixes’ to the method procedure prototypes during an incremental update of the implementation template. This option applies only when rerunning the C or XC emitter on an IDL data set that previously did <i>not</i> specify a functionprefix. A class implementer who later decides to use prefixes should add a line in the <i>implementation</i> section of the IDL data set containing the specification:
	<pre>functionprefix = prefix</pre>
	(as described earlier in “Modifier Statements” on page 3-25) and then rerun the c or cxx emitter using the -maddprefixes option. The method procedure prototypes in the implementation data set will then be updated so that each method name includes the assigned prefix. (This option does not support changes to existing prefix names.)
addstar	This option causes all references to interfaces to have a '*' added to them for the C bindings. See “Object Types” on page 3-13 for further details.
comment=comment string	where <i>comment string</i> 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 SOMobjects compiler and will <i>not</i> be included in the output files.
	Note: Comments on lines beginning with “//#” are always ignored by the SOMobjects compiler.
corba	This option directs the SOMobjects 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 an IDL data set surrounded by #ifdef __SOMIDL__ and #endif directives are ignored. This option can be used to determine whether all nonstandard constructs (those specific to SOMobjects IDL) are properly protected by #ifdef __SOMIDL__ and #endif directives.

	Note: This option issues error messages only to warn users of non-CORBA compliant IDL data sets. It does not generate any bindings.
emitappend	This option causes the compiler to append each emitter's output to the output file if it already exists. This is useful for creating a single language binding file for an entire class library or collection of related classes if their IDL is split into multiple PDS members. Using emitappend on the compiles, combined with specifying the filestem modifier in each of the IDL members, will force the same output member name to be used for each IDL member's compile step.
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.
noheader	This option ensures that the SOMobjects compiler does not add a header to the beginning of an emitted file.
noint	This option directs the SOMobjects compiler not to warn about the portability problems of using int types in the source.
nopp	This option directs the SOMobjects compiler not to run the SOMobjects preprocessor on the IDL input file.
noqualifytypes	This option prevents the use of C-scoped names in emitter output, and is used in conjunction with the .hh emitter.
nouseshort	This option directs the SOMobjects 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. Passthru are added to InterfaceDef objects in the IR as modifiers whose value equals the passthru string, including line breaks. The modifiers are: <ul style="list-style-type: none"> • passthruC.h, passthruC.ih • passthruC.xh, passthruC.xih • passthruC.h_after, passthruC.ih_after • passthruC.xh_after, passthruC.xih_after The IR browser uses these modifiers to show the information associated with a class. See Chapter 15, "The Interface Repository Framework" on page 15-1 for a full discussion of the IR and the ir emitter.
tcconsts	This option directs the SOMobjects compiler to generate TypeCode constants in the H and XH public header files.
-p	Causes the "private" sections of the IDL data set to be included in the compilation (that is, sections preceded by #ifdef __PRIVATE__ that contain private methods and attributes). Note: The -p option is equivalent to the -D__PRIVATE__ option.
-r	Checks that all names specified in the release order statement are valid method names (default: FALSE).

- S n** Sets the total allowable amount of unique string space used in the IDL specification for names and passthru lines (default: 49152). This is only needed for very large IDL data sets.
 - s string** Substitutes *string* in place of the contents of the **SMEMIT** environment variable for the duration of the current invocation of the SOMobjects compiler. This determines which emitters will be run and, therefore, which output files will be produced.
- The **-s** option is a convenient way to override the **SMEMIT** environment variable. For example:
- ```
-sh:ih:c
```
- will cause the H, IH, and C emitters to be called.
- u** Updates the Interface Repository (default: no update). With this option, the Interface Repository will be updated even if the **ir** emitter is not explicitly requested in the **SMEMIT** environment variable or the **-s** option.
  - V** Displays version information about the SOMobjects compiler.
  - v** Uses verbose mode to display informational messages (default: FALSE). This option is primarily intended for debugging purposes and for writers of emitters.
  - w** Suppresses warning messages (default: FALSE).

## Using SOMobjects Compiler Examples

We'll now issue some SOM Compiler commands from a TSO READY screen using some of the above options.

**Example of a .h header dataset along with the version:** The following example of the **sc** command generates the **obj1** member for the **h** header data set and will give the version of the SOM Compiler. Displaying the **-V** option can be helpful if problem determination is later needed and you're asked for what version of the SOM Compiler you're using.

```
sc -V -sh myobjs.idl(obj1)
```

**Example of a C++ implementation template with the obj1 member:** The following example of the **sc** command generates the **obj1** member for the **C++** implementation template. Once the implementation template is created, you can proceed to implement the code to run your defined methods.

```
sc -sxc myobjs.idl(obj1)
```

**Example of a C implementation template with the obj1 member and .h and .ih header data sets:** The following example of the **sc** command generates the **obj1** member for the **C** implementation template and **h** and **ih** header data sets:

```
sc -sc:h:ih myobjs.idl(obj1)
```

**Example of a C template with obj1 member, .h, .ih headers (private):** The following example of the **sc** command generates the obj1 member for the **C** implementation template and **h** and **ih** header data sets, and lists the private header files. The **-p** option is important to someone who is building class libraries and wants to see the complete public and private portions of the header files.

```
sc -p -sc:h:ih myobjs.idl(obj1)
```

**Considerations when using the SOMobjects Compiler from the OpenEdition Shell:** If you are planning to develop SOMobjects applications completely within the OpenEdition Shell and using the hierarchical file system, then you should specify the SMOE environment variable under the [somc] stanza in your profile data set, located by the SOMENV file name:

```
[somc]
SMOE=YES
```

This environment variable tells SOMobjects to interpret any ambiguous file names as HFS files. (The default is for ambiguous file names to be interpreted as MVS data set names.) The export statement only needs to be done once per shell session.

An example of an ambiguous file name is my.idl. It does not contain any "/" characters to indicate if the file name is MVS or HFS. If you want use an unambiguous name for an MVS file, prefix the file name with "//", for example:

```
//my.idl(test)
```

If you want to use an unambiguous name for an HFS file, add the absolute or relative path to the file name:

```
/u/sombear/my.idl <--- absolute path name
./my.idl <--- relative path name
```

If you are using OS/390 file names from the OpenEdition Shell you may run into problems when using data set names that contain "" or "(" because these are considered special characters by the OpenEdition Shell. Use double quotes around the strings containing special characters to prevent the OpenEdition Shell from trying to process these characters. Suppose you want to use the SOMobjects com-

piler to compile an idl source file from an HFS file and direct the output to an MVS data set. The shell command to use would be similar to this:

```
sc -sh -d"//'sombear.toys'" teddy.idl
```

The binding would be in the MVS data set:

```
'sombear.toys.h(teddy)'
```

## Continuation of the “Payroll” Example

In the previous chapter, we created the IDL statements for our “Payroll” example.

We will now do a SOMobjects compile of the IDL dataset named *sommvs.SGOSSMPI.IDL(PAYROLLS)*, that we created in “Building a “Payroll” Class Library with Two Classes” on page 3-45.

This example is presented with two figures and one table:

- The first figure (Figure 4-3) is a display of a JCL batch job that contains an inline procedure to simplify running the SOMobjects Compiler.
- The second figure is the output from Figure 4-3.
- The table (Table 4-2 on page 4-19) displays the file mapping of the “Payroll” IDL to the C implementation file created with the SOMobjects compile.

**Batch job containing an inline procedure:** We discussed earlier that you have various options when running the SOMobjects Compiler on MVS, that is, TSO READY, ISPF command line, ISPF option 6 and by running a batch job. We'll choose to do this compile with a batch job.

Figure 4-3 shows the JCL job that was submitted with an inline procedure and *parm* statement.

```
/*PAYSC JOB <job card parameters>
/*
----- Start of JCL Procedure -----
//SC PROC INDSN=,
// SCPARMS='',
// SOMPRFX='SOMMVS',
// LEPRFX='CEE'
//SOMC EXEC PGM=SC,REGION=40M,
// PARM='(&SCPARMS ''&INDSN '')'
//STEPLIB DD DSN=BLEPRFX..SCEERUN,DISP=SHR
// DD DSN=&SOMPRFX..SGOSLOAD,DISP=SHR
//SOMENV DD DSN=&SOMPRFX..SGOSPROF(GOSENV),DISP=SHR
//SYSPRINT DD SYSOUT=*
// PEND
/*
----- End of JCL Procedure -----
//PAYSHORT EXEC SC,
// SCPARMS='-V -v -sh:ih:c -maddstar mnoint',
// INDSN=SOMMVS.SGOSSMPI.IDL(PAYROLL)
```

Figure 4-3. SOMobjects Compiler batch job containing an inline procedure.

**Output from the above batch job:** Figure 4-4 on page 4-18 displays the output from this batch job.

```

GOSCO057I SOMobjects Compiler: "SC", Version: 2.55.
GOSCO058I 5645-001 (C) Copyright IBM Corp. 1992,1997. All Rights Reserved.
GOSCO059I Licensed Materials - Property of IBM.
GOSCO060I US Government Users Restricted Rights - Use, duplication or
GOSCO061I Disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
GOSCO062I Date Last Modified: 6/7/96 s309624a
GOSCO063I Date Last Compiled: 15:06:54"

Processing 'SOMMVS.SGOSSMPI.IDL(payroll)'

GOSCO159I Running preprocessor against //SOMMVS.SGOSSMPI.IDL(payroll)

somipc -mpfppfile=/som.internal.cppout -v -V -m addstar -e emitchk -e emith -e emitih -e emitctm -e emitc -o
SGOSSMPI(PAYROLL) //SOMMVS.SGOSSMPI.IDL(payroll)
GOSCO057I SOM Pre-Compiler: "somipc", Version: 2.48.
GOSCO058I 5645-001 (C) Copyright IBM Corp. 1992,1997. All Rights Reserved.
GOSCO059I Licensed Materials - Property of IBM.
GOSCO060I US Government Users Restricted Rights - Use, duplication or
GOSCO061I Disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
GOSCO062I Date Last Modified: 6/7/96 s309624a
GOSCO063I Date Last Compiled: 15:10:17

GOSCO227I Running emitter: emitchk
"payroll", "payrolls_person", "payrolls_employee"
GOSCO227I Running emitter: emith
"payroll", "payrolls_person", "payrolls_employee"
GOSCO228I Emitter completed: emith Output Dataset=SGOSSMPI.h(PAYROLL)
GOSCO227I Running emitter: emitih
"payroll", "payrolls_person", "payrolls_employee"
GOSCO228I Emitter completed: emitih Output Dataset=SGOSSMPI.ih(PAYROLL)
GOSCO227I Running emitter: emitctm
"payroll", "payrolls_person", "payrolls_employee"
GOSCO227I Running emitter: emitc
"payroll", "payrolls_person", "payrolls_employee"
GOSCO228I Emitter completed: emitc Output Dataset=SGOSSMPI.c(PAYROLL)
"SGOSSMPI(PAYROLL)" GOSCO068I Removed //som.internal.cppout".

```

Figure 4-4. Output of the SOMobjects Compiler batch job.

**“Payroll” IDL to C implementation template file mapping:** Table 4-2 shows the .c implementation template that was created by the JCL inline procedure batch job and how it matches up with the IDL created in “Building a “Payroll” Class Library with Two Classes” on page 3-45.

**Note:** This example overrides **somUninit**. This method now execute under the overall control of the **somDestruct** method. See “Multiple Inheritance” on page 2-5 for more information.

Table 4-2. IDL statements for class “Payroll” and corresponding SOM Compiled .c implementation template.

| IDL Statements                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | SOM Compiled IDL Output (.c implementation template)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#include &lt;somobj.idl&gt;  module payroll {      //***** Person Class *****     interface person : SOMObject {         // Attributes         attribute string name;         attribute string address;         attribute string home_phone;     #ifdef __SOMIDL__         // SOM Extensions to IDL         implementation {             releaseorder: _get_name, _set_name,                         _get_address, _set_address,                         _get_home_phone, _set_home_phone;             somUninit: override;         };     #endif /* __SOMIDL__ */     }; /* end interface person */      //***** Employee Class *****     interface employee : person {         // Attributes         attribute string work_phone;         attribute int health_benefits;         attribute float salary;     #ifdef __SOMIDL__         // SOM Extensions to IDL         implementation {             functionprefix = base_;             releaseorder: _get_work_phone, _set_work_phone,                         _get_health_benefits, _set_health_benefits,                         _get_salary, _set_salary,                         calc_pay,                         hire, fire, promote;         };     #endif /* __SOMIDL__ */     }; /* end interface employee */ }; /* end module payroll */</pre> | <pre>??=ifdef __COMPILER_VER     ??=pragma filetag ("IBM-1047") ??=endif #pragma nosequence nomargins #ifndef _MVS #define _MVS #endif /*  * This file was generated by the SOM Compiler and  * Emitter Framework.  * Generated using template emitter:  *      SOM Emitter emitctm: 2.48  */ #define SOM_Module_payroll_Source #include &lt;dd:h(PAYROLL)&gt;  SOM_Scope void SOMLINK payroll_personsomUninit(     payroll_person *somSelf) {     payroll_personData *somThis =         payroll_personGetData(somSelf);     payroll_personMethodDebug(         "payroll_person", "payroll_personsomUninit"); }  SOM_Scope void SOMLINK base_calc_pay(     payroll_employee *somSelf, Environment *ev) {     payroll_employeeData *somThis =         payroll_employeeGetData(somSelf);     payroll_employeeMethodDebug(         "payroll_employee", "base_calc_pay");      SOM_Scope void SOMLINK base_hire(         payroll_employee *somSelf, Environment *ev)     {         payroll_employeeData *somThis =             payroll_employeeGetData(somSelf);         payroll_employeeMethodDebug(             "payroll_employee", "base_hire");          SOM_Scope void SOMLINK base_fire(             payroll_employee *somSelf, Environment *ev)         {             payroll_employeeData *somThis =                 payroll_employeeGetData(somSelf);             payroll_employeeMethodDebug(                 "payroll_employee", "base_fire");              SOM_Scope void SOMLINK base_promote(                 payroll_employee *somSelf, Environment *ev)             {                 payroll_employeeData *somThis =                     payroll_employeeGetData(somSelf);                 payroll_employeeMethodDebug(                     "payroll_employee", "base_promote");             }         }     } }</pre> |

## The Public Definition Language (PDL) Utility

As discussed earlier in this chapter, the SOMobjects Compiler provides a PDL emitter. This emitter generates a file that is the same as the IDL data set from which it is produced, except that it removes all items within the IDL data set that are marked as “private.” (An item is marked as private by surrounding it with “#ifdef \_\_PRIVATE\_\_” and “#endif” directives.) Thus, the PDL emitter can be used to generate a “public” version of an IDL data set. (Generally, client programs will need only the “public” methods and attributes.)

SOMobjects also provides a separate program, the PDL Utility, which performs the same function, but can be invoked independently of the SOMobjects Compiler.

The PDL utility can be used from TSO or batch.

## PDL Utility Syntax

The PDL Utility is invoked as follows:

```
pdl option dataset_stem.idl(member)
```

where

*option*        is one of the options specified in “PDL Utility Option.”

*dataset\_stem*   specifies an IDL partitioned data set name

*member*        specifies the member whose “private” sections are to be removed.  
Filenames must be completely specified (with the .idl extension).

## PDL Utility Option

The PDL Utility supports the following options. (Selected options can be specified individually, as a string of option characters, or as a combination of both. The option that takes an argument either must be specified individually or must appear as the final option in a string of option characters.)

**-d pds**        Specifies a partitioned dataset (PDS) in which the output files are to be placed.

**-h**              Requests this description of the pdl command syntax and options.

For example, there are two ways of creating an output file with the *-d* option:

- Take the default generated output dataset name
- Create your own output dataset name.

***Creating the default output dataset name without using the -d option:*** If you want to create a dataset named *hlq.pdl* (where *hlq* is the user-definable high level qualifier) using the default option, you issue the following command from the TSO READY line:

```
pdl myclass.idl(test)
```

You would create or update the following output dataset name:

```
myclass.pdl(test)
```

Dataset name *myclass.pdl(test)* would contain only public IDL.

***Creating your own output dataset name with the -d option:*** To create an output dataset with a name that you choose, you would use the **-d** option. You need to use the **-d** option prior to naming your output dataset name, followed by the input dataset name. You can do this by running the following command from the TSO READY line:

```
pdl -d public.idl(test) myclass.idl(test)
```

You would create or update the following output dataset name:

```
public.idl(test)
```

Note that *public.idl(test)* is the same name that you gave it in the output dataset name after the **-d** option.

Dataset name *public.idl(test)* would contain only public IDL.

Now that you've seen how the PDL Facility works and understand how the SOMobjects Compiler creates an implementation template, it's time to fill your implementation template and build the dynamic link library (DLL) for the "Payroll" example.



---

## Chapter 5. Building Class Libraries

Your next step as a class library builder, after SOMobjects compiling your interface definition language (IDL) datasets, is to create your method implementation code and create a class library where your client can access the new classes you've built.

This book deals primarily with building DLL class libraries, but there are two approaches to building class libraries. They are:

- Direct-to-SOM (DTS) build (building class libraries using DTS C++)
- Dynamic Link Library (DLL) build

We will quickly go over using a DTS build, before we go into greater detail on doing a DLL build of a class library.

---

### Direct-to-SOM (DTS) Build

DTS languages, such as C++/MVS and IBM COBOL for OS/390 and VM Version 2 Release 1, allow the class library application programmer to go directly to SOM for:

- Creation of SOM-enabled classes directly from C++ or COBOL classes. SOM-enabled classes have upward binary compatibility which allows new versions of a class library without requiring applications to recompile.
- Creation of Interface Definition Language (IDL) directly from C++ or COBOL classes. The IDL can be compiled with the SOM compiler to enable other programming languages to access the classes.

For the C++/MVS and COBOL programming languages, you can use the programming language compiler to create SOM-enabled classes without having to write IDL yourself. For C++/MVS, you can also use the C++/MVS compiler to generate IDL to make the classes and methods accessible to C or COBOL. For COBOL, you can use the COBOL compiler to generate IDL to make the classes and methods accessible to C or C++.

The tasks to create the SOM-enabled classes and IDL are performed separately in C++ and can be performed separately or as one combined task in COBOL.

The general steps for creating a DTS build in the C, C++ and COBOL classes follow. For examples of how to do a DTS build and other examples of language neutrality, see Chapter 10, "Language Neutrality with SOMobjects: Examples" on page 10-1.

#### **Creating SOM-enabled Classes for C++/MVS**

To create SOM-enabled classes for C++/MVS, you perform these general steps:

- Step 1. Make any changes that are necessary to make your C++ class library code SOM-compliant. You can use the SOM pragmas provided with C++.
- Step 2. Create a header or binding .hh file that declares the class rather than the usual .h file used for C++/MVS class declarations. For more information on header or binding files, see "Step 3: Compile Your IDL Datasets with

the SOM Compiler, Creating Your Client Header File (Bindings)" on page 2-15.

- Step 3. Compile the C++/MVS modified class library code (which should #include the .hh file) using the appropriate compiler options and pragmas to produce SOM-enabled class object files.

### **Creating C++/MVS Classes that are Accessible to C or COBOL**

To make a C++/MVS class accessible to C or COBOL, you must follow these general steps:

- Step 1. Make any changes that are necessary to make your C++ class library code SOM-compliant. You can use the SOM pragmas provided with C++.
- Step 2. Compile the .hh file itself with the C++/MVS compiler using the IDL option. The compiler produces an IDL source data set containing the class IDL definition.
- Step 3. SOM compile the IDL source data set.
- For a C client, compile the IDL source data set with the SOM compiler to produce the header file for C.
  - For a COBOL client, ensure that the Interface Repository (IR) is populated. This can be done either by the class library builder or the client programmer by using the SOM Compiler option *-u -sir* or equivalently *-usir*.

See the appropriate C/C++ for OS/390 documentation for details about performing these steps.

### **Creating SOM-enabled Classes for COBOL and Making COBOL Classes Accessible to C or C++**

To create SOM-enabled classes for COBOL and make a COBOL class accessible to C or C++, you perform these general steps:

**Note:** These steps show the combined task of creating the SOM-enabled classes and IDL in COBOL.

- Step 1. Compile the COBOL class definition source file with the COBOL compiler using the IDLGEN(ON) compiler option.
- Step 2. Compile the IDL source data sets with the SOM compiler, using the appropriate language emitters to create header files for C or C++.

Refer to the appropriate IBM COBOL for OS/390 and VM Version 2 Release 1 publications for details about performing these steps.

---

## **Dynamic Link Library (DLL) Build**

Dynamic Link Libraries (DLLs) are a collection of functions and variables that can be dynamically accessed from an external program or application. The rest of this section on "Building Class Libraries" will describe the process of creating a DLL build using IDL and consists of the following:

- What are DLLs?
- The uses and benefits of DLLs
- Understanding DLLs

- Using DLLs
  - Creating a DLL

Steps 4, 5, and 6 from the “Building Class Libraries” side of Figure 5-1 shows the part of the SOMobjects “Big Picture” process that will be used in updating the implementation template, running that implementation template through your language compiler which creates a text deck, and prelinking and linking that text deck into a DLL load module for the simple “Payroll” example.

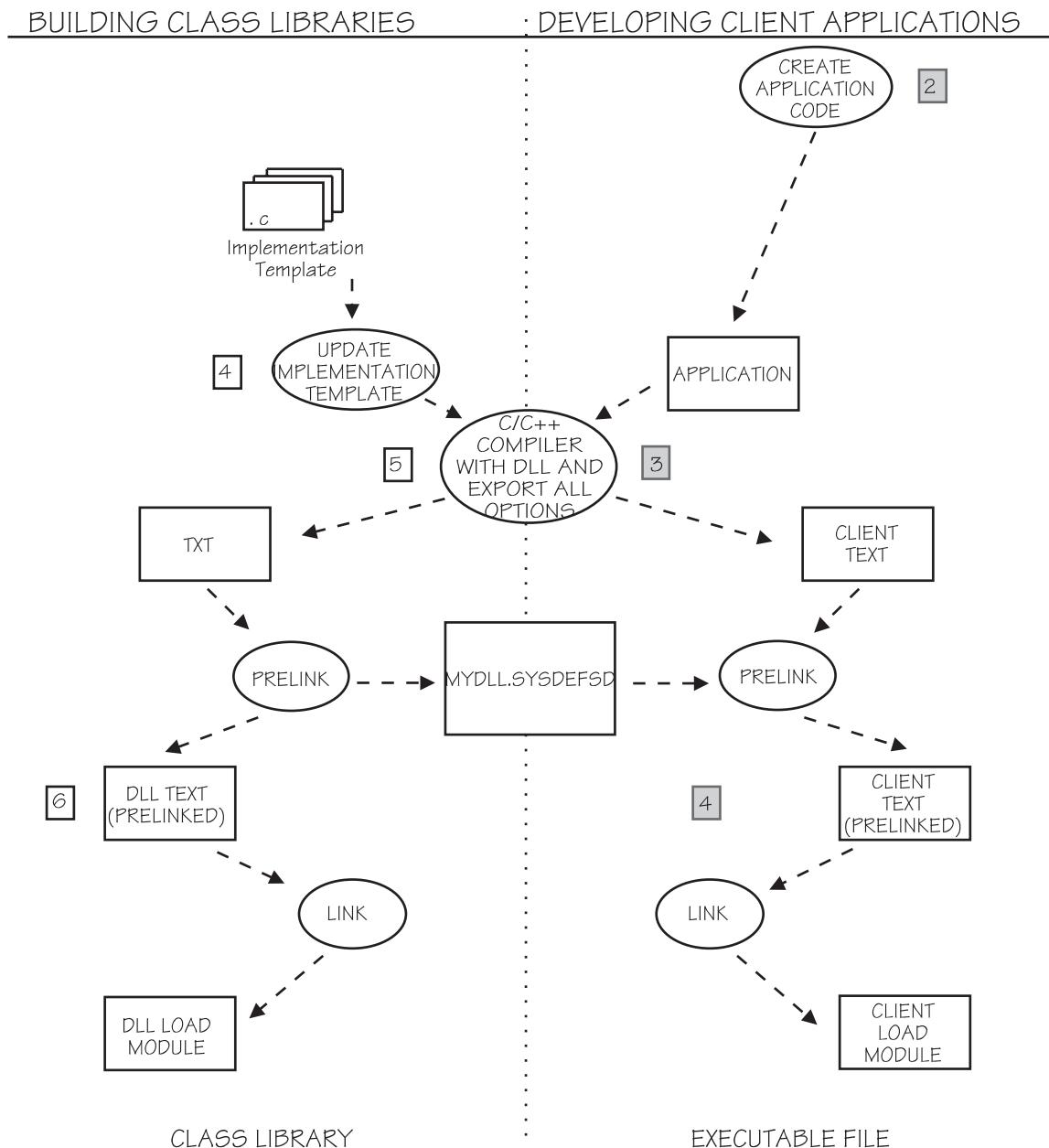


Figure 5-1. Steps for building and accessing DLLs.

## What are DLLs?

Dynamic link libraries (DLLs) are a collection of functions and variables that can be accessed from an external program.

**Note:** *Functions* are synonymous with *methods*.

- Individual DLL functions are not executable by themselves, that is, they:
  - Must be called
  - Must be reentrant (if C/C++, they don't have a *MAIN* in their code).
- DLL variables are accessed by references from applications.

## The Uses and Benefits of DLLs

Making classes and their methods and variables available through DLLs provides the following uses and benefits:

- Allows client applications the ability to call the functions within the DLL
- Identifies all functions, methods and attributes of the class (via the definition side deck)
- Eliminates the need to statically bind class libraries, such as the SOMobjects runtime, to client applications
- Saves system storage by loading a DLL only when needed by the client application.

## Understanding DLLs

The primary thing to understand about DLLs as a class library builder is how to build them. For information on how to access DLLs, see “Application Access to DLLs” on page 6-34.

### DLL Building

DLL building is a process whereby someone creating class libraries provides the functions and variables of the classes within a load module that can be accessed by client applications. They do this by:

- Specifying the DLL option on the compiler command invocation
- Note:** The DLL option is required for the C/MVS Compiler only. The C++/MVS Compiler always generates DLL-enabled text decks.
- Identifying specific functions and variables to be exported by using the *#pragma export* directive in the DLL source file or, alternatively by specifying EXPORTALL on the compiler command invocation
  - Prelinking the DLL object deck
  - Link-editing the prelinked object deck to create the DLL load module.

## Using DLLs

Now that you understand what a DLL is, let's continue creating the “Payroll” application that we've built in the previous two chapters.

A DLL can be set up and run in various ways. This section describes creating a DLL.

## **Creating a DLL**

Building a class library DLL consists of the six steps shown in the “Big Picture” in Figure 2-4 on page 2-13.

- Step 1 was covered in Chapter 2, “Understanding SOMobjects Programming” on page 2-1.
- Step 2 was covered in Chapter 3, “SOMobjects Interface Definition Language (IDL)” on page 3-1.
- Step 3 was covered in Chapter 4, “SOMobjects Compiler” on page 4-1.

This section will address Steps 4, 5 and 6 in completing a DLL, as seen in Figure 5-1 on page 5-3:

- **Step 4:** Update the .c implementation template with your method (DLL source) code
- **Step 5:** Compile the implementation code with the DLL option.
- **Step 6:** Prelink and link-edit the object deck (or text file) created by the C compiler.

## **Continuation of the “Payroll” Example**

The following section uses the “Payroll” example as we walk you through steps 4, 5 and 6.

### **Step 4: Update the Implementation Template with Your Method Code**

Taking the .c implementation template that was created in Table 4-2 on page 4-19, and adding the C method code to make the methods work, you would end up with the code shown in Figure 5-2 on page 5-6:

```

??=ifdef __COMPILER_VER_
 ??=pragma filetag ("IBM-1047")
??=endif

#pragma nosequence nomargins
#ifndef _MVS
#define _MVS
#endif

/*
 * This file was generated by the SOM Compiler.
 * Generated using:
 * SOM incremental update: <current version>
 * Timestamp: <current time stamp>
 *
 */

#define SOM_Module_payroll_Source
#include <DD:IH(PAYROLL)>

/* user required header files */
#include <stdio.h>
#include <stdlib.h>

SOM_Scope void SOMLINK \
payroll_personSomUninit(
 payroll_person *somSelf)
{
 payroll_personData *somThis =
 payroll_personGetData(somSelf);
 payroll_personMethodDebug(
 "payroll_person",
 "payroll_personSomUninit");

 /* clean up storage to avoid a memory leak */
 if(somThis->name)
 SOMFree(somThis->name);

 if(somThis->address)
 SOMFree(somThis->address);

 if(somThis->home_phone)
 SOMFree(somThis->home_phone);

 payroll_person_parent_SOMObject_somUninit(
 somSelf);
}

SOM_Scope void SOMLINK \
base_calc_pay(
 payroll_employee *somSelf, Environment *ev)
{
 payroll_employeeData *somThis =
 payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug(
 "payroll_employee",
 "base_calc_pay");

 printf("Your pay is %8.2f.\n",
 (__get_salary(somSelf,ev)/52));
}

```

Figure 5-2 (Part 1 of 2). “Payroll” implementation code.

```

SOM_Scope void SOMLINK \
base_hire(
 payroll_employee *somSelf, Environment *ev)
{
 char buf[100];
 payroll_employeeData *somThis =
 payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug(
 "payroll_employee",
 "base_hire");

 printf("Enter name: \n");
 __set_name(somSelf, ev,
 strcpy(SOMMalloc(strlen(gets(buf))+1), buf));

 printf("Enter Annual Wage: \n");
 __set_salary(somSelf, ev, atof(gets(buf)));
 __set_health_benefits(somSelf, ev, 1);
}

SOM_Scope void SOMLINK \
base_fire(
 payroll_employee *somSelf, Environment *ev)
{
 payroll_employeeData *somThis =
 payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug(
 "payroll_employee",
 "base_fire");

 /* This method is intentionally left blank. It is reserved for
 * future expansion of the employee class.
 */
}

SOM_Scope void SOMLINK \
base_promote(
 payroll_employee *somSelf, Environment *ev)
{
 payroll_employeeData *somThis =
 payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug(
 "payroll_employee",
 "base_promote");

 /* This method is intentionally left blank. It is reserved for
 * future expansion of the employee class.
 */
}

```

Figure 5-2 (Part 2 of 2). “Payroll” implementation code.

## **Steps 5 and 6: Compile the Implementation Code with the DLL Option, Prelink and Link**

**Compiling the Object Deck:** You would now run this code (dataset named *sommvs.SGOSSMPC.C(PAYROLL)*) through the C compiler using the DLL option on the compiler command invocation. (Use the EXPORTALL option to specify all the functions and variables in the DLL.)

**Note:** For C++, the DLL option is not required. Because C++ is an object-oriented language, it automatically creates the stubs required to create a DLL.

Alternatively, you can identify specific functions and variables to be exported by using the #pragma export directive in the DLL source file. The syntax of this directive is:

```
#pragma export (name)
```

where *name* is the name of the function or variable to be exported.

After creating the C compile step to create your object deck, your next steps are to prelink and link this object deck to create a DLL load module.

**Prelinking the object deck:** The prelink JCL step must identify the name of a dataset to receive the definition side deck (SYSDEFSD) generated by the Language Environment for MVS & VM prelinker. The definition side deck is identified in JCL with the label SYSDEFSD as shown in the following example:

```
//SYSDEFSD DD DSNAME=hlq(member),DISP=SHR
```

where *hlq* is the user-definable high level qualifier.

**Linking the object deck:** Link-edit the prelinked object deck. The output of the prelinker is an updated object deck that contains additional CSECTs to resolve the names and offsets of the DLL's functions and variables. This object deck is used as input to the linkage editor which in turn creates the DLL load module.

**Note:** This load module is not independently executable, and it must be reentrant (for C and C++ it does not contain a MA/N function). However, functions and variables exported by this load module are callable and accessible by application programs.

Figure 5-3 on page 5-9 shows the C compiler JCL with the EXPORTALL option specified on the C compiler command invocation and the prelink and link JCL to create the DLL load module:

```

//PAYDLL JOB <JOB CARD PARAMETERS>
//*
// SET SOM=SOMMVS
// SET IDL=SOMMVS.SGOSSMPI
// SET HDSN=SOMMVS.SGOSSMPH.H
// SET CDSN=SOMMVS.SGOSSMPC.C
//ORDER JCLLIB ORDER=(CBC.SCBCPRC,CEE.SCEEPROC)
//-----*
//** Description: This JCL is used to build the payroll class
//** library and a client application. The steps are
//** as follows:
//**
//** Before submitting this job, the JCL must be customized
//** for your installation. The following changes need to be
//** made:
//**
//** 1. Update the JOB card with your installation specific
//** parameters.
//** 2. Change the IDL variable above so it is set to the high
//** level qualifiers for the location of the example
//** IDL source code on your system.
//** 3. If necessary, change the SOM variable
//** to the high level qualifiers under which the SOM/MVS
//** code has been installed.
//** 4. This batch job uses the C/C++ JCL procedures
//** EDCC to compile and the Language Environment for MVS & VM
//** JCL procedure EDCPL to prelink.
//-----*
//**
//** 1) SOM Compile the IDL to get the private headers (h & ih).
//** 2) Compile the employee class library, using the private header
//** files.
//** 3) Prelink and linkedit the PAYROLL class library to create a DLL.
//**
//-----*
//** SOM JCL Procedure
//-----*
//SC PROC INDSN=,
// SCPARMS='',
// SOMPRFX=&SOM.,
// LEPRFX=&LE.
//SOMC EXEC PGM=SC,REGION=40M,
// PARM='&SCPARMS.' '&INDSN.'')
//STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
// DD DSN=&SOMPRFX..SGOSLOAD,DISP=SHR
//SOMENV DD DSN=&SOMPRFX..SGOSPROF(GOENV),DISP=SHR
//SYSPRINT DD SYSOUT=*
// PEND
//-----*
//** SOM Compile - private
//-----*
//SCPRIV EXEC SC,
// SCPARMS='-V -v -p -sh:ih -maddstar',
// INDSN=&IDL..IDL(PAYROLL)
//-----*
//** Compile employee class library using private header file *
//-----*
//PAYCC EXEC EDCC,
// INFIL=&CDSN..C(PAYROLL),
// CPARM='RENT LO SO SHOW DLL'
//COMPILE.SYSLIB DD
// DD DSN=&HDSN..H,DISP=SHR
// DD DSN=&SOM..SGOSSH.STARS.H,DISP=SHR
// DD DSN=&SOM..SGOSH.H,DISP=SHR
//COMPILE.IH DD DSN=&IDL..IH,DISP=SHR

```

Figure 5-3 (Part 1 of 2). "Payroll" JCL to compile the implementation code, prelink and link into a DLL load module.

```

/*
-----*
/* Run the pre-link and link processing to create *
/* the PAYROLL DLL *
-----*
//PAYRPL EXEC EDCPL,
//INFILE=*.PAYROLL.COMPILE.SYSLIN
//PLKED.SYSDEFSD DD DSN=&&IMPORTS(PAYROLL),DISP=(NEW,PASS),
// UNIT=SYDA,SPACE=(TRK,(3,3,1)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//PLKED.IMPORT DD DSN=SOMMVS.SGOSIMP,DISP=SHR
//PLKED.SYSIN DD
// DD *
INCLUDE IMPORT(GOSSOMK)
NAME PAYROLL(R)
/*

```

Figure 5-3 (Part 2 of 2). "Payroll" JCL to compile the implementation code, prelink and link into a DLL load module.

The Language Environment for MVS & VM prelinker creates a definition side deck that contains control records identifying each exported function and variable defined in the DLL. This file will be used during the prelink of applications that use the functions and variables contained in the DLL. The following is the contents of the SYSDEFSD for "Payroll":

```

IMPORT CODE 'PAYROLL' base_get_health_benefits
IMPORT CODE 'PAYROLL' base_get_salary
IMPORT CODE 'PAYROLL' base_get_work_phone
IMPORT CODE 'PAYROLL' base_set_health_benefits
IMPORT CODE 'PAYROLL' base_set_salary
IMPORT CODE 'PAYROLL' base_set_work_phone
IMPORT CODE 'PAYROLL' base_calc_pay
IMPORT CODE 'PAYROLL' base_fire
IMPORT CODE 'PAYROLL' base_hire
IMPORT CODE 'PAYROLL' base_promote
IMPORT DATA 'PAYROLL' payroll_employeeClassData
IMPORT DATA 'PAYROLL' payroll_employeeCClassData
IMPORT CODE 'PAYROLL' payroll_employeeNewClass
IMPORT CODE 'PAYROLL' payroll_person_get_address
IMPORT CODE 'PAYROLL' payroll_person_get_home_phone
IMPORT CODE 'PAYROLL' payroll_person_get_name
IMPORT CODE 'PAYROLL' payroll_person_set_address
IMPORT CODE 'PAYROLL' payroll_person_set_home_phone
IMPORT CODE 'PAYROLL' payroll_person_set_name
IMPORT CODE 'PAYROLL' payroll_personsomUnit
IMPORT DATA 'PAYROLL' payroll_personClassData
IMPORT DATA 'PAYROLL' payroll_personCClassData
IMPORT CODE 'PAYROLL' payroll_personNewClass

```

Now that you've seen how a "Payroll" class library DLL is built, it's time to move to Chapter 6, "Developing Client Applications" on page 6-1 and see what a client programmer has to do to access that DLL.

---

## Chapter 6. Developing Client Applications

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

This chapter continues using the “Payroll” example to help you understand client programs and also demonstrates running a client program by accessing the “Payroll” DLL created in the previous chapter.

This chapter discusses the following topics:

- “Who Should Read This Chapter?” on page 6-2
- “An Example Client Program” on page 6-4
- “Using Classes” on page 6-5
- “Compiling, Prelinking, and Linking” on page 6-33
- “Running the Client Program” on page 6-34
- “C/C++ Methods and Functions” on page 11-25

Steps 2, 3, and 4 from the “Developing Client Applications” side of Figure 6-1 on page 6-2 shows the part of the SOMobjects “Big Picture” process that will be used in creating the application code, prelinking, linking and compiling the application code to use the DLL created by the class library builder for the simple “Payroll” example.

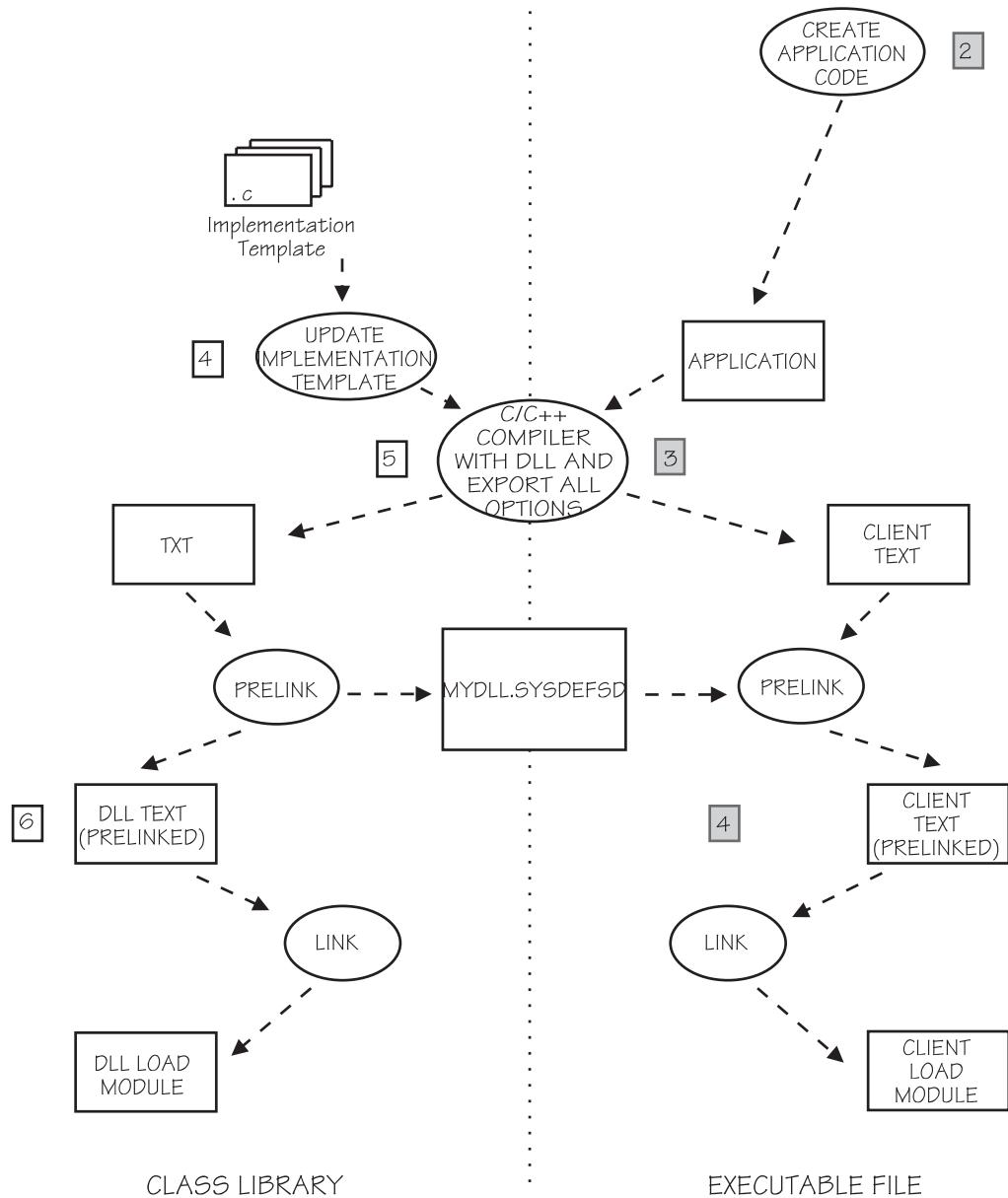


Figure 6-1. Steps for building and accessing DLLs.

## Who Should Read This Chapter?

Programmers who wish to use SOMobjects classes that were originally developed by someone else will need to know the information in this chapter. These programmers often may not need the information from any subsequent chapters. By contrast, class implementers who are creating their own SOMobjects classes should read Chapter 8, "Non-Distributed SOMobjects Examples" on page 8-1.

## Understanding Client Programs

Programs that use a class are referred to as *client programs*. A client program can be written in C, C++ or COBOL. As noted, this chapter describes how client programs can use SOMobjects classes. Using a SOMobjects class involves creating instances of a class, invoking methods on objects, and so forth. All of the methods, functions, and macros described here can also be used by class implementers within the implementation data set for a class.

**Note:** “Using a SOMobjects class,” as described in this chapter, does *not* include subclassing the class in a client program. In particular, the C++ compatible SOMobjects classes made available in the XH binding can *not* be subclassed in C++ to create new C++ or SOMobjects classes.

## SOMobjects Usage Bindings

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

### C/C++ Header Dataset Member Names

To use the C or C++ bindings for a class, a client program must include a header data set member for the class (using the `#include` preprocessor directive). For a C language client program, the header data set member `<dataset name.h>` must be included. For a C++ language client program, the header data set member `<dataset name.xh>` must be included. The SOMobjects Compiler generates these header data sets from an IDL interface definition. The header data sets contain definitions of the macros and functions that make up the C or C++ bindings for the class. Whether the header data sets include bindings for the class's private methods and attributes (in addition to the public methods and attributes) depends on the IDL interface definition available to the user, and on how the SOMobjects 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 SOMobjects object (for example, a client program that does not know at compile time what classes it will be using) should simply include the SOMobjects-supplied bindings for SOMObject. These bindings are provided in:

- The C header data set members `SOMMVS.SGOSH.H(SOMOBJ)` or `SOMMVS.SGOSSH.STARS.H(SOMOBJ)`  
`#include <somobj.h>`
- The C++ header data set members `SOMMVS.SGOSXH.XH(SOMOBJ)`  
`#include <somobj.xh>`

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

- A C programmer using the C bindings,
- A C++ programmer using the C++ bindings, or
- A programmer not using SOMobjects's C or C++ language bindings.

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

---

## An Example Client Program

Figure 6-2 is a C program that uses the class "Payroll" that was created in the previous chapters. The "Payroll" class provides four attributes (*name*, *address*, *salary*, and *health\_benefits*), and one method (*calc\_pay*). The *calc\_pay* method calculates the bi-monthly salary of the employee object on which the method is invoked.

```
#include "payroll.h" /* include the header data set for payroll */

int main(int argc, char *argv[]) {
 /* declare a variable 'emp_obj' that is a pointer
 * to an instance of the 'employee' class
 */
 payroll_employee *emp_obj;
 Environment *ev = somGetGlobalEnvironment();

 /* create an instance of the 'employee' class
 * and assign it to 'emp_obj'
 */
 emp_obj = payroll_employeeNew();

 /* set some attributes in the 'employee' class object
 */
 __set_name(emp_obj, ev, "John Baker");
 __set_address(emp_obj, ev, "123 South Rd., New York, NY 12345");
 __set_salary(emp_obj, ev, 32000.00);
 __set_health_benefits(emp_obj, ev, 0);

 /* Calculate the pay for the employee pointed to by 'emp_obj'
 */
 _calc_pay(emp_obj, ev);

 /* free memory associated with the employee object
 */
 _somFree(emp_obj);

 return(0);
}
```

Figure 6-2. A C client program that accesses the "Payroll" DLL.

The C++ version of the foregoing client program is shown in Figure 6-3 on page 6-5.

```

#include "payroll.xh" /* include the header data set for payroll */

int main(int argc, char *argv[]) {
 /* declare a variable 'emp_obj' that is a pointer
 * to an instance of the 'employee' class
 */
 payroll_employee *emp_obj;
 Environment *ev = somGetGlobalEnvironment();

 /* create an instance of the 'employee' class
 * and assign it to 'emp_obj'
 */
 emp_obj = new payroll_employee;

 /* set some attributes in the 'employee' class object
 */
 emp_obj->_set_name(ev, "John Baker");
 emp_obj->_set_address(ev, "123 South Rd., New York, NY 12345");
 emp_obj->_set_salary(ev, 32000.00);
 emp_obj->_set_health_benefits(ev, 0);

 /* Calculate the pay for the employee pointed to by 'emp_obj'
 */
 emp_obj->calc_pay(ev);

 /* free memory associated with the employee object
 */
 emp_obj->somFree();

 return(0);
}

```

Figure 6-3. A C++ client program that accesses the “Payroll” DLL.

These client programs both instantiate an employee object (*emp\_obj*) of class *payroll\_employee* and set the objects attributes. They then calculate the salary of the employee pointed to by *emp\_obj* and then the object is freed.

## Using Classes

The previous sections discussed the SOMobjects class structure and concepts and information class designers and builders need to know. This section describes the basics for using SOMobjects classes. For information about understanding SOM programming, refer to “Initializing the SOMobjects Programming Environment” on page 2-11 .

This section discusses the following:

- “Declaring Object Variables” on page 6-6
- “Creating Instances of a Class” on page 6-7
- “Invoking Methods on Objects” on page 6-12
- “Using Class Objects” on page 6-20
- “Using va\_list Methods” on page 6-24

The examples in this section will have the following naming conventions:

- “Payroll” is the module
- *employee* is the class
- *payroll\_employee* is the <*className*>

## Declaring Object Variables

When declaring an object variable, the class name defined in IDL is used as the type of the variable. The exact syntax is slightly different for C vs. C++ programmers. Specifically,

|                               |                                                                    |
|-------------------------------|--------------------------------------------------------------------|
| <code>className obj ;</code>  | in C programs or                                                   |
| <code>className *obj ;</code> | in C programs accessing classes created using the addstar modifier |
| <code>className *obj ;</code> | in C++ programs                                                    |

declares “obj” to be a pointer to an object that has type *className*. In SOMobjects, objects of this type are instances of the SOMobjects class named *className*, or of any SOMobjects class derived from this class. Thus, for example,

|                                     |                                             |
|-------------------------------------|---------------------------------------------|
| <code>payroll_employee obj;</code>  | in C programs or                            |
| <code>payroll_employee *obj;</code> | in C programs using the addstar modifier or |
| <code>payroll_employee *obj;</code> | in C++ programs                             |

declares “obj” as pointer to an object of type “payroll\_employee” that can be used to reference an instance of the SOMobjects class “payroll\_employee” or any SOMobjects class derived from “payroll\_employee”. Note that the type of an object need not be the same as its class; an object of type “payroll\_employee” might not be an instance of the “payroll\_employee” class (rather, it might be an instance of some subclass of “payroll\_employee” — the “special\_payroll\_salary” class, perhaps).

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

|                              |                                             |
|------------------------------|---------------------------------------------|
| <code>SOMObject obj;</code>  | in C programs or                            |
| <code>SOMObject *obj;</code> | in C programs using the addstar modifier or |
| <code>SOMObject *obj;</code> | in C++ programs.                            |

Because the sizes of SOMobjects objects are not known at compile time, instances of SOMobjects 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 “Understanding the Syntax of the SOMobjects Compiler” on page 4-11.

Users of other programming languages must also define object variables to be pointers to the data structure used to represent SOMobjects objects. The way this is done is programming-language dependent. The header data set “`somtype.h`” defines the structure of SOMobjects objects for the C and C++ languages, and is included by the C and C++ usage bindings.

## Creating Instances of a Class

### For C programmers with Usage Bindings

SOMobjects provides the `classNameNew` and the `classNameRenew` macros for creating instances of a class.

These macros are illustrated with the following two examples, each of which creates a single instance of class “employee”:

```
obj = payroll_employeeNew();
obj = payroll_employeeRenew(buffer);
```

**Using <className> New:** After verifying that the *className* class object exists, the `classNameNew` macro invokes the `somNew` method on the class object. This allocates enough space for a new instance of **className**, creates a new instance of the class, initializes this new object by invoking `somDefaultInit` method on it, and then returns a pointer to it. The `classNameNew` 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 `classNameNew` macro, the object should be freed by invoking the method `somFree` on it:

```
_somFree(obj);
```

After uninitialized the object by invoking **somDestruct Method** on it, **somFree** calls the class object for storage deallocation. Storage for an object created using the *classNameNew* macro is allocated by the class of the object. Thus, only the class of the object can know how to reclaim the object's storage.

**Using <className> Renew:** After verifying that the *className* class object exists, the *classNameRenew* macro invokes the **somRenew** method on the class object. *classNameRenew* 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 *classNameRenew* 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 *classNameNew*, the *classNameRenew* macro automatically creates any required class objects that have not already been created.

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

In addition, use of the*classNameRenew* macro requires that the class object has already been created (otherwise an error will result).

**Using <className> NewClass:** For example, the C code in Figure 6-4 on page 6-9 uses the function **payroll\_employeeNewClass** to create the “employee” 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. Once the class object has been created, the example invokes the method **somGetInstanceSize** on this class to determine the size of an “employee” object, uses SOMMalloc to allocate storage, and then uses the **payroll\_employeeRenew** macro to create ten instances of the “employee” class:

```

#include "payroll.h" /* include the header data set for payroll */

int main(int argc, char *argv[]) {
 SOMClass *employeeCls; /* A pointer for the payroll_employee object */
 payroll_employee *emp_objs[10]; /* array of payroll_employee instances */
 Environment *ev = somGetGlobalEnvironment();
 unsigned char *buffer;
 int size, x;

 /* create the payroll_employee object */
 employeeCls = payroll_employeeNewClass
 (payroll_employee_MajorVersion,
 payroll_employee_MinorVersion);

 /* determine the space needed for a payroll_employee instance */
 size = _somGetInstanceSize(employeeCls);
 size = ((size+3)/4)*4;

 /* allocate the total space needed for ten instances */
 buffer = SOMMalloc(10*size);

 /* convert the space into ten separate payroll_employee instances */
 for(x=0; x<10; x++)
 emp_objs[x] = payroll_employeeRenew(buffer+x*size);

 /* hire ten employees */
 for(x=0; x<10; x++)
 _hire(emp_objs[x], ev);

 /* Calculate the pay for the employees */
 for(x=0; x<10; x++)
 _calc_pay(emp_objs[x], ev);

 /* free memory associated with the employee objects */
 for(x=0; x<10; x++)
 _somDestruct(emp_objs [x],0,0);
 SOMFree(buffer);

 return(0);
}

```

Figure 6-4. Creating ten instances of the “employee” class in C.

When an object created with the **classNameRenew** 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 **classNameNew** 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 uninitialized. See “Initializing and Uninitializing Objects” on page 11-8.

## 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 11-8 . 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 payroll_employee;
obj1 = new payroll_employee();
```

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.

```
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 SOMobjects methods **somNew** and **somRenew**, invoked on the class object. As discussed and illustrated above for the C bindings, the class object must first be created using the **classNameNewClass** procedure (or, perhaps, using the **somFindClass** method). The **somNew** method invoked on the class object creates a new instance of the class and returns a pointer to it. For instance, the following C example creates a new object of the “employee” class.

```

#include <payroll.h>
main()
{
 SOMClass employeeCls; /* a pointer to the employee class */
 payroll_employee obj; /* a pointer to a payroll_employee instance*/
 /* create the employee class */
 employeeCls = payroll_employeeNewClass(payroll_employee_MajorVersion,
 payroll_employee_MinorVersion);
 obj = _somNew(employeeCls); /* create the employee instance */
}

```

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

The somRenew method invoked on the class object creates a new instance of a class using the given space, rather than allocating new space for the object. The method converts the given space into an instance of the class and returns a pointer to it. The argument to somRenew must point to a block of storage large enough to hold the new instance. The method somGetInstanceSize can be used to determine the amount of memory required. For example, the following C++ code in Figure 6-5 creates ten instances of the “employee” class:

```

#include <payroll.xh>
#include <somcls.xh>
main()
{
 SOMClass *employeeCls; // a pointer to the employee class
 payroll_employee *objA[10] //an array of payroll_employee
 instance pointers
 unsigned char *buffer;
 int i;
 int size;

 // create the payroll_employee class object
 employeeCls = payroll_employeeNewClass
 (payroll_employee_MajorVersion,
 payroll_employee_MinorVersion);

 // get the amount of space needed for a payroll_employee instance:
 size = employeeCls-> 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 payroll_employee objects
 for (i=0; i<10; i++)
 emp_objA[i] = employeeCls-> somRenew(buffer+i*size);

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

```

Figure 6-5. Creating ten instances of the “employee” class in C++.

The somNew and somRenew methods are useful for creating instances of a class when the header data set for the class is not included in the client program at compile time. (The name of the class might be specified by user input, for example.) However, the *classNameNew* macro (for C) and the **new** operator (for

C++) can *only* be used for classes whose header data set is included in the client program at compile time.

Objects created using the somRenew method should be freed by the client program that allocated it, using the dual to whatever allocation approach was initially used. If the method somFree is not appropriate (because the method somNew was not initially used), then, before memory is freed, the object should be explicitly deinitialized by invoking the somDestruct method on it. 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 (\_). An underscore is recommended for method invocations because it is a macro interface to the method provided by the SOM usage bindings. 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 the section on "Handling Exceptions" in *OS/390 SOMobjects Messages, Codes, and Diagnosis*.

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 an 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, somGetGlobalEnvironment() provides the pointer to the Environment structure, 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 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 SOMobjects, including **SOMObject**, **SOMClass** and **SOMClassMgr**, are defined in this way to ensure compatibility with previous releases of SOM. The *OS/390 SOMobjects Programmer's Reference, Volume 1* specifies when to use these arguments for each method.

**Note:** 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 *classNameNew* 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 3-21.

**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 is not valid 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) /* C invocation */
```

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 6-24), 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,** use the standard C++ form shown below to invoke a method:

```
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); /* C++ invocation */
```

#### *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 *OS/390 SOMobjects Messages, Codes, and Diagnosis* for more information on exceptions.

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**) the classes supplied in SOMobjects (including **SOMObject**, **SOMClass** and **SOMClassMgr**) are defined in this way, to ensure compatibility with the previous release of SOM. The *OS/390 SOMobjects Programmer's Reference, Volume 1* 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 3-21.

#### **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** function. The **somResolve** function 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:

```
<className>ClassData.<methodName>
```

For example, the method token for method *calc\_pay* introduced by class *payroll\_employee* is stored at location *payroll\_employeeClassData.calc\_pay*, for C and C++ programs. The way method tokens are accessed in other languages is language-specific.

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 the *OS/390 SOMobjects Programmer's Reference, Volume 1* 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 6-19.

## 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 *payroll\_employee* declares an attribute called *name*, then object variables of type *payroll\_employee* will support the methods *\_get\_name* and *\_set\_name* to access or set the value of the *name* 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 "payroll.h"

int main(int argc, char *argv[]) {
 payroll_employee *emp_obj;
 Environment *ev = somGetGlobalEnvironment();
 emp_obj = payroll_employeeNew();

 /* set the name attribute in the 'employee' class object
 and print it */
 __set_name(emp_obj, ev, "John Baker");

 printf("%s\n", __get_name(emp_obj, ev));

```

For C++:

```

#include "payroll.xh"
int main(int argc, char *argv[]) {
 payroll_employee *emp_obj;
 Environment *ev = somGetGlobalEnvironment();
 emp_obj = new payroll_employee;

 /* set the name attribute in the 'employee' class object
 and print it */
 emp_obj->_set_name(ev, "John Baker");

 printf("%s\n", emp_obj->_get_name(ev));

```

Attributes available with each class, are described in the documentation of each class in *OS/390 SOMobjects Programmer's Reference, Volume 1*.

## 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 2-9 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. 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 the appropriate linkage and 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 *calc\_pay*, introduced by class *payroll\_employee*, and uses it to invoke the method on *emp\_obj*. The only argument required by the *calc\_pay* method is the **Environment** pointer.

```

somMethodProc *p;
SOMObject emp_obj = payroll_employeeNew();
Environment *ev = somGetGlobalEnvironment();
p = SOM_Resolve(emp_obj, payroll_employee, calc_pay);
((somTD_payroll_employee_calc_pay)p) (emp_obj, ev);

```

**SOM\_Resolve** and **SOM\_ResolveNoCheck** can only be used to obtain method procedures for static methods (methods that have been declared in an IDL specification for a class) and not methods that are added to a class at run time. See the *OS/390 SOMobjects Programmer's Reference, Volume 1* 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 **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 *OS/390 SOMobjects Programmer's Reference, Volume 1*.

### 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 *OS/390 SOMobjects Programmer's Reference, Volume 1*.

## 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, SOMobjects provides a method called **somGetClass**. The **somGetClass** method takes an object as its only argument and returns a pointer to the class object of which it is an instance. For example, the following statements store in "myClass" the class object of which "obj" is an instance.

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

Getting the class of an object is useful for obtaining information about the object; in some cases, such information cannot be obtained directly from the object, but only from its class. "Getting Information about a Class" on page 11-25 describes the methods that can be invoked on a class object after it is obtained using **somGetClass**.

The **somGetClass** method can be overridden by a class to provide enhanced or alternative semantics for its objects. Because it is usually important to respect the intended semantics of a class of objects, the **somGetClass** method should normally be used to access the class of an object.

In a few special cases, it is not possible to make a method call on an object in order to determine its class. For such situations, SOMobjects 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 <className>Renew or somRenew:** It is sometimes necessary to create a class object before creating any instances of the class. For example, creating instances using the **<className>Renew** macro or the **somRenew** method requires knowing how large the created instance will be, so that memory can be allocated for it. Getting this information requires creating the class object. A class object must be explicitly created when a program does not use the SOMobjects bindings for a class. Without SOMobjects bindings for a class, its instances must be created

using `somNew` or `somRenew`, and these methods require that the class object be created in advance.

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

- When using the C/C++ language bindings for the class, and
  - When the name of the class is known at compile time.

**Using <className>NewClass:** The <className>NewClass procedure initializes the SOMobjects runtime 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

<sup>1</sup> See, e.g., *U.S. v. Sandoval*, 100 F.3d 1250, 1256 (10th Cir. 1996) (“[T]he term ‘substantial’ is not defined in the statute.”); *U.S. v. Gandy*, 100 F.3d 1250, 1256 (10th Cir. 1996) (“[T]he term ‘substantial’ is not defined in the statute.”).

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

of the expression

(for C and C++ programs)

or the expression

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

The **<className>NewClass** procedure takes two arguments, the major version number and minor version number of the class. These numbers are checked against the version numbers built into the class library to determine if the class is compatible with the client's expectations. The class is compatible if it has the same major version number and the same or a higher minor version number. If the class is not compatible, an error is raised. Major version numbers usually only change when a significant enhancement or incompatible change is made to a class. Minor version numbers change when minor enhancements or fixes are made. Downward compatibility is usually maintained across changes in the minor version number. Zero can be used in place of version numbers to bypass version number checking.

When using SOMobjects 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:

creates the class object for class “employee”. Thereafter, `_payroll_employee` can be used to reference the “employee” 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, runtime 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 `somFindClzInFile`:** 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 SOMobjects runtime environment by calling the `somEnvironmentNew` function (unless it is known that the SOMobjects runtime environment has already been initialized).
- Then, use the `somFindClass` or `somFindClIsInFile` method to create the class object. (The class must already be defined in a dynamically linked library, or DLL.)

The `somEnvironmentNew` function initializes the SOMobjects runtime environment. That is, it creates the four primitive SOMobjects objects (`SOMClass`, `SOMObject`, `SOMClassMgr`, and the `SOMClassMgrObject`), and it initializes SOMobjects global variables. The function takes no arguments and returns a pointer to the `SOMClassMgrObject`.

**Note:** Although `somEnvironmentNew` must be called before using other SOMobjects 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 SOMobjects runtime environment has been initialized, the methods `somFindClass` and `somFindClIsInFile` 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:

|                             |                                                                                                                                                          |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>classId</i>              | A somId identifying the name of the class to be created. The <code>somIdFromString</code> function returns a <i>classId</i> given the name of the class. |
| <i>major version number</i> | The expected major version number of the class.                                                                                                          |
| <i>minor version number</i> | 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 "employee" and stores a pointer to it in "myClass":

```

SOMClassMgr cm = somEnvironmentNew();
somId classId = somIdFromString("employee");
SOMClass myClass = _somFindClass(SOMClassMgrObject, classId,
 payroll_employee_MajorVersion,
 payroll_employee_MinorVersion);
...
SOMFree(classId);

```

The **somFindClass** method uses **somLocateClassFile** to get the name of the library data set containing the class. If the class was defined with a “**dllname**” class modifier, then **somLocateClassFile** returns the DLL load module name that was specified by **dllname**; otherwise, it returns the class name as the name of the DLL module to load. The **somFindClsInFile** method is similar to **somFindClass**, except that it takes an additional (final) argument—the name of the library data set 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.

**Attention:** The **somFindClass** and **somFindClsInFile** methods should *not* be used to create a class whose implementation is statically linked with the client program. Instead, the class object should be created using the **<className>NewClass** procedure provided by the class's H/XH header data set. Static linkage is not created by including usage bindings in a program, but is implied by use of the offset-resolution invocation macros.

**Attention:** **somFindClass** is not allowed in the CICS environment.

**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 provide 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>
#include "payroll.xh"
int main()
{
 SOMClass *classobj;
 somId tempId;
 somId methId;
 SOMObject *s2;
 Environment * main_ev = somGetGlobalEnvironment();
 tempId = SOM_IdFromString("employee");
 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("calc_pay");
 if (s2->somDispatch((somToken *) 0, methId, s2, ev))
 printf("Method successfully called.\n");
 }
 return 0;
}

```

**Getting a class 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 payroll\_employee (or NULL, if the class cannot be located):

```

SOMClassMgr cm = somEnvironmentNew();
somId classId = somIdFromString("payroll_employee");
SOMClass myClass = _somClassFromId(SOMClassMgrObject,
 classId,
 payroll_employee_MajorVersion,
 payroll_employee_MinorVersion)
SOMFree(classId);

```

## Using va\_list Methods

**Note:** As stated in “Migration Considerations” on page 1-11, in the previous releases, a **va\_list** was an array of two pointers. With this release, a **va\_list** is a pointer to string. This is done with:

```
#define _VARARG_EXT_
```

This is compatible with other platforms so porting applications to the OS/390 platform becomes easier.

If used, make sure #define is specified before includes of stdio.h and/or stdarg.h. If not used, va\_list will continue to be defined as an array of two pointers.

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 *OS/390 SOMobjects Programmer's Reference, Volume 1*.

## 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",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", 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 6-16) 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 6-13.

## 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 **lookupMethodName** 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 “Name-lookup Resolution” on page 11-30.

### 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)
Tookup_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 *OS/390 SOMobjects Programmer's Reference, Volume 1*.

**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).

For each method declared using OIDL or IDL, C and C++ usage bindings provide a typedef to name a type with correct linkage. 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 6-17.

### 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 and multiple- inheritance scenarios 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 6-6 on page 6-32 illustrates the class hierarchy for this example.

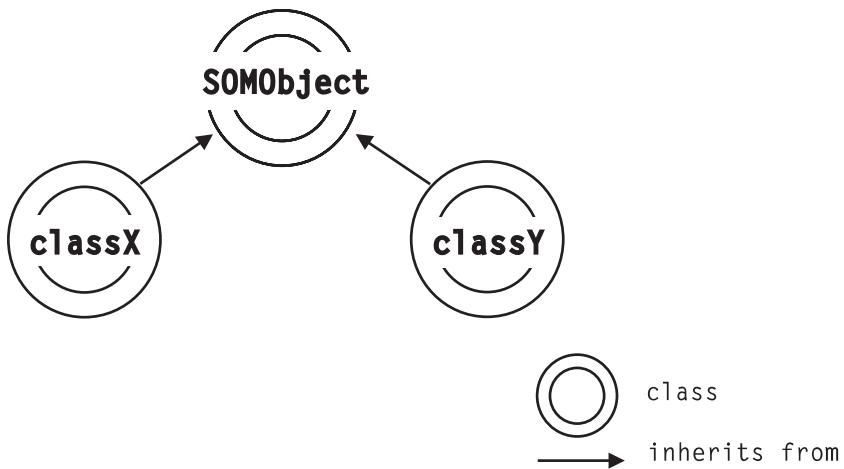


Figure 6-6. Name-Lookup Resolution

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);
}

```

## Compiling, Prelinking, and Linking

This section describes how to compile and link C and C++ client programs. Compiling, prelinking, and linking a client program with a SOMobjects class is done in one of two ways, depending on how the class is packaged.

*If the class is not packaged as a library* (that is, the client program is bound statically with the implementation source code for the class), then the client program can be compiled together with the class implementation data set 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 data set and the resulting object data sets linked together to form an executable program.)

The following are the recommended C compiler options to use:

|       |                                                                                |
|-------|--------------------------------------------------------------------------------|
| DLL   | - causes function descriptors to be used for function calls                    |
| EXPO  | - Export all no static functions and variables                                 |
| SO    | - Put the source code in the listing                                           |
| LO    | - Long names                                                                   |
| RENT  | - Generate reentrant code                                                      |
| SHOW  | - Put the includes in the listing                                              |
| EXP   | - Expand macros                                                                |
| NOMAR | - No margins. keeps headers from getting truncated at the default column of 72 |
| NOSEQ | - No sequence numbers in columns 73 through 80                                 |
| LIST  | - Generate pseudo assembler code.                                              |

Here is an example of using these options:

```
CPARM='DLL EXPO SO LO RENT SHOW EXP NOMAR NOSEQ LIST'
```

For more information on the C or C++ compiler and their options, see *OS/390 C/C++ Compiler and Run-Time Migration Guide*.

Refer to *sommvs.SGOSJCL(GOS1PAY)* for the JCL used to compile, prelink and link your application program to access the DLL load module.

If the class is packaged as a class library, then the client program, “main”, is compiled as in Step 6, except that the class implementation data set 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 “Payroll”, see Figure 6-8 on page 6-37.

Both of these examples assume that the header data sets and the import library for the "Payroll" class reside in the "binding data sets" and "import" data set provided with SOMobjects. If this is not the case, the SMINCLUDE path information should be supplied for these data sets.

---

## Running the Client Program

This section describes how to run your client program. It uses the "Payroll" application to illustrate how to run a client program.

This section involves the following three topics:

- Application access to DLLs
- Setting up your environment to run your client program
- Running your client program

## Application Access to DLLs

An application accesses a DLL function or variable by coding a reference to the target function or variable. The mechanisms for loading the particular DLL are transparent to the application program. In general, the application programmer codes only a function call or variable reference as if they were defined within the application program.

Load-time DLL access is handled by the trigger routines residing in the Language Environment for MVS. During initialization of an application load module, the MVS C/C++ Language Support feature on MVS/ESA initializes the application's writeable static area. This initialization includes setting up the variable and function descriptors that initially contain the addresses of the DLL runtime trigger routines. The names of the referenced DLLs, their functions, and variables are obtained by the runtime initialization routine from the information that the prelinker added to the application's object deck (see Figure 5-1 on page 5-3). This information maps back to the definition side deck ("MYDLL.SYSDEFSD") that was included during the prelink of the application.

When a DLL function or variable is referenced, the Language Environment for MVS & VM invokes the appropriate trigger routine to load the target DLL and to pass control to the target function or to enable application access to the target variable.

## Setting Up Your Environment to Run Your Client Program

For an example of setting up your environment, see "Step 5: Go (Run the Client Application Calling the "Payroll" Class)" on page 6-39.

## Running Your Client Application Program

This section discusses steps 2, 3, and 4 from the "Developing Client Applications" side of Figure 5-1 on page 5-3 and shows the part of the SOMobjects "Big Picture" process that will be used in creating an application program that will access the "Payroll" DLL.

- **Step 2:** Create the application code.
- **Step 3:** Compile the application with the DLL compiler option.
- **Step 4:** Prelink and link-edit the application.
- **Step 5:** Go (run the application program)

The following section walks you through these four steps:

## Step 2: Create the Application Code

Let's create the application code that will access the "Payroll" DLL load module so that we can run the methods named calc\_pay and hire. For C and C++ programs, be sure to include a function prototype for the functions that reside in DLLs. These can be explicitly coded or imbedded with a #include directive.

Figure 6-7 is the application code for the functions that reside in the DLL created in "Creating a DLL" on page 5-5 .

```
#include "payroll.h" /* include the header data set for payroll */

int main(int argc, char *argv[]) {
 /* declare a variable 'emp_obj' that is a pointer
 * to an instance of the 'employee' class
 */
 payroll_employee *emp_obj;
 Environment *ev = somGetGlobalEnvironment();

 /* create an instance of the 'employee' class
 * and assign it to 'emp_obj'
 */
 emp_obj = payroll_employeeNew();

 /* set some attributes in the 'employee' class object
 */
 __set_name(emp_obj, ev, "John Baker");
 __set_address(emp_obj, ev, "123 South Rd., New York, NY 12345");
 __set_salary(emp_obj, ev, 32000.00);
 __set_health_benefits(emp_obj, ev, 0);

 /* Calculate the pay for the employee pointed to by 'emp_obj'
 */
 _calc_pay(emp_obj, ev);

 /* free memory associated with the employee object
 */
 _somFree(emp_obj);

 return(0);
}
```

Figure 6-7. Client application code which calls the "Payroll" DLL load module.

## Steps 3 and 4: Compile the Application with the DLL Compiler Option, Prelink and Link

**Compiling the application with the DLL compiler option:** Take the application created in Step 2 and compile it with the DLL compiler option.

### Notes:

1. For C++, the DLL option is not required. Because C++ is an object-oriented language, it automatically creates the stubs required to create a DLL.
2. Do not specify the EXPORTALL option for the compile of DLL applications.

**Prelinking and Linking the Application:** In the prelink JCL procedure, be sure to specify an INCLUDE control card for the definition side decks of each DLL used by the application.

Link-edit the prelinked application program to create the application load module.

Figure 6-8 on page 6-37 is the JCL used to compile, prelink and link the application created in Step 2:

```

//GOS1PAY JOB <JOB CARD PARAMETERS >
//*
// SET SOM=SOMMVS
// SET LE=CEE
// SET CXX=CBC
// SET IDL=&SOM..SGOSSMPI
// SET HDSN=&SOM..SGOSSMPI
// SET CDSN=&SOM..SGOSSMPC
// SET LDSN=&SOM..SGOSLOAD
/*JOBPARM T=1,L=50
//ORDER JCLLIB ORDER=(&CXX..SCBCPRC,&LE..SCEEPROC)
//-----
//*
//* GOS1PAY --- Payroll Sample Application
//*
//** This JCL compiles, and links the PAY Sample application.
//*
//** Before submitting this job, the JCL must be customized
//** for your installation. The following changes need to be
//** made:
//*
//** 1. Update the JOB card with the installation specific
//** parameters.
//** 2. Change the value on the // SET SOM= statement to
//** the high level qualifiers used by SOMobjects for MVS on
//** your system if it is something other than SOMMVS.
//** 3. Change the value on the // SET LE= statement to
//** the high level qualifiers used by IBM LE/370 on
//** your system if it something other than CEE
//** 4. Change the value on the // SET CXX= statement to
//** the high level qualifiers used by IBM C/C++ Compiler on
//** your system if it something other than CBC
//** 5. Make a copy of the SGOSMPI.IDL dataset that was
//** shipped with SOMobjects for MVS. Change the value on
//** the // SET IDLPRFX= statement to the high level
//** qualifiers of the new data set, NOT including the final
//** .IDL qualifier.
//** 6. Make a copy of the SGOSMPI.H dataset that was
//** shipped with SOMobjects for MVS. Change the value on
//** the // SET HDSN= statement to the name of the new
//** data set.
//** 7. Make a copy of the SGOSMPC.C dataset that was
//** shipped with SOMobjects for MVS. Change the value on
//** the // SET CDSN= statement to the name of the new
//** data set.
//** 8. Make a copy of the SGOSLOAD dataset that was
//** shipped with SOMobjects for MVS. Change the value on
//** the // SET LDSN= statement to the name of the new
//** data set.
//** 9. This batch job uses the IBM C/C++ Compiler JCL procedures
//** EDCC and EDPL to compile and compile/prelink/link,
//** and the IBM LE/370 JCL procedure EDCPL to prelink.
//** Change these names if they have been changed at your
//** installation.
//-----*
//** This JCL does the following.
//-----*
//*
//** 1) SOM Compile the IDL to get the private headers (h & ih).
//**
//** 2) Compile the employee class library, using the private header
//** files.
//**
//** 3) SOM Compile the IDL to get the public headers (h only).
//**
//** 4) Compile the main application, using the public header file.

```

Figure 6-8 (Part 1 of 3). JCL to compile, prelink, and link the client application to create the executable file.

```

/*
/* 5) Prelink and linkedit the PAYROLL class library to create a DLL.
*/
/*
/* 6) Prelink and linkedit the PAYMAIN client application.
*/
/*
-----*
/* SOM JCL Procedure *
-----*
//SC PROC INDSN=,
// SCPARMS='',
// SOMPRFX=&SOM.,
// LEPRFX=&LE.
//SOMC EXEC PGM=SC,REGION=40M,
// PARM='&SCPARMS. ''&INDSN.''
//STEPLIB DD DSN=&SOMPRFX..SGOSLOAD,DISP=SHR
// DD DSN=&LEPRFX..SCEERUN,DISP=SHR
//SOMENV DD DSN=&SOMPRFX..SGOSPROF(GOSENV),DISP=SHR
//SYSPRINT DD SYSOUT=*
// PEND
/*
-----*
/* SOM Compile - private *
-----*
//SCPRIV EXEC SC,
// SCPARMS=' -V -v -p -sh:ih -maddstar',
// INDSN=&IDL..IDL(PAYROLL)
/*
-----*
/* Compile employee class library using private header file *
-----*
//PAYCC EXEC EDCC,
// INFILE=&CDSN..C(PAYROLL),
// CPARM='RENT LO SO SHOW DLL'
//COMPILE.SYSLIB DD
// DD DSN=&HDSN..H,DISP=SHR
// DD DSN=&SOM..SGOSSH.STARS.H,DISP=SHR
// DD DSN=&SOM..SGOSH.H,DISP=SHR
//COMPILE.IH DD DSN=&HDSN..IH,DISP=SHR
/*
-----*
/* SOM Compile - public *
-----*
//SCPUB EXEC SC,
// SCPARMS=' -V -v -sh -maddstar',
// INDSN=&IDL..IDL(PAYROLL)
/*
-----*
/* Compile main application using public header file *
-----*
//PAYMAIN EXEC EDCC,
// INFILE=&CDSN..C(PAYMAIN),
// CPARM='RENT LO SO SHOW DLL'
//COMPILE.SYSLIB DD
// DD DSN=&HDSN..H,DISP=SHR
// DD DSN=&SOM..SGOSSH.STARS.H,DISP=SHR
// DD DSN=&SOM..SGOSH.H,DISP=SHR
/*
-----*
/* Run the pre-link and link processing to create *
/* the PAYROLL DLL *
-----*
//PAYRPL EXEC EDCPL,
// INFILE=*.PAYCC.COMPILE.SYSLIN
//PLKED.SYSDEFSD DD DSN=&IMPORTS(PAYROLL),DISP=(NEW,PASS),
// UNIT=SYSDA,SPACE=(TRK,(3,3,1)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//PLKED.IMPORT DD DSN=&SOM..SGOSIMP,DISP=SHR
//PLKED.SYSIN DD
// DD *
INCLUDE IMPORT(GOSSOMK)
NAME PAYROLL(R)
/*

```

Figure 6-8 (Part 2 of 3). JCL to compile, prelink, and link the client application to create the executable file.

```

/*-----*
/* Run the pre-link and link processing to create *
/* the PAYMAIN client program *
/*-----*
//PAYLINK EXEC EDCPL,
// INFILE=*.PAYMAIN.COMPILE.SYSLIN
//PLKED.SYSDEFSD DD DUMMY
//PLKED.IMPORT DD DSN=&&IMPORTS,DISP=(SHR,DELETE)
// DD DSN=&SOM..SGOSIMP,DISP=SHR
//PLKED.SYSIN DD
// DD *
INCLUDE IMPORT(GOSSOMK)
INCLUDE IMPORT(PAYROLL)
NAME PAYMAIN(R)
//PLKED.SYSLIB DD DSN=&SOM..SGOSPLKD,DISP=SHR
//LKED.SYSLMOD DD DSN=&LDSN.(PAYMAIN),DISP=SHR
//

```

*Figure 6-8 (Part 3 of 3). JCL to compile, prelink, and link the client application to create the executable file.*

We have now:

- Designed a class library named “Payroll”
- Created the IDL for its classes
- SOM compiled the output and created the implementation templates
- Updated the implementation template with C method code
- Created a DLL load module by compiling, prelinking and linking the completed implementation template and
- Created a client application which accesses the “Payroll” DLL load module.

Let's now create the job to set up the environment to run the client application.

### **Step 5: Go (Run the Client Application Calling the “Payroll” Class)**

Before you can run the “Payroll” client application program which is named **PAYMAIN**, you have to set up your environment. Since the **PAYMAIN** program is an online interactive program, we will use a REXX exec to set up our environment. The REXX exec is in Figure 6-9 on page 6-40:

*Figure 6-9. "Payroll" REXX exec to set up your online interactive environment.*

After running the REXX exec, you can enter the following command on your TSO Ready command line and your "Payroll" interactive program will proceed:

PAYMAIN

---

## Chapter 7. Distributed SOMobjects

---

### Who Should Read This Chapter

This chapter is for the programmer who understands the SOMobjects concepts discussed in the previous chapters, and who wants to create or access reusable classes on other servers.

DSOM should be used for those applications that require *sharing* of objects among multiple programs. It can also be used to access remote objects which can access data on those remote machines. The object actually exists in only one process (this process is known as the object's *server*); the other processes (known as *clients*) access the object via remote method invocations, made transparently by DSOM.

DSOM should also be used for applications that require objects to be *isolated* from the main program. This is usually done in cases where reliability is a concern — either to protect the object from failures in other parts of the application, or (less often) to protect the application from an object.

Distributed SOMobjects (DSOM) provides security controls for client access to servers, classes and methods. Once you've decided to access your classes on a SOMobjects server, SOMobjects allows security control for accessing your class methods and data on that server.

Implementing classes in separate server processes gives you more flexibility with balancing storage consumption, in meeting performance requirements and implementing security controls.

The previous chapters discussed how to build and access reusable classes using SOM. This chapter will explain the differences between SOMobjects and Distributed SOMobjects.

This chapter addresses the following topics:

- “What is DSOM?” on page 7-2
- “DSOM Tutorial” on page 7-7
- “Setting Up the SOMobjects Environment” on page 7-15
- “Programming the Client Application” on page 7-16
- “Programming the Server” on page 7-32
- “Implementing SOM Classes in DSOM” on page 7-33
- “How DSOM complies with the Common Object Request Broker Architecture (CORBA)” on page 7-35

Additional information on advanced features of DSOM can be found in Chapter 12, “Distributed SOMobjects (DSOM) Advanced Topics” on page 12-1

---

## What is DSOM?

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 Distributed SOMobjects (DSOM) lies in the fact that DSOM insulates the client of an object from the object's location.

SOMobjects is a CORBA (Common Object Request Broker Architecture) compliant ORB (Object Request Broker). DSOM provides interobject client/server communications allowing application programs to access objects across address spaces, and across a network. This includes creating, destroying, identifying, locating, and invoking methods on remote objects.

DSOM is an extension to SOM that allows a program to invoke methods on SOM objects in other processes, even on different machines or platforms. Location and implementation of the object are transparent to the user, since the client accesses the object as if it was stored locally. DSOM also provides the services and underlying execution environment required by server applications which accept client requests for remote object activation and method execution. The server model is one in which clients are logically connected to a server address space through use of an authenticated binding. After the connection is established, the client may cause the activation and destruction of objects within the server. Objects created by a given client may persist beyond the life of the client connection and may be shared by other clients connected to the same server.

Distributed SOMobjects also exploits capabilities of OS/390, such as WLM server activation and scheduling functions, WLM performance management and RMF reporting.

## Other Uses, Benefits, and Features of DSOM

The following describes other uses, benefits and features of DSOM.

- SOMobjects for OS/390 is similar to *Workstation* DSOM on AIX and OS/2 in that it supports distribution among multiple processes on a single OS/390 system image. SOMobjects for OS/390 is also similar to *Workgroup* DSOM as well, since distribution across networks comprised of AIX, OS/2 and OS/390 systems is supported.
- SOMobjects uses the standard SOM compiler, Interface Repository (IR), language bindings, and class libraries; it provides a growth path for non-distributed SOMobjects applications.
- It 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.
- It 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.
- It provides a *default object server program* that you can use to create SOM objects and make them accessible to client programs. If the default server program is used, SOM class libraries are loaded upon demand, so no server programming or compiling is necessary.

DSOM object references contain:

- A reference to a server

- A reference to a specific object on that server.
- It complies with the CORBA 1.1 specification, which is important for portability of applications to other CORBA-compliant ORBs. In addition, SOMobjects compiles with the CORBA 2.0 IIOP 1.0 specification which addresses ORB interoperability.

## A Conceptual Overview of Distributed SOMobjects

The following distributed SOMobjects overview will give you a better understanding of what some of the components of SOMobjects are and where SOMobjects runs. This overview consists of the following topics:

- Platforms DSOM runs on and architecture it complies with
- SOM subsystem
- Naming Server
- Security Server
- OpenEdition Integrated Socket Support over TCP/IP and SNA.
- DSOM repository files (Interface Repository and Implementation Repository)
- Execution environment overview

Figure 7-1 on page 7-4 is an overview of DSOM. The text below the figure describes the overview topics and matches the topics pictured in the figure.

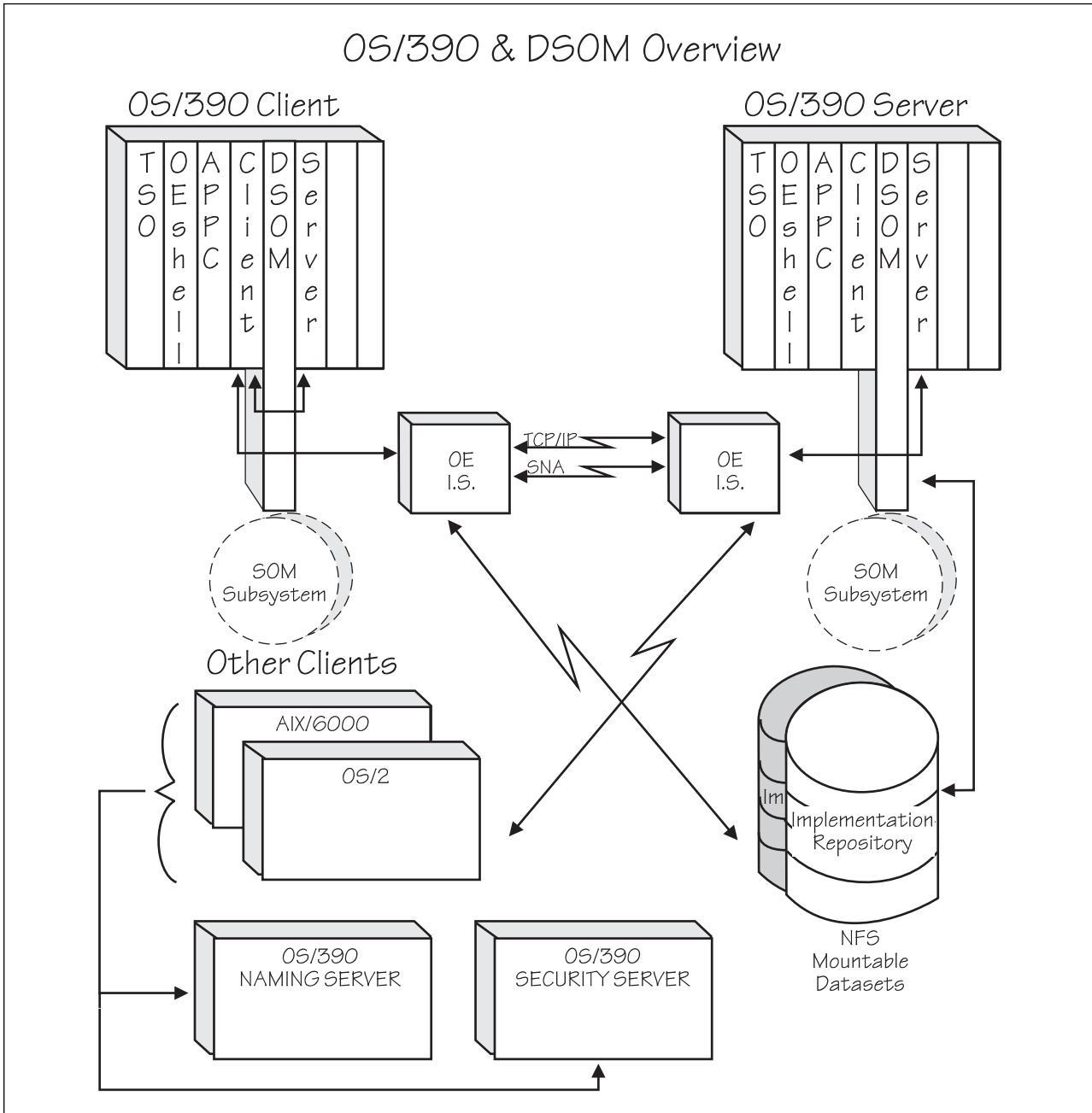


Figure 7-1. Overview of OS/390 Distributed SOMobjects

### Platforms SOMobjects Runs On and Architecture It Complies With

SOMobjects is an Object Request Broker (ORB), that is a standardized “transport” and set of runtime services for distributed object interaction among MVS, AIX/6000, and OS/2. SOMobjects complies with the CORBA 2.0, IIOP 1.0 specification published by the Object Management Group (OMG) and X/Open.

## **SOM Subsystem**

A SOM subsystem is required to enable receipt of inbound client connection requests and the activation of object server processes. The SOM subsystem is the initial point of contact for remote clients. In addition, the SOM subsystem provides services required by distributed object-oriented applications running on the OS/390 system. The SOM subsystem can be started under installation control. It is required that the SOM subsystem be started on every OS/390 system image in the sysplex if you want those systems to participate with DSOM.

## **Naming Server**

The Naming Server contains a service that enables you to name objects. You can assign a name to objects within a particular context, and then instances of naming contexts can be organized into a name hierarchy. During the configuration process, SOMobjects builds a default global name tree and binds certain distinguished objects within that name tree. The Naming Server is used by DSOM to store information about what servers support a particular class. It also stores location information for such servers.

## **Security Server**

The Security Server contains a service that allows you to implement secure object servers. It provides the means to authenticate clients which allows later verification of their authorization to requested servers, classes, and methods.

## **OpenEdition Integrated Socket Support (TCP/IP and SNA)**

SOMobjects leverages the integrated socket support provided by OS/390 OpenEdition, and supports a single communication interface, such as Berkeley socket interface, for multiple transport protocols such as TCP/IP and SNA

## **DSOM Repository Files (Interface Repository and Implementation Repository)**

SOMobjects requires two sets of repository files; the Interface Repository (IR) and the Implementation Repository to remotely access objects. The Interface Repository is used by SOMobjects to guide the construction and interpretation of request messages, and the Implementation Repository contains DSOM server information.

## **Execution Environment Overview (TSO, OpenEdition Shell, APPC, etc.)**

Many of the traditional OS/390 execution environments will support object-oriented applications and their use of DSOM. In particular, DSOM applications will be supported in the batch, TSO, OpenEdition Shell, APPC and started task execution environments.

## **DSOM Application 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 SOM subsystem location service running on the same machine as the servers. The SOM subsystem 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. All SOM processes running on the OS/390 system can make use of the global configuration file value. See the *OS/390 SOMobjects Configuration and Administration Guide* for more information on the configuration file.
- The Interface Repository files that are primarily used to load libraries dynamically in both client and server processes. These files are created and updated using the SOM Compiler. See "Register Your Classes in the Interface Repository" on page 7-15 for more information.
- Implementation Repository files contain information used by the SOM subsystem to start servers (in conjunction with the Workload Manager). These files also contain information that servers require for initialization. This repository is created and updated by registering servers using REGIMPL. See the sections on REGIMPL and the Workload Manager (WLM) in *OS/390 SOMobjects Configuration and Administration Guide* for additional information.
- Naming Service files that store information from the Naming Service and the DSOM Factory Service that store the files persistently on VSAM data sets. This includes information about which application classes are supported on each registered server, collected when the servers are registered.

The following configuration events are handled by the system programmer or administrator. These tasks need to be done in order for the application programmer to use DSOM. See *OS/390 SOMobjects Configuration and Administration Guide*.

The typical sequence of events that occurs when configuring and running a DSOM application is as follows:

1. Create a JCL procedure for the application server.
2. Define the security identity and authorizations for the application server.
3. Define the application server to WLM.
4. Register the application with the SOM Subsystem.
5. 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.

The following sections —

- “DSOM Tutorial”
- “Setting Up the SOMobjects Environment” on page 7-15
- “Programming the Client Application” on page 7-16
- “Programming the Server” on page 7-32
- “Implementing SOM Classes in DSOM” on page 7-33

provide more detail on how to use, manage, and implement remote objects, respectively.

---

## DSOM Tutorial

The DSOM tutorial presents a sample stack application as an introduction to DSOM. It will demonstrate that for simple examples, like stack, the class can be used to implement remotely accessed distributed objects. This tutorial discusses the various aspects of the code and various programming techniques required. For information on compiling, linking, and running distributed applications, see *OS/390 SOMobjects: Getting Started*.

## Application Components

The application components used to illustrate DSOM basics consist of the stack IDL interface provided with SOMobjects, client coding examples of SOM to DSOM stack changes, stack server implementation and application compiling.

### The Stack Interface

The following IDL example is included in the *sommvs.SGOSSMPI.IDL* data set in member STACK. This sample has already been emitted into the sample interface repository that is shipped with the product in data set *sommvs.SGOSIRSM*.

```
#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
};
```

Figure 7-2. Sample IDL from *sommvs.SGOSSMPI.IDL(STACK)*.

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 7-34 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, exID");
 somdExceptionFree(ev);
 return (FALSE) ;
 case USER_EXCEPTION:
 exID = somExceptionId(ev) ;
 somPrintf("User Exception: %s, 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 7-29 for more information on allocating and freeing memory. “Stack Example Run-Time Scenario” on page 7-13 describes the run time operations of the previous application.

## Code Differences and Similarities

- Every DSOM program must #include the file **somd.h** for C, **somd.xh** for C++ or **somd.hh** for DTS 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** function initializes the DSOM run-time environment, including allocation of global objects. **SOMD\_Init** function 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** 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** function. **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 “Finding Existing Objects” on page 12-21.

## 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.
- 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 **Stackserver** 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 SOM Classes in DSOM” on page 7-33 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 “Programming the Server” on page 7-32.

## 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 SOM subsystem 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 SOM subsystem, Workload Manager (WLM), 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 SOM subsystem for that server, and the communications protocols that can be used to contact it.

DSOM sends a message to that SOM subsystem requesting the location of the server.

The SOM subsystem 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 SOM subsystem of its port. The SOM subsystem 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 SOM subsystem.

- 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);
```

---

## Setting Up the SOMobjects Environment

Before you can build or use distributed SOMobjects classes, SOMobjects must be configured on your system. Complete information on configuring SOMobjects can be found in *OS/390 SOMobjects Configuration and Administration Guide*.

Along with the above tasks that are done by the system administrator, you must also register your classes in the Interface Repository.

## Register Your Classes in the Interface Repository

DSOM relies heavily on the Interface Repository for information on method signatures (that is, a description of the method's parameters and return value). It is important to compile the IDL for all application classes into the IR before running the application.

For each class in the DLL, compile the IDL description of the class into the Interface Repository. This is accomplished by invoking the following command syntax:

```
sc -usir stack.idl
```

If the default SOM IR (supplied with SOMobjects) is not used by the application, the user's IR must include the interface definitions for:

- the server class (derived from **SOMDServer**), and
- the definitions of the standard DSOM exceptions (found in file "stexcep.idl") that may be returned by a method call.

For more information on the Interface Repository, see Chapter 15, "The Interface Repository Framework" on page 15-1.

## DLL Considerations

The generic server uses SOM's run-time facilities to load class libraries dynamically. DLLs should be created for the classes. During the development of the DLLs, 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 *OS/390 SOMobjects Programmer's Reference, Volume 1*. 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.

---

## **Programming the Client Application**

Client programming in DSOM is generally the same as client programming in SOMobjects, 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 SOMobjects 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.

Programming the client application consists of the following topics:

- Process flow of a distributed SOMobjects application
- Initializing a client program
- Creating remote objects.
- Making remote method calls
- Destroying remote objects
- Exiting a client program
- Compiling and linking clients

## **Process Flow of a Distributed SOMobjects Application**

Figure 7-3 on page 7-17 shows the basic flow of what happens when a client application invokes methods on SOM objects in other processes. More details of the components within Figure 7-3 on page 7-17 will be covered later in this chapter.

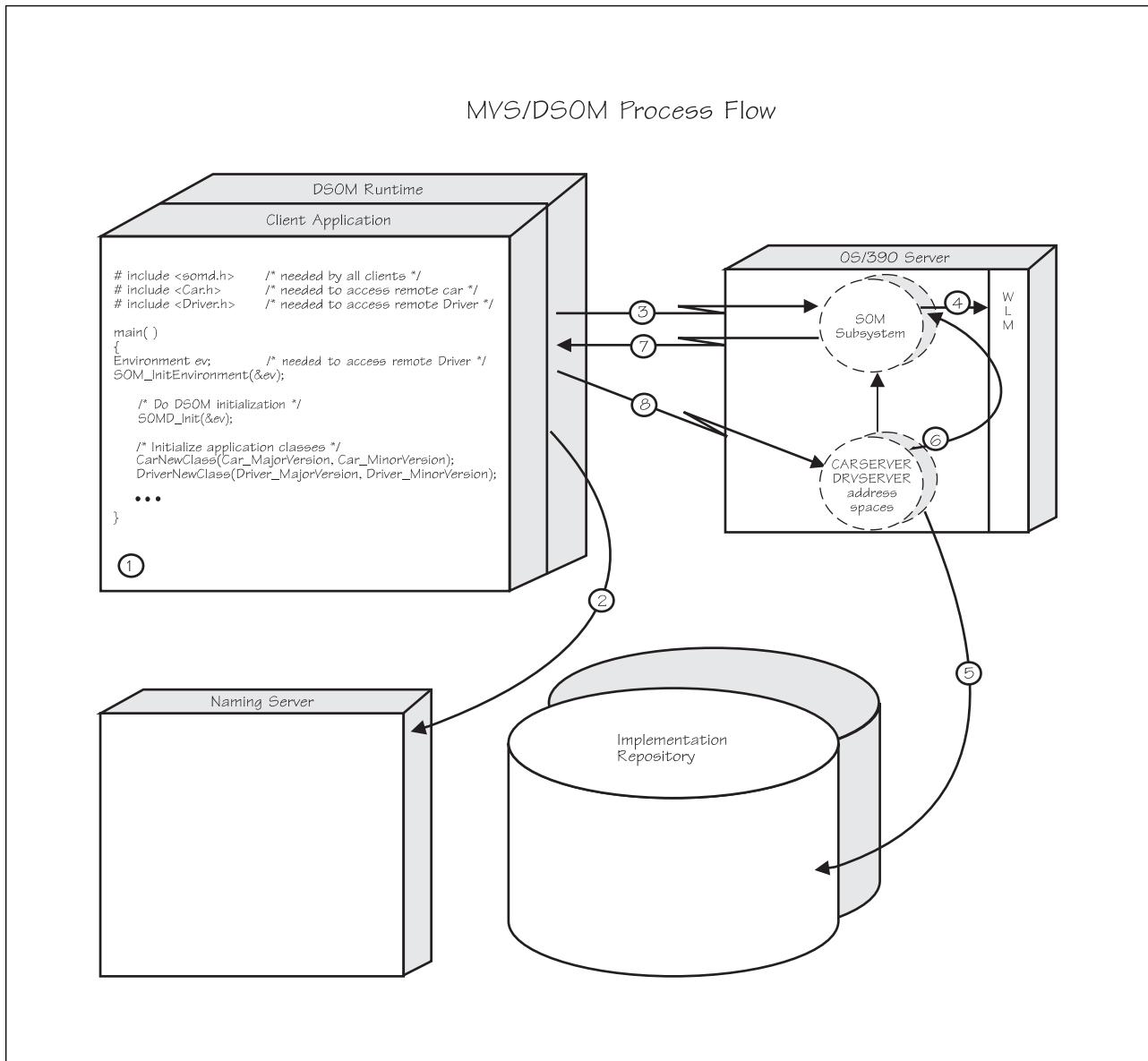


Figure 7-3. DSOM process flow of a client application invoking methods on SOM objects in other processes.

The following describes the process flow steps in Figure 7-3 of a client application invoking methods on SOM objects in other processes.

1. Add these statements in your client application to invoke methods on SOM objects ("Car" and "Driver") on another server (OS/390).
2. DSOM determines which servers support the "Car" and "Driver" classes by consulting the Naming Server.
3. DSOM talks to the SOM subsystem via TCP/IP which serves as a location service.
4. The SOM subsystem requests that WLM start the Car and Driver servers.
5. The Car and Driver server is started and uses values stored in the implementation repository during its initialization.
6. The Car and Driver server registers with the SOM subsystem via WLM providing its location information (hostname and port).

7. The SOM subsystem provides Car and Driver server location information to the client.
8. The client talks to server to create "Car" and "Driver" objects.

## Initializing a Client Program

This topic introduces the **ORB** object, describes how it is initialized by the **SOMD\_Init** function 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 (DII)" on page 7-40.

### 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++ or **somd.hh** for DTS 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

A factory is any object that can be used to create and return a reference to a new object.

Creating factory objects is done via **somdCreateFactory**. This method is called by the DSOM run time when a client requests a SOM object factory that must be created dynamically.

- 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. These two basic steps make use of the DSOM Factory Service to find object factories. The DSOM Factory Service provides a basic mechanism whereby applications can locate factories, implemented as a customization of the Naming Service.

The Life Cycle Service provides an alternative mechanism for creating remote objects and is layer of abstraction on top of the DSOM Factory Service. Applications may choose to find factories and create objects using the Life Cycle Service rather than interacting directly with the DSOM Factory Service.

There are advantages of the Life Cycle Service over the DSOM Factory Service which include:

- The Life Cycle Service offers a layer of abstraction that allows the system administrator to control the development environment. The Life Cycle Service interacts with the DSOM Factory Service on behalf of the user and offers additional policy control, filtering, creation and initialization capabilities above those offered by the DSOM Factory Service.
- The Life Cycle Service provides OMG-compliant interfaces.

For more information on the Life Cycle Service, see *OS/390 SOMobjects Object Services*.

The DSOM Factory Service, which will be discussed further in the following sections, has the following advantages:

- The DSOM Factory Service requires less configuration and administration. All configuration is done using the DSOM REGIMPL tool.
- 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.

Application developers should choose whether to use the DSOM Factory Service or the Life Cycle Service based on the degree to which the application requires policy administration and OMG compliance.

## 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:

- |                 |                                                                                                                     |
|-----------------|---------------------------------------------------------------------------------------------------------------------|
| <b>class</b>    | specifies the class name of the object to be created, as specified when the class/server association was registered |
| <b>alias</b>    | specifies the server alias where the factory is located                                                             |
| <b>serverId</b> | 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.

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 7-7 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 finds 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 stores 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 12-48 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 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 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);
 ...
}
```

## 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 “How DSOM complies with the Common Object Request Broker Architecture (CORBA)” on page 7-35.

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. 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 (DII)” on page 7-40.

**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 12-29.

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.

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,**  
**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** is forwarded to the target object and the proxy is destroyed.
- **release**: Uninitializes the proxy then calls **somDestruct** to destroy it.
- **somFree**: invokes **somFree** on the target object then calls **release** to uninitialized and destroy the proxy.

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 the client to server is for an object in that server, DSOM, as part of demarshalling the request in the server, gives the object references 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 provide they have executed their **impl\_is\_ready** initialization call.

In the server DSOM will not do anything to your object. DSOM creates a SOMDOObject to reference your object, marshals that SOMDOObject, 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 marshalling 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 implementers 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.

The referenced SOM IDL modifiers and other available memory-ownership policies are discussed in "Advanced Memory-Management Options" on page 12-24.

**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 case 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 7-29 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, **TypeCodes**, or the **\_value** field of **anys**, when used as **inout** or **out** arguments or return results
- **array** 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 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 memory not reused.
5. As part of returning the response to the client, the DSOM run time in the server process deallocates all parameter memory, including all introduced pointers for **inout** and **out** parameters.
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 memory, including the introduced pointers for **inout** and **out** parameters.

For details on inout memory, see "Reusing Memory for Inout Parameters" on page 7-29; for details on deallocating parameter memory, see "Memory-Management Functions" on page 7-29.

Although the client or DSOM run-time in the server process is responsible for allocating the various 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 ensuring that all **out** parameter and return-result memory contains valid values before returning.

**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. For additional information, see the description of the **suppress\_inout\_free** IDL modifier.

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 original, 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 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, since 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.

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** function 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.

**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 remote objects is discussed in “Creating Remote Objects” on page 7-19. Destroying remote objects is discussed in “Destroying Remote Objects.” 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**.

## 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.

If the proxy or remote object is to be 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.

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.

## 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 other system resources. **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.

## Compiling and Linking Clients

All client programs must include the header file "somd.h" (or for C++, "somd.xh") in addition to any "<className>.h" (or "<className>.xh") header files they require from application classes. All DSOM client programs must link to the SOMobjects library:

Once the source (in this case, C and header files) is in place, the OS/390 C/C++ compiler is used to compile all source files to produce object files, which also reside in PDSs. The **GOSSMPCC** sample JCL file shows how the compiler is invoked for an application program. Note that the DLL EXPORTALL compiler options are specified, since this SOMobjects code is packaged as DLLs. Optionally, EXPORT parameters would be specified as PRAGMA statements within the C code.

DSOM programs and class libraries are compiled and linked like any other SOM program or library. The header file "somd.h" (or for C++, "somd.xh") should be included in any source program that uses DSOM services. DSOM run-time calls can be resolved by linking with the SOMobjects library.

For more information, see the section on compiling, prelinking, and linking in "Compiling, Prelinking, and Linking" on page 6-33.

---

## Programming the Server

Server programs execute and manage object implementations. That is, they are responsible for:

- Notifying the SOM subsystem that they are ready to begin processing requests
- Accepting client requests
- Loading class library DLLs when required
- Creating/locating/destroying local objects
- Demarshalling client requests into method invocations on their local objects
- Marshalling method invocation results into responses to clients

- 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. A second server program, somossrv, is also provided for use with the SOMobjects object services. (For information about using somossrv, see the SOMOS chapter in *OS/390 SOMobjects Object Services*.) Each of these server programs can be found in data set sommvs.SGOSLOAD. The server program name is specified in the PROC for a given server. The Workload Manager will start that PROC when it receives a request to start a given server.

Some applications may require additional flexibility or functionality than what is provided by the generic server program. In that case, application-specific server programs can be developed. For more information on programming servers, see "Programming a Server" on page 12-33.

## Implementing SOM Classes in DSOM

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 multi-process DSOM-based applications as exemplified by the sample DSOM application "Stack" included in the SOMobjects product data sets. In fact, in many cases, existing SOM class libraries may be used in DSOM applications without requiring any special coding or recoding for distribution. This is possible through the use of DSOM's generic server program, which uses SOM and the SOM Object Adapter (**SOMOA**) to load SOM class libraries on demand, whenever an object of a particular class is created or activated.

The following topics will help you better understand how to use SOM class libraries:

- Role of DSOM generic server program
- Role of SOM Object Adapter
- Role of SOMDServer
- Implementation constraints

### Role of DSOM Generic Server Program (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. If it is not loaded, somdsvr finds and loads the DLL for the object's class.

## Role of SOM Object Adapter

The SOM Object Adapter is DSOM's standard object adapter. It provides basic support for receiving and dispatching requests on objects. As an added feature, the **SOMOA** and the server process's server object collaborate to automate the task of converting SOM object pointers into DSOM object references, and vice versa. That is, whenever an object pointer is passed as an argument to a method, the **SOMOA** and the server object convert the pointer to a DSOM object reference (since a pointer to an object is meaningless outside the object's address space).

For more information, refer to "Object Adapters" on page 7-41.

## 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.

For more information, refer to "Servers" on page 7-41.

## Implementation Constraints

The generic server program (**somdsvr**), the **SOMOA**, and the SOMDServer make it easy to use SOM classes with DSOM. However, if there are any parts of the class implementation that were written expecting a single-process environment, the class may have to be modified to behave properly in a client-server environment. Some common implementation practices to avoid are listed below:

- **Printing to standard output.** Any text printed by a method will appear at the server, 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. This is because these "methods"

are called directly, rather than via the normal method resolution mechanisms on which DSOM relies.

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 **structs**, including reordering of **struct** members, or **unions**. 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**.

---

## How DSOM complies with the Common Object Request Broker Architecture (CORBA)

The Object Management Group (OMG) consortium defines an *object request broker (ORB)* that provides access to remote objects in a distributed environment. DSOM is a type of ORB. SOM and DSOM together comply with the OMG specification of the Common Object Request Broker Architecture (CORBA).

The CORBA specification defines the components and interfaces that must be present in an ORB, including the:

- Interface Definition Language (IDL) for defining classes (discussed in Chapter 3, "SOMobjects Interface Definition Language (IDL)" on page 3-1)
- C usage bindings (procedure-call formats) for invoking methods on remote objects
- Dynamic Invocation Interface and an Interface Repository, which support the construction of requests (method calls) at run time (for example, for interactive desktop applications), and
- Object Request Broker run-time programming interfaces.

SOM and DSOM were developed to comply with these specifications (with only minor extensions to take advantage of SOM services). Although the capabilities of SOM are integral to the implementation of DSOM, the application programmer need not be aware of SOM as the implementation technology for the ORB.

This section assumes some familiarity with *The Common Object Request Broker: Architecture and Specification, Revision 1.1* (also referred to as "CORBA 1.1"). The specification is published jointly by the Object Management Group and x/Open.

The mapping of some CORBA 1.1 terms and concepts to DSOM terms and concepts is described in the next section.

## Mapping OMG CORBA Terminology onto DSOM

This section discusses how various CORBA concepts and terms are defined in terms of DSOM's implementation of the CORBA 1.1 standard. The topics that make up mapping OMG CORBA terminology onto DSOM are:

- Object Request Broker Run-Time Interfaces
- Object References and Proxy Objects
- Creation of Remote Objects
- Interface Definition Language
- C Language Mapping
- Dynamic Invocation Interface (DII)
- Implementations
- Servers
- Object Adapters
- DSOM Limitations
- DSOM Extensions

### Object Request Broker Run-Time Interfaces

In the previous sections, the SOMDOBJECTMGR and SOMDSERVER classes were introduced. These are classes defined by DSOM to provide basic support in managing objects in a distributed application. These classes are built upon Object Request Broker interfaces defined by CORBA for building and dispatching requests on objects. The ORB interfaces, SOMDOBJECTMGR and SOMDSERVER, together provide the support for implementing distributed applications in DSOM.

CORBA 1.1 defines the interfaces to the ORB components in IDL. In DSOM, the ORB components are implemented as SOM classes whose interfaces are expressed using the same CORBA 1.1 IDL. Thus, an application can make calls to the DSOM run time using the SOM language bindings of its choice.

Interfaces for the following ORB run-time components are defined in CORBA 1.1, and are implemented in DSOM. They are introduced briefly here, and discussed in more detail throughout this chapter. (See the *OS/390 SOMobjects Programmer's Reference, Volume 2* for the complete interface definitions.)

#### **BOA**

(Basic Object Adapter) An Object Adapter provides the primary interface between an implementation and the ORB "core". An ORB may have a number of Object Adapters, with interfaces that are appropriate for specific kinds of objects. The Basic Object Adapter is intended to be a general-purpose Object Adapter available on all CORBA-compliant Object Request Brokers. The **BOA** interface provides support for generation of object references, identification of the principal making a call, activation and deactivation of objects and implementations, and method invocation on objects.

In DSOM, **BOA** is defined as an abstract class. The **SOMOA** (SOM Object Adapter) class, derived from **BOA**, is DSOM's primary Object Adapter implementation. The **SOMOA** interface extends the **BOA** interface with several of its own methods that are not defined by CORBA 1.1.

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Context</b>           | A <b>Context</b> object contains a list of "properties" that represent information about an application process's environment. Each <b>Context</b> property consists of a <name,string_value> pair, and is used by application programs or methods much like the "environment variables" commonly found in operating systems like AIX 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.                                                 |
| <b>ImplementationDef</b> | An <b>ImplementationDef</b> object is used to describe an object's implementation. Typically, the <b>ImplementationDef</b> describes the program that implements an object's server, how the program is activated, and so on. (CORBA 1.1 introduces <b>ImplementationDef</b> as the name of the interface, but leaves the remainder of the IDL specification to the particular ORB. DSOM defines an interface for <b>ImplementationDef</b> .)<br><br>ImplementationDef objects are stored in the implementation repository (defined in DSOM by the <b>ImplRepository</b> class). |
| <b>InterfaceDef</b>      | An <b>InterfaceDef</b> 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, which is described in Chapter 15, "The Interface Repository Framework" on page 15-1.                                                                                                                                                                                                                                                |
| <b>NVList</b>            | An <b>NVList</b> is a list of NamedValue structures, used primarily in building <b>Request</b> objects. A NamedValue structure consists of a name, typed value, and some flags indicating how to interpret the value, how to allocate/free the value's memory, and so on.                                                                                                                                                                                                                                                                                                        |
| <b>Object</b>            | The Object interface defines operations on an "object reference", which is the information needed to specify an object within the ORB. In DSOM, the class SOMDObject implements the CORBA 1.1 Object interface. (The "SOMD" prefix was added to distinguish this class from <b>SOMObject</b> .) The subclass <b>SOMDClientProxy</b> extends SOMDObject with support for proxy objects.                                                                                                                                                                                           |
| <b>ORB</b>               | (Object Request Broker) The ORB interface defines utility routines for building requests and saving references to distributed objects. The global variable SOMD_ORBObject is initialized by <b>SOMD_Init</b> function and provides the reference to the ORB object.                                                                                                                                                                                                                                                                                                              |
| <b>Principal</b>         | A Principal object identifies the principal ("user") on whose behalf a request is being performed. (CORBA 1.1 introduces the name of the interface, Principal, but leaves the remainder of the IDL specification to the particular ORB. DSOM defines an interface for Principal.)                                                                                                                                                                                                                                                                                                |
| <b>Request</b>           | A <b>Request</b> object represents a specific request on an object, constructed at run time. The <b>Request</b> object contains the target object reference, operation (method) name, a list of input and output arguments. A Request can be invoked syn-                                                                                                                                                                                                                                                                                                                        |

chronously (wait for the response), asynchronously (initiate the call, and later, get the response), or as a “one way” call (no response expected).

## Object References and Proxy Objects

CORBA 1.1 defines the notion of an *object reference*, which is the information needed to specify an object in the ORB. An object is defined by its **ImplementationDef**, **InterfaceDef**, and application-specific “reference data” used to identify or describe the object. An object reference is used as a handle to a remote object in method calls. When a server wants to export a reference to an object it implements, it supplies the object’s **ImplementationDef**, **InterfaceDef**, and reference data to the Object Adapter, which returns the reference.

The structure of an object reference is opaque to the application, leaving its representation up to the ORB. In DSOM, an object reference is represented as an object that can simply be used to identify the object on that server. The DSOM class that implements simple object references is called **SOMDObject** (corresponding to *Object* in CORBA 1.1.) However, in a client’s address space, DSOM represents the remote object with a *proxy object* in order to allow the client to invoke methods on the target object as if it were local. When an object reference is passed from a server to a client, DSOM *dynamically* and *automatically* creates a proxy in the client for the remote object.

Proxies are specialized forms of **SOMDObject**; accordingly, the base proxy class in DSOM, **SOMDClientProxy**, is derived from **SOMDObject**. In order to create a proxy object, DSOM must first build a proxy class. It does so automatically using SOM facilities for building classes at run time. The proxy class is constructed using multiple inheritance: the proxy object functionality is inherited from **SOMDClientProxy**, while just the *interface* of the target class is inherited (see Figure 7-4).

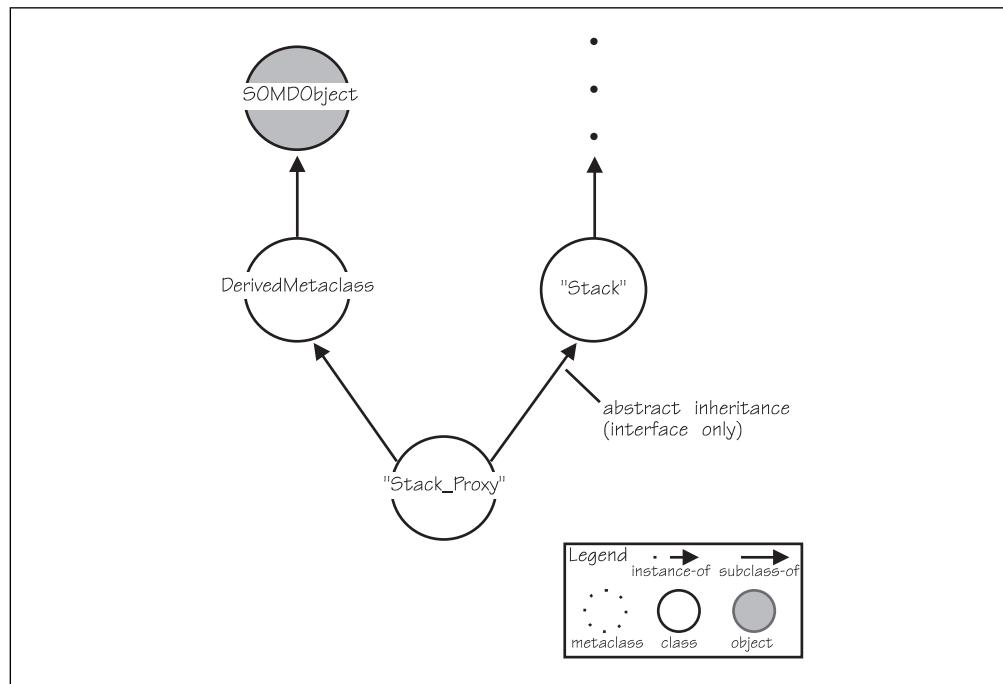


Figure 7-4. Construction of a Proxy Class in DSOM

In the newly derived proxy class, DSOM overrides each method inherited from the target class with a "remote dispatch" method that forwards an invocation request to the remote object. Consequently, the proxy object provides location transparency, and the client code invokes operations (methods) on the remote object using the same language bindings as if it were a local target object.

For example, the "Stack" class used in the "Stack" sample program demonstrates that when a server returns a reference to a remote "Stack" object to the client, DSOM builds a "Stack\_\_Proxy" class (note two underscores in the name), derived from **SOMDClientProxy** and "Stack", and creates a proxy object from that class. When the client invokes the "push" method on the proxy,

```
_push(stk, &ev, 100);
```

the method is redispached using the remote-dispatch method of the **SOMDClientProxy** class, and the method is forwarded to the target object.

CORBA defines several special operations on object references that operate on the local references (proxies) themselves, rather than on the remote objects. These operations are defined by the classes **SOMOA** (SOM Object Adapter), **SOMDObject** (which is DSOM's implementation of CORBA's **Object** "pseudo-class") and **ORB** (Object Request Broker class). Some of these operations are listed below, expressed in terms of their IDL definitions.

#### **SOMOA** methods (inherited from **BOA**):

```
sequence <octet,1024> ReferenceData;
SOMDObject create (in ReferenceData id, in InterfaceDef intf,
 in ImplementationDef impl);
```

Creates and returns an object reference.

#### **SOMDObject** methods:

```
SOMDObject duplicate ();
```

Creates and returns a duplicate object reference.

```
void
release ();
```

Destroys an object reference.

```
boolean is_nil ();:
```

Tests to see if the object reference is NULL.

**ORB** methods:

```
string object_to_string (SOMDOBJECT obj);
```

Converts an object reference to a (storable) string form.

```
SOMDOBJECT string_to_object (string str);
```

Converts a string form back to the original object reference.

## 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 implementer to include implementation information that will be used in the implementation bindings for a particular language.

## 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 *OS/390 SOMobjects Messages, Codes, and Diagnosis* for how to “get” and “set” error information in the Environment structure.

## Dynamic Invocation Interface (DII)

The CORBA specification defines a Dynamic Invocation Interface (DII) that can be used to dynamically build requests on remote objects. 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 12-59.

## 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 7-41 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 "Updating the Implementation Repository Dynamically Using the Programmatic Interface" on page 12-22.

## 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.

## DSOM Limitations

DSOM implementation has the following limitations in implementing CORBA specification:

- DSOM provides null implementations for the **obj\_is\_ready** or **deactivate\_impl** 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.
- The OUT\_LIST\_MEMORY, IN\_COPY\_VALUE and DEPENDENT\_LIST flags, used with the Dynamic Invocation Interface, are not yet supported. The **Context::get\_values** method currently does not support the CTX\_RESTRICT\_SCOPE flag.
- 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.



---

## Chapter 8. Non-Distributed SOMobjects Examples

Now that we've created a DLL as a class library builder and accessed that DLL as a client application, it's useful to explore an expanded version of the "Payroll" example, as well as other examples.

This chapter is organized into three major sections:

1. The expanded "Payroll" Example with IDL, SOM Compiled Output, Method Code, and Client Code" on page 8-2 includes:
  - An overview of the two major tasks that the "Payroll" example will cover
  - Basic steps for building and using SOMobjects classes (using JCL)
  - A DLL build and access of the "Payroll" module using a REXX example (found in "Building and Running an Application from the TSO Environment" on page 8-36).

2. "Setting Up and Running Non-distributed Vehicle in Different OS/390 Environments" on page 8-36

In this section, the following OS/390 environments will be used with the non-distributed "Vehicle" example:

- "Building and Running a Non-Distributed Application in the Batch Environment" on page 8-37
- "Building and Running a Non-Distributed Application from the OpenEdition Shell Environment" on page 8-37
- "Building and Running a Non-Distributed Application from the TSO Environment" on page 8-39

3. "Summarized Non-Distributed Examples" on page 8-40

These include:

- "Animals" on page 8-40
- "Text Processing" on page 8-42
- "Hello" on page 8-44

### Notes:

1. Before you can begin running any of these examples, the SOMobjects environment needs to be configured on your system. Consult with your system administrator to ensure that the SOMobjects environment has been configured. For more information on how to configure the SOMobjects environment, see the *OS/390 SOMobjects Configuration and Administration Guide*.
2. Some of these examples show that **somInit** and **somUninit** are overridden. These methods now execute under the overall control of the **somDefaultInit** method and the **somDestruct** method. When you, in your program, use the **somDefaultInit** method and the **somDestruct** method instead of the **somInit** and **somUninit** methods, then you must override them as well. See "Multiple Inheritance" on page 2-5 for more information.

---

## 'Payroll' Example with IDL, SOM Compiled Output, Method Code, and Client Code

See *OS/390 SOMobjects: Getting Started* for a high level overview of how to build and run a non-distributed example. This chapter covers the topic in more detail and in additional environments.

This section incorporates as many of the topics, tasks and options as possible from the previous chapters by elaborating on the "Payroll" example introduced in "Building a "Payroll" Class Library with Two Classes" on page 3-45.

This section consists of the following topics:

- Overview of the two major tasks that the "Payroll" example will cover
- Basic steps for building and using SOMobjects classes

**Note:** The "Payroll" example will be done in the C programming language and will be run both in the batch environment with JCL and in the TSO environment using REXX. If you would like to see other examples in the batch, OE, or TSO environments, see "Setting Up and Running Non-distributed Vehicle in Different OS/390 Environments" on page 8-36. If you would like to see examples using the C++ or COBOL languages, see Chapter 10, "Language Neutrality with SOMobjects: Examples" on page 10-1.

### Overview of the Two Major Tasks for the "Payroll" Example

The "Payroll" example will cover two major tasks in building the "Payroll" class library. Table 8-1 on page 8-3 defines the two major tasks and what each task demonstrates.

*Table 8-1. Two major tasks to be covered in the “Payroll” class library example.*

| <b>Task 1</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | <b>Task 2</b>                                                                                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Build a class</b> in “C” for a “C” client<br><br>1. Design OO methodology (flowchart)<br>Design class libraries for client application<br>2. Create IDL<br>Multi-language bindings with “ <b>one</b> ” compile<br>Demonstrates:<br><ul style="list-style-type: none"> <li>• The basic OO concepts, such as:           <ul style="list-style-type: none"> <li>– Encapsulation</li> <li>– Diverse use of methods</li> <li>– Messaging</li> <li>– Instances</li> </ul> </li> <li>• More advanced OO topics, such as:           <ul style="list-style-type: none"> <li>– Inheritance</li> <li>– Multiple Inheritance</li> <li>– Overriding</li> <li>– Polymorphism</li> </ul> </li> </ul> 3. Run Som Compiler<br>Generate bindings<br>Implementation skeleton<br>4. Update implementation template using “C”<br>5. Use “C” compile<br>6. Prelink & Link<br>Packaging <ul style="list-style-type: none"> <li>• DLL</li> </ul> | <b>“C” Client Program to use class built in Task 1</b><br>(This task can be seen in “Running Your Client Program” on page 8-32.)<br><br>1. Create application code<br>2. C Compile<br>3. Prelink and link<br>4. Go |

## Basic Steps for Building and Using SOMobjects Classes

Building and using SOMobjects classes are the two primary tasks in SOMobjects. First, we'll discuss building SOMobjects classes:

### Building SOMobjects Classes

Building SOMobjects classes involves the following steps, which are described in this section and can be seen in Figure 2-4 on page 2-13.

- “Step 1. Determine Classes/Objects Needed and SOM Runtime Option/Mode/Services to use” on page 8-4
- “Step 2. Create the Source IDL” on page 8-6
- “Step 3. Run the SOMobjects Compiler” on page 8-11
- “Step 4. Update the Implementation Template” on page 8-16
- “Steps 5 and 6: Compile the Implementation Code with the DLL Option, Prelink and Link” on page 8-31

We will now proceed to implement **Task 1** from Table 8-1 by completing the six steps. Each of the six steps will contain more detail and will build upon each other using “Payroll” as the example.

## **Step 1. Determine Classes/Objects Needed and SOM Runtime Option/Mode/Services to use**

Beyond understanding the general runtime concepts, you must plan for implementation of classes, including performing the following general tasks:

- Determine the class structure and attributes

Before building your class libraries, you need to research existing libraries to determine which provide the function your application programmers will need and then determine which classes you will need to provide using SOMobjects.

- Establish a naming convention for classes, objects, and methods

SOMobjects has specific naming conventions that you must follow to make sure that your source IDL data sets, the SOMobjects compiler, and your application programs work properly. The basic conventions that apply can be found in "Emitters the SOMobjects Compiler generates" on page 4-3.

**General Application Development Considerations:** Before beginning to develop your application class library and client program, contact your system programmer to determine what *hlq* (high level qualifier) has been assigned to the SOMobjects header data sets. The SOMobjects header data sets are required for SOMobjects compiles, C or C++ compiles, and prelinking of the client program.

### **Your Mission as a Class Library Builder:**

Here's your mission as a class library builder programming in "C" for a "C" client:

- You need to design a payroll application for human resources at the client's company.
- You will be designing this application using OO concepts.
- Your design will consist of a group of classes that model your client's payroll requirements.
- Your DLL will consist of eight classes, including:
  - ***SOMObject***

*SOMObject* is the root class for all SOMobjects classes. All SOMobjects classes must be subclasses of *SOMObject* or of some other class derived from *SOMObject*. They do inherit several methods that provide the behavior required of all SOMobjects objects.

- ***check***

*check* provides a generic "check" template which can be printed and provides information such as the payee and the amount. The *check* class inherits from *SOMObject*.

- ***pay\_check***

*pay\_check* is an extension of the *check* class that is used to print employee paychecks. The relationship of *pay\_check* to *check* is an example of *inheritance*.

- ***person***

*person* inherits from *SOMObject* and provides basic information for all people, such as their name, address and home phone number.

- ***tax\_payer***

`tax_payer` inherits from `SOMObject` and is a generic class that provides information for all taxpayers, whether personal or corporate. It contains the social security number or tax number/ID and the amount of tax to pay.

- ***employee***

As well as including the person and `tax_payer` attributes, the `employee` class gives you new information and capabilities with all employees, such as hiring and promoting employees. The `employee` class is an example of multiple inheritance. `employee` has an attribute, named `check_obj`, which uses (or is an instance of) the `pay_check` class.

- ***salary\_employee***

`salary_employee` is a specialization of, and inherits from, the `employee` class. It adds the capability of handling an annual salary.

- ***hourly\_employee***

`hourly_employee` is a specialization of, and inherits from, the `employee` class. It adds the capability of handling an hourly salary.

Figure 8-1 on page 8-6 shows the hierarchy of the “Payroll” class library, along with the attributes and methods of its classes.

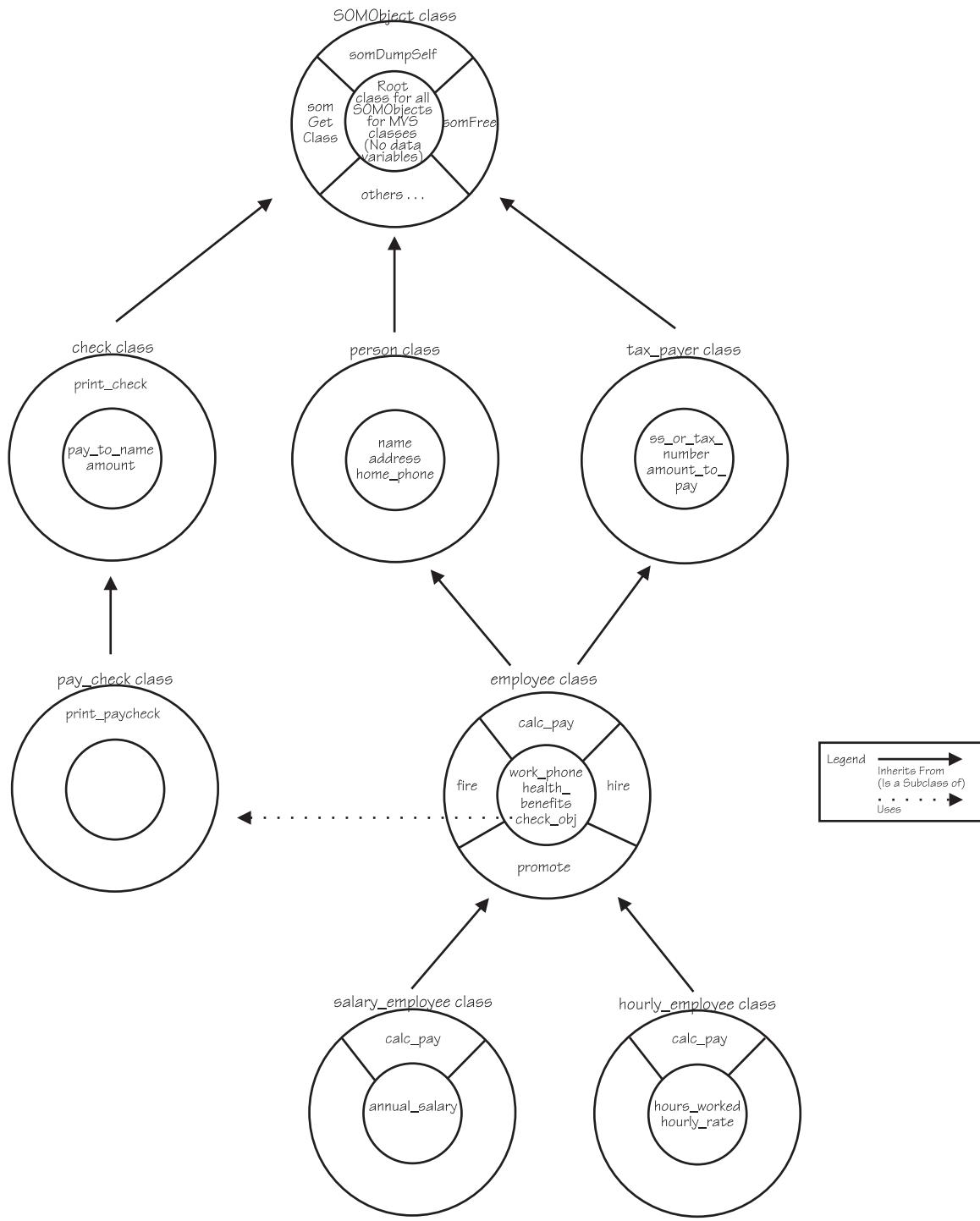


Figure 8-1. "Payroll" class hierarchy with data attributes and methods of its classes.

## Step 2. Create the Source IDL

Allocate a partitioned data set with a member for your source IDL code in the format `dataset_stem.idl(member)`. (Refer to "Step 1. Determine Classes/Objects Needed and SOM Runtime Option/Mode/Services to use" on page 8-4 for information.)

Define the interface to objects of the new class (that is, the interface declaration), by adding the IDL syntax to the member.

Figure 8-2 on page 8-8 shows the IDL statements necessary to define the “Payroll” class library. These IDL statements can be found in *sommvs.SGOSSMPI.IDL(PAYROLL)*.

**Note:** *sommvs* is the high level qualifier we refer to in these default data sets. Your system administrator may have modified the high level qualifier or the entire names of these default data sets on your system.

This IDL demonstrates how the concept of overriding methods can be used in SOM. In Figure 8-2 on page 8-8, the employee class introduces the *calc\_pay* method. Both the *salary\_employee* and *hourly\_employee* classes extend the *calc\_pay* method by overriding the functionality of the inherited *calc\_pay* method from *employee*. This allows both the *salary\_employee* and *hourly\_employee* classes to have a specialized implementation of the *calc\_pay* method.

The *employee* class introduces the abstract method *calc\_pay()*. This method is abstract in the sense that it has no meaningful implementation for an *employee* instance (although it could have). It is used as an anchor that will eventually permit a client application to exploit the concept of polymorphism. In the “Payroll” example, polymorphism is achieved for client applications by introducing classes such as *salary\_employee* and *hourly\_employee* that provide specific implementation of the overridden *calc\_pay()* method.

Defining the class library in this manner, enables client applications that use this library to exploit polymorphism at Runtime. Note that the design done in Figure 8-1 on page 8-6 maps one-to-one with the class interfaces defined in Figure 8-2 on page 8-8.

```

#include <somobj.idl>

module payroll {

 /** ***** Check Class *****/
 interface check : SOMObject {
 // Attributes
 #ifdef __PRIVATE__
 attribute string pay_to_name;
 attribute float amount;
 #endif /* __PRIVATE__ */
 // Methods
 #ifdef __PRIVATE__
 void print_check();
 #endif /* __PRIVATE__ */
 #ifdef __SOMIDL__
 // SOM Extentions to IDL
 implementation {
 #ifdef __PRIVATE__
 releaseorder: _get_pay_to_name, _set_pay_to_name,
 _get_amount, _set_amount,
 print_check;
 #else
 releaseorder: dummy1, dummy2,
 dummy3, dummy4,
 dummy5;
 #endif /* __PRIVATE__ */
 };
 #endif /* __SOMIDL__ */
 }; /* end interface check */

 /** ***** Pay Check Class *****/
 interface pay_check : check {
 // Attributes
 // none
 // Methods
 #ifdef __PRIVATE__
 void print_paycheck(in char *name, in float dollars,
 in int benefits, in char *ss);
 #endif /* __PRIVATE__ */
 #ifdef __SOMIDL__
 // SOM Extentions to IDL
 implementation {
 #ifdef __PRIVATE__
 releaseorder: print_paycheck;
 #else
 releaseorder: dummy1;
 #endif /* __PRIVATE__ */
 };
 #endif /* __SOMIDL__ */
 }; /* end interface pay_check */
}

```

Figure 8-2 (Part 1 of 3). "Payroll" class library IDL statements.

```

//# **** Person Class *****
interface person : SOMObject {
 //# Attributes
 attribute string name;
 attribute string address;
 attribute string home_phone;
#endif _SOMIDL_
 //# SOM Extentions to IDL
 implementation {
 releaseorder: _get_name, _set_name,
 _get_address, _set_address,
 _get_home_phone, _set_home_phone;
 somUninit: override;
 };
#endif /* _SOMIDL__ */
}; /* end interface person */

//# **** Tax Payer Class *****
interface tax_payer : SOMObject {
 //# Attributes
 attribute string ss_or_tax_number;
 attribute float amount_to_pay;
#endif _SOMIDL_
 //# SOM Extentions to IDL
 implementation {
 releaseorder: _get_ss_or_tax_number, _set_ss_or_tax_number,
 _get_amount_to_pay, _set_amount_to_pay;
 somUninit: override;
 };
#endif /* _SOMIDL__ */
}; /* end interface tax_payer */

//# **** Employee Class *****
interface employee : person, tax_payer {
 //# Attributes
 attribute string work_phone;
 attribute int health_benefits;
 attribute pay_check check_obj;
 //# Methods
 void calc_pay(); /* abstract method */
 void hire();
 void fire();
 void promote();
#endif _SOMIDL_
 //# SOM Extentions to IDL
 implementation {
 functionprefix = base_;
 releaseorder: _get_work_phone, _set_work_phone,
 _get_health_benefits, _set_health_benefits,
 _get_check_obj, _set_check_obj,
 calc_pay,
 hire, fire, promote;
 };
#endif /* _SOMIDL__ */
}; /* end interface employee */

```

Figure 8-2 (Part 2 of 3). "Payroll" class library IDL statements.

```

/** ***** Salary Employee Class *****/
interface salary_employee : employee {
 // Attributes
 attribute float annual_salary;
#endif _SOMIDL_
implementation {
 releaseorder: _get_annual_salary, _set_annual_salary;
 functionprefix = s_;
 calc_pay: override; 1
};
#endif /* _SOMIDL_ */
}; /* end interface salary_employee */

/** ***** Hourly Employee Class *****/
interface hourly_employee : employee {
 // Attributes
 attribute int hours_worked;
 attribute float hourly_rate;
#endif _SOMIDL_
implementation {
 releaseorder: _get_hours_worked, _set_hours_worked,
 _get_hourly_rate, _set_hourly_rate;
 functionprefix = h_;
 calc_pay: override; 2
};
#endif /* _SOMIDL_ */
}; /* end interface hourly_employee */
}; /* end module payroll */

```

Figure 8-2 (Part 3 of 3). “Payroll” class library IDL statements.

**Adding attributes with IDL:** Declaring an attribute causes accessor methods (`_get_` and `_set_`) to be automatically defined. For example, specifying:

```
attribute float amount;
```

causes SOM to generate the following two methods:

```
float _get_amount();
void _set_amount(in float amount);
```

Thus, for convenience, an attribute can be used (rather than an instance variable) in order to use the automatically defined `_get_` and `_set_` methods without having to write their method procedures.

**Overriding a method using IDL:** To override the `calc_pay` method in “Payroll”, additional information must be provided in `hlq.sgossmpi.idl(payroll)` (where `hlq` is your user-definable high level qualifier) in the form of an **implementation** statement, which gives extra information about the class, its methods and attributes, and any instance variables.

In the expanded “Payroll” example IDL in 1 and 2, the “implementation” statements for classes `salary_employee` and `hourly_employee` introduce the **override** keyword for method `calc_pay`. The **override** keyword is a method *modifier*.

Here, calc\_pay introduces a *modifier* for the (inherited) calc\_pay method in the classes salary\_employee and hourly\_employee. Modifiers are like C/C ++ #pragma commands and give specific implementation details to the compiler. This example uses only one modifier, “override”. Because of the “override” modifier, when calc\_pay is invoked on an instance of class salary\_employee or hourly\_employee, their implementation of calc\_pay (in the implementation template) will be called, instead of the implementation inherited from the parent class, *employee*.

The “#ifdef\_\_SOMIDL\_\_” and “#endif” are standard C and C++ preprocessor commands that cause the “implementation” statement to be read only when using the SOMobjects IDL compiler (and not some other IDL compiler).

### Step 3. Run the SOMobjects Compiler

The following information provides an overview of the process to run the SOMobjects compiler. For more details, refer to Chapter 4, “SOMobjects Compiler” on page 4-1.

To specify whether the SOMobjects 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 4, “SOMobjects Compiler” on page 4-1. The default is for the SOMobjects Compiler to produce C bindings.

To compile your classes, you must also decide whether you will use the STARS or CORBA implementation.

- STARS Implementation

STARS Implementation has to do with the syntax of an object type in C programs. For “stars”, an object declaration looks like:

```
payroll_pay_check *paycheck;
```

This syntax is compatible with the syntax for declaring a C++ object.

- CORBA Implementation

The other option is CORBA compliant where an object declaration looks like:

```
payroll_pay_check paycheck;
```

This implementation is done with the SMADDSTAR environment variable.

Refer to Appendix A, “Setting up Configuration Files” on page A-1 and the *OS/390 SOMobjects Configuration and Administration Guide* for more information on setting up environment variables.

**SOMobjects Compiler Execution Options:** The JCL in Figure 8-3 on page 8-12 is used to SOM compile the “Payroll” IDL. Note that the SOM Compiler section is highlighted. This JCL can be found in *sommvs.SGOSJCL(GOS1PAY)*.

```

//GOS1PAY JOB <JOB CARD PARAMETERS >
//*
// SET SOM=SOMMVS
// SET LE=CEE
// SET CXX=CBC
// SET IDL=&SOM..SGOSSMPI
// SET HDSN=&SOM..SGOSSMPI
// SET CDSN=&SOM..SGOSSMPC
// SET LDSN=&SOM..SGOSLOAD
/*JOBPARM T=1,L=50
//ORDER JCLLIB ORDER=(&CXX..SCBCPRC,&LE..SCEEPROC)
//-----*
//*
//** GOS1PAY --- Text Processing Sample Application
//*
//** This JCL compiles, and links the PAY Sample application.
//*
//** Before submitting this job, the JCL must be customized
//** for your installation. The following changes need to be
//** made:
//*
//** 1. Update the JOB card with the installation specific
//** parameters.
//** 2. Change the value on the // SET SOM= statement to
//** the high level qualifiers used by SOMobjects for MVS on
//** your system if it is something other than SOMMVS.
//** 3. Change the value on the // SET LE= statement to
//** the high level qualifiers used by IBM LE/370 on
//** your system if it something other than CEE
//** 4. Change the value on the // SET CXX= statement to
//** the high level qualifiers used by IBM C/C++ Compiler on
//** your system if it something other than CBC
//** 5. Make a copy of the SGOSSMPI.IDL dataset that was
//** shipped with SOMobjects for MVS. Change the value on
//** the // SET IDLPRFX= statement to the high level
//** qualifiers of the new data set, NOT including the final
//** .IDL qualifier.
//** 6. Make a copy of the SGOSSMPI.H dataset that was
//** shipped with SOMobjects for MVS. Change the value on
//** the // SET HDSN= statement to the name of the new
//** data set.
//** 7. Make a copy of the SGOSMPC.C dataset that was
//** shipped with SOMobjects for MVS. Change the value on
//** the // SET CDSN= statement to the name of the new
//** data set.
//** 8. Make a copy of the SGOSLOAD dataset that was
//** shipped with SOMobjects for MVS. Change the value on
//** the // SET LDSN= statement to the name of the new
//** data set.
//** 9. This batch job uses the IBM C/C++ Compiler JCL procedures
//** EDCC and EDPL to compile and compile/prelink/link,
//** and the IBM LE/370 JCL procedure EDCPL to prelink.
//** Change these names if they have been changed at your
//** installation.
//**-----*
//** This JCL does the following.
//**-----*
//*
//** 1) SOM Compile the IDL to get the private headers (h & ih).
//**
//** 2) Compile the employee class library, using the private header
//** files.
//**
//** 3) SOM Compile the IDL to get the public headers (h only).
//**
//** 4) Compile the main application, using the public header file.
//*/

```

*Figure 8-3 (Part 1 of 3). JCL to SOM compile "Payroll".*

```

/* 5) Prelink and linkedit the PAYROLL class library to create a DLL.
/*
/* 6) Prelink and linkedit the PAYMAIN client application.
/*
/*
-----*
/* SOM JCL Procedure *
/*
-----*
//SC PROC INDSN=,
// SCPARMS='',
// SOMPRFX=&SOM.,
// LEPRFX=&LE.
//SOMC EXEC PGM=SC,REGION=40M,
// PARM='&SCPARMS.' '&INDSN.'')
//STEPLIB DD DSN=&SOMPRFX..SGOSLOAD,DISP=SHR
// DD DSN=&LEPRFX..SCEERUN,DISP=SHR
//SOMENV DD DSN=&SOMPRFX..SGOSPROF(GOSENV),DISP=SHR
//SYSPRINT DD SYSOUT=*
// PEND
/*
-----*
/* SOM Compile - private *
/*
-----*
//SCPRI EXEC SC,
// SCPARMS='-V -v -p -sh:ih -maddstar',
// INDSN=&IDL..IDL(PAYROLL)
/*
-----*
/* Compile employee class library using private header file *
/*
-----*
//PAYCC EXEC EDCC,
// INFILE=&CDSN..C(PAYROLL),
// CPARM='RENT LO SO SHOW DLL'
//COMPILE.SYSLIB DD
// DD DSN=&HDSN..H,DISP=SHR
// DD DSN=&SOM..SGOSSH.STARS.H,DISP=SHR
// DD DSN=&SOM..SGOSH.H,DISP=SHR
//COMPILE.IH DD DSN=&HDSN..IH,DISP=SHR
/*
-----*
/* SOM Compile - public *
/*
-----*
//SCPUB EXEC SC,
// SCPARMS='-V -v -sh -maddstar',
// INDSN=&IDL..IDL(PAYROLL)
/*
-----*
/* Compile main application using public header file *
/*
-----*
//PAYMAIN EXEC EDCC,
// INFILE=&CDSN..C(PAYMAIN),
// CPARM='RENT LO SO SHOW DLL'
//COMPILE.SYSLIB DD
// DD DSN=&HDSN..H,DISP=SHR
// DD DSN=&SOM..SGOSSH.STARS.H,DISP=SHR
// DD DSN=&SOM..SGOSH.H,DISP=SHR
/*
-----*
/* Run the pre-link and link processing to create *
/* the PAYROLL DLL *
/*
-----*
//PAYRPL EXEC EDCPL,
// INFILE=*.PAYCC.COMPILE.SYSLIN
//PLKED.SYSDEFSD DD DSN=&&IMPORTS(PAYROLL),DISP=(NEW,PASS),
// UNIT=SYSDA,SPACE=(TRK,(3,3,1)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//PLKED.IMPORT DD DSN=&SOM..SGOSIMP,DISP=SHR
//PLKED.SYSIN DD
// DD *
INCLUDE IMPORT(GOSSOMK)
NAME PAYROLL(R)
/*

```

Figure 8-3 (Part 2 of 3). JCL to SOM compile "Payroll".

```

/*
-----*
/* Run the pre-link and link processing to create *
/* the PAYMAIN client program *
-----*
//PAYLINK EXEC EDCPL,
// INFIL=*.PAYMAIN.COMPILE.SYSLIN
//PLKED.SYSDEFSD DD DUMMY
//PLKED.IMPORT DD DSN=&&IMPORTS,DISP=(SHR,DELETE)
// DD DSN=&SOM..SGOSIMP,DISP=SHR
//PLKED.SYSIN DD
// DD *
INCLUDE IMPORT(GOSSOMK)
INCLUDE IMPORT(PAYROLL)
NAME PAYMAIN(R)
//PLKED.SYSLIB DD DSN=&SOM..SGOSPLKD,DISP=SHR
//LKED.SYSLMOD DD DSN=&LDSN.(PAYMAIN),DISP=SHR
//

```

Figure 8-3 (Part 3 of 3). JCL to SOM compile "Payroll".

### Output of the SOMobjects Compiler

Running the SOMobjects Compiler on the IDL data set can produce three data sets:

- Implementation template: to contain the application-specific code
- Implementer header binding: to be included in the implementation template
- Client program header binding: to be included in client programs that use the class.

Running the following SOM Compiler commands

```
sc -sc:h:ih 'hlq.sgoosmpi.idl(payroll)'
```

produces the following three outputs:

- The ih binding for implementers that would be created would be found in *hlq.SGOSSMPI.IH(PAYROLL)*.
- The h binding for the client program that would be created would be found in *hlq.SGOSSMPI.H(PAYROLL)*.
- The C implementation template that would be created would be found in *hlq.SGOSSMPI.C(PAYROLL)* where *hlq* is your user-definable high level qualifier. The contents of the implementation template can be found in Figure 8-4 on page 8-15.

```

??=ifdef __COMPILER_VER_
 ??=pragma filetag ("IBM-1047")
 ??=endif

#pragma nosequence nomargins
#ifndef _MVS
#define _MVS
#endif

/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using template emitter:
 * SOM Emitter emitctm: 2.48
 */

#define SOM_Module_payroll_Source

#include <dd:ih(PAYROLL)>

SOM_Scope void SOMLINK payroll_personsomUninit(payroll_person somSelf)
{
 payroll_personData *somThis = payroll_personGetData(somSelf);
 payroll_personMethodDebug("payroll_person", "payroll_personsomUninit");
}

SOM_Scope void SOMLINK payroll_tax_payersomUninit(payroll_tax_payer somSelf)
{
 payroll_tax_payerData *somThis = payroll_tax_payerGetData(somSelf);
 payroll_tax_payerMethodDebug("payroll_tax_payer", "payroll_tax_payersomUninit");
}

/*
 * abstract method
 */
SOM_Scope void SOMLINK base_calc_pay(payroll_employee somSelf,
 Environment *ev)
{
 payroll_employeeData *somThis = payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug("payroll_employee", "base_calc_pay");
}

```

Figure 8-4 (Part 1 of 2). “Payroll” class library C implementation template created with the SOMobjects compiler.

```

SOM_Scope void SOMLINK base_hire/payroll_employee somSelf, Environment *ev)
{
 payroll_employeeData *somThis = payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug("payroll_employee", "base_hire");
}

SOM_Scope void SOMLINK base_fire/payroll_employee somSelf, Environment *ev)
{
 payroll_employeeData *somThis = payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug("payroll_employee", "base_fire");
}

SOM_Scope void SOMLINK base_promote/payroll_employee somSelf,
Environment *ev)
{
 payroll_employeeData *somThis = payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug("payroll_employee", "base_promote");
}

SOM_Scope void SOMLINK s_calc_pay/payroll_salary_employee somSelf,
Environment *ev)
{
 payroll_salary_employeeData *somThis = payroll_salary_employeeGetData(somSelf);
 payroll_salary_employeeMethodDebug("payroll_salary_employee", "s_calc_pay");

 payroll_salary_employee_pcall_calc_pay(somSelf, ev);
}

SOM_Scope void SOMLINK h_calc_pay/payroll_hourly_employee somSelf,
Environment *ev)
{
 payroll_hourly_employeeData *somThis = payroll_hourly_employeeGetData(somSelf);
 payroll_hourly_employeeMethodDebug("payroll_hourly_employee", "h_calc_pay");

 payroll_hourly_employee_pcall_calc_pay(somSelf, ev);
}

```

Figure 8-4 (Part 2 of 2). “Payroll” class library C implementation template created with the SOMobjects compiler.

## Step 4. Update the Implementation Template

After compiling your source IDL code with the SOMobjects compiler, customize your class implementation by adding code to the implementation template (.c or .cxx data set member).

You need to understand how SOMobjects classes are implemented which includes the following topics:

- Implementing SOMobjects classes
- The implementation template
- Extending the implementation template

## Implementing SOMobjects Classes

The IDL specification for a class defines only the *interface* to the class's instances. The *implementation* of those objects (the procedures that perform their methods) is defined in an implementation data set. To assist users in implementing classes, the SOMobjects Compiler produces an implementation template—a type-correct guide for how the implementation of a class should look. The class implementer then modifies this template to implement the class's methods.

The SOMobjects Compiler can also update the implementation data set to reflect later changes made to a class's interface definition data set (the IDL data set). These *incremental updates* include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. These updates to the implementation data set, however, do *not* disturb existing code in the method procedures. These updates are discussed further in "Running Incremental Updates of the Implementation Template" on page 8-23.

**Note:** Implementing a SOMobjects class in C++ means that the new methods introduced by the class, and the methods overridden by the class are written in C++. Implementing a SOMobjects class in C++ does not mean that C++ client applications can subclass the SOMobjects class to create new C++ or SOMobjects classes.<sup>1</sup>

## The Implementation Template

The following is a subsection of the IDL description for the "Payroll" class:

```
#include <somobj.idl>

module payroll {

 //# ***** Check Class *****
 interface check : SOMObject {
 //# Attributes
 attribute string pay_to_name;
 attribute float amount;
 //# Methods
 void print_check();
 }; /* end interface check */

}; /* end module payroll */
```

From this IDL description, the SOMobjects Compiler generates the following C implementation template, (a C++ implementation template is identical except that the #included file is <payroll.xih>):

---

<sup>1</sup> The reason why the C++ implementation of a SOMobjects class involves the definition of C++ procedures (not C++ methods) to support SOMobjects methods is that there is no language-neutral way to call a C++ method. Only C++ code can call C++ methods, and this calling code must be generated by the same compiler that generates the method code. In contrast, the method procedures that implement SOMobjects methods must be callable from any language, without knowledge on the part of the object client as to which language is used to implement the resolved method procedure.

```

#define SOM_Module_payroll_Source

#include <dd:ih(PAYROLL)>

SOM_Scope void SOMLINK payroll_checkpoint_check(payroll_check somSelf,
 Environment *ev)
{
 payroll_checkData *somThis = payroll_checkGetData(somSelf);
 payroll_checkMethodDebug("payroll_check","payroll_checkpoint_check");
}

```

The first line defines the “SOM\_Module\_payroll\_Source” symbol, which is used in the SOMobjects-generated implementation header files for C to determine when to define various functions, such as “payroll\_checkNewClass.” 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 <dd:ih(PAYROLL)> for C in an OS/390 PDS, #include <payroll.ih> for C in an HFS file, or #include <payroll.xih> for C++) includes the SOM-generated implementation header file. This file defines a **struct** holding the class's attributes, macros for accessing attributes, 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 “print\_check” above, the SOMobjects Compiler generates the following prototype of the stub procedure:

```

SOM_Scope void SOMLINK payroll_checkpoint_check(payroll_check somSelf,
 Environment *ev)

```

The “SOM\_Scope” symbol is defined in the implementation header file as either “extern” or “static,” as appropriate. The term “void” signifies the return type of method “print\_check”. The “SOMLINK” symbol is defined by SOMobjects; it represents the keyword needed to link to the C or C++ compiler, and its value is system-specific. Using the “SOMLINK” symbol allows the code to work with a variety of compilers without modification.

Following the “SOMLINK” symbol is the name of the procedure that implements the method. If no **functionprefix** modifier has been specified for the class, then the procedure name is <moduleName>\_<className>\_<methodName>. If a **functionprefix** modifier is in effect, then the procedure name is generated by preceding 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 “print\_check” would be:

```
SOM_Scope void SOMLINK xx_print_check(payroll_check somSelf, Environment *ev)
```

The **functionprefix** can not 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 SOMobjects method always receives at least one argument (a pointer to the SOMobjects 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 require this convention.) The somSelf parameter is a pointer to an object that is an instance of the class being implemented (here, class “payroll\_check”) 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 *OS/390 SOMobjects Messages, Codes, and Diagnosis* and the book *The Common Object Request Broker: Architecture and Specification*, published by Object Management Group and X/Open.

The first statement in the stub procedure for method “print\_check” is the statement:

```
payroll_checkData *somThis = payroll_checkGetData(somSelf);
```

This statement sets up addressability to the class' attributes. If the class does not introduce any new attributes, this statement will be enclosed in a block comment. The purpose of this statement, for classes that do introduce attributes, is to initialize a local variable (somThis) that points to a structure representing the attributes introduced by the class. The somThis pointer is used by the macros defined in the “payroll\_check” implementation header file to access those attributes. (These macros are described below.) In this example, the “payroll\_check” class introduces

attributes, so the statement is not commented out. If a class had no attributes, then this statement would be commented out.

The “payroll\_checkData” type and the “payroll\_checkGetData” macro used to initialize the somThis pointer are defined in the implementation header file. Within a method procedure, class implementers can use the somThis pointer to access instance data, or they can use the convenience macros defined for accessing each attribute, as described below.

To implement a method so that it can modify a local copy of an object's instance data without affecting the object's real instance data, declare a variable of type <className>Data (for example, <className>Data “payroll\_checkData”) and assign to it the structure that somThis points to; then make the somThis pointer point to the copy. For example:

```
payroll_checkData myCopy = *somThis;
somThis = &myCopy;
```

Next in the stub procedure for method “print\_check” is the statement:

```
payroll_checkMethodDebug(“payroll_check”, “payroll_checkprint_check”);
```

This statement facilitates debugging. The “payroll\_checkMethodDebug” macro is defined in the implementation header file. It takes two arguments, a class name and a method name. If debugging is turned on (that is, if global variable SOM\_TraceLevel is set to one in the calling program), the macro produces a message each time the method procedure is entered. If the debugging is turned on (that is, if global variable SOM\_TraceLevel is set to 3, 4, or 5 in the calling program), the macro produces a message each time the method procedure is entered. (See *OS/390 SOMobjects Messages, Codes, and Diagnosis* for information on debugging with SOMobjects.) Debugging can be permanently disabled (regardless of the setting of SOM\_TraceLevel in the calling program) by redefining the <className>MethodDebug macro to be SOM\_NoTrace following the #include directive for the implementation header file. (This can yield a slight performance improvement.) For example, to permanently disable debugging for the “payroll\_check” class, insert the following lines in the *dataset\_stem.c(payroll)* implementation data set following the line “#include <dd:ih(payroll)>” (“#include <payroll.ih>” for C files in the HFS or “#include <payroll.xih>” for classes implemented in C++):

```
#undef payroll_checkMethodDebug
#define payroll_checkMethodDebug(c,m) SOM_NoTrace(c,m)
```

The way in which the stub procedure ends is determined by whether the method is a new or an overriding method.

- For non-overriding (new) methods, the stub procedure ends with a return statement (unless the return type of the method is **void**). The class implementer must customize this return statement.
- For overriding methods, the stub procedure ends by making a “parent method call” for each of the class's parent classes. If the method has a return type that is not **void**, the last of these parent method calls is returned as the result of the method procedure. The class implementer can customize this return statement if needed (for example, if some other value is to be returned, or if the parent method calls should be made before the method procedure's own processing). See the next section for a discussion of parent method calls.

If a **classinit** modifier was specified to designate a user-defined procedure that will initialize the “payroll\_check” class object, as in the statement:

```
classinit = checkInit;
```

then the implementation template would include the following stub procedure for “checkInit”, in addition to the stub procedures for payroll\_check's methods:

```
void SOMLINK checkInit(SOMClass *cls)
{
}
```

This stub procedure is then filled in by the class implementer. If the class definition specifies a **functionprefix** modifier, the **classinit** procedure name is generated by pre-pending 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 implementers can also use any of the functions, methods, and macros SOMobjects provides for manipulating classes and objects.

In addition to the functions, methods, and macros SOMobjects provides for both class clients and class implementers, SOMobjects provides two facilities especially for class implementers. They are for (1) accessing attributes of the object responding to the method and (2) making parent method calls, as follows.

**Accessing Attributes Within Class Implementations:** Attributes can be directly accessed only within the implementation data set of the class that introduces the attribute, and not within the implementation of subclasses or within client programs. (To allow access to instance data from a subclass or from client programs, use an *attribute* rather than an instance variable to represent the instance data.) For C++ programmers, the *\_variableName* form is available only if the macro **VARIABLE\_MACROS** is defined (that is, **#define VARIABLE\_MACROS**) in the implementation data set prior to including the XIH binding for the class.

To access attributes within class implementations, class implementers can use either of the following forms:

`_variableName`  
`somThis->variableName`

To access attributes “a”, “b”, and “c”, for example, the class implementer 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.

**Making Parent Method Calls:** In addition to macros for accessing instance variables, the implementation header data set that the SOMObjects Compiler generates also contains definitions of macros for making “parent method calls.” When a class overrides a method defined by one or more of its parent classes, often the new implementation simply needs to augment the functionality of the existing implementation(s). Rather than completely re-implementing the method, the overriding method procedure can conveniently invoke the procedure that one or more of the parent classes uses to implement that method, then perform additional computation (redefinition) as needed. The parent method call can occur anywhere within the overriding method.

The SOMObjects-generated implementation header data set defines the following macros for making parent-method calls from within an overriding method:

- `<className>_parent_<parentClassName>_<methodName>`

**Note:** There is a macro of the above format defined for each parent class of the class overriding the method.

- `<className>_parents_<methodName>`.

For example, given class “employee” with parents “person” and “tax\_payer” and overriding method `somInit` (the SOMObjects method that initializes each object), the SOMObjects Compiler defines the following macros in the implementation header data set for “employee”:

```
employee_parent_tax_payer_somInit
employee_parent_person_somInit
employee_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>`. Therefore, using the macro “`employee_parent_person_somInit`” invokes the person class’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, “`employee_parents_somInit`” would invoke person class’s implementation of `somInit`, followed by tax\_payer class’s implementation of `somInit`. The `<className>_parents_<methodName>` macro is redefined in the binding 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 SOMobjects Classes:** For C++ programmers implementing SOMobjects classes, SOMobjects provides a macro that simplifies the process of converting C++ classes to SOMobjects 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 data set is used, the macro **METHOD\_MACROS** must be defined (that is, use `#define METHOD_MACROS`) prior to including the XIH data set for the class.

**Running Incremental Updates of the Implementation Template:** Refining the IDL data set for a class is typically an iterative process. For example, after running the IDL source data set through the SOMobjects Compiler and writing some code in the implementation template, the class implementer 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 SOMobjects Compiler (when run using the **c** or **xc** emitter) assists in this development by reprocessing the IDL data set and making *incremental updates* to the current implementation data set. 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 data set for any new methods added to the IDL data set.
- New comments in the IDL data set are transferred to the implementation data set, reformatted appropriately.
- If the interface to a method has changed, a new method procedure prototype is placed in the implementation data set. As a precaution, however, the old prototype is also preserved within comments. The body of the method procedure is left untouched (as are the method procedures for all methods).
- Similarly left intact are preprocessor directives, data declarations, constant declarations, non-method functions, and additional comments—in essence, everything else in the implementation data set.

Some changes to the IDL data set are *not* reflected automatically in the implementation data set after an incremental update. The class implementer must manually edit the implementation data set after changes such as these in the IDL data set:

- Changing the name of a class or a method.
- Changing a **functionprefix** class **modifier** statement.

- Changing the content of a **passthru** statement directed to the implementation data set. As previously emphasized, however, **passthru** statements are primarily recommended only for placing #include statements in a binding (IH, XIH, H, or XH data set) used as a header data set in the implementation data set or in a client program.
- If the class implementer has placed “forward declarations” of the method procedures in the implementation data set, those are not updated. Updates occur only for method prototypes that are part of the method procedures themselves.

To ensure that the SOMobjects Compiler can properly update method procedure prototypes in the implementation data set, class implementers should avoid editing changes such as 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. Thus, a newline must *not* be inserted before the procedure name.

Error messages occur while updating an existing implementation data set if it contains syntax that is not ANSI C. For example, “old style” method definitions such as the example on the left generate errors:

| <u>Invalid “old” syntax</u> | <u>Required ANSI C</u> |
|-----------------------------|------------------------|
| void foo(x)                 | void foo(short x) {    |
| short x;                    | ...                    |
| {                           |                        |
| ...                         |                        |
| }                           |                        |

Similarly, error messages occur if anything in the IDL data set would produce an implementation data set that is not syntactically valid for C/C++ (such as nested comments). If update errors occur, either the IDL data set or the implementation data set may be at fault. One way to track down the problem is to run the implementation data set through the C/C++ compiler. Or, move the existing implementation data set to another dataset, generate a completely new one from the IDL data set, and then run *it* through the C/C++ compiler. One of these steps should pinpoint the error, if the compiler is strict ANSI.

Conditional compilation (using #if and #ifdef directives) in the implementation data set can be another source of errors, because the SOMobjects Compiler does not invoke the preprocessor (it simply recognizes and ignores those directives). The programmer should be careful when using conditional compilation, to avoid a situation such as shown below; here, with apparently two open braces and only one closing brace, the **c** or **xc** emitter would report an unexpected end-of-file:

| <u>Invalid syntax</u> | <u>Required matching braces</u> |
|-----------------------|---------------------------------|
| #ifdef FOOBAR         | #ifdef FOOBAR                   |
| {                     | {                               |
| ...                   | ...                             |
| #else                 | }                               |
| {                     | #else                           |
| ...                   | {                               |

```
#endif ...
}
```

```
#endif
```

Now that you understand what the implementation template is and how to build and extend it, we will continue building upon the “Payroll” class library example by putting the implementation code into the template.

### **“Payroll” Class Library Implementation Code (in C)**

In Figure 8-5 on page 8-26, we have placed the C implementation code into the template for the “Payroll” example. This code, shipped in *sommvs.SGOSSMPC.C(PAYROLL)*, will allow each of the methods with implementation code in “Payroll” to function.

```

??=ifdef __COMPILER_VER__
 ??=pragma filetag ("IBM-1047")
??=endif

#pragma nosequence nomargins
#ifndef _MVS
#define _MVS
#endif

/*
 * This file was generated by the SOM Compiler.
 * Generated using:
 * SOM incremental update: <current version>
 * Timestamp: <current time stamp>
 *
 */

#define SOM_Module_payroll_Source
#include <DD:IH(PAYROLL)>

/* user required header files */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>

/* **** */
/* Begin section for class library support functions */

char *aNumtoWord(char *aNum) {
/*****
 * description: This function converts an ascii representation
 * of a string of numbers ('aNum') into a textual
 * representation and returns a pointer to the
 * new string. This function does not handle numbers
 * of ten thousand or greater.
 * side effects: Each invocation of this function modifies
 * the internal array pointed to by the return
 * value.
*****/
static char word[70];
char *numnames[] = { "", "one ", "two ", "three ",
 "four ", "five ", "six ", "seven ",
 "eight ", "nine "};

char *tensnames[] = { "", "", "twenty ", "thirty ",
 "forty ", "fifty ", "sixty ",
 "seventy ", "eighty ", "ninety "};

char *teennames[] = { "ten", "eleven", "twelve",
 "thirteen", "fourteen", "fifteen",
 "sixteen", "seventeen", "eighteen",
 "nineteen" };

word[0] = '\0'; /* reset string to NULL */

/* Handle Thousands */
if(*(++aNum) != ' ') {
 strcat(word, numnames[(atoi(aNum)/1000)]);
 strcat(word, "thousand ");
}

/* Handle Hundreds */
if(*(++aNum) != ' ' && *(aNum) != '0') {
 strcat(word, numnames[(atoi(aNum)/100)]);
 strcat(word, "hundred ");
}

```

Figure 8-5 (Part 1 of 6). “Payroll” class library implementation code.

```

/* Handle Tens */
if(*(++aNum) != ' ') {
 if(*(aNum) == '1') {
 strcat(word, teennames[(atoi(aNum)%10)]);
 }
 else {
 strcat(word, tensnames[(atoi(aNum)/10)]);
 /* Handle Ones */
 if(*(aNum+1) != '0')
 strcat(word, numnames[(atoi(aNum)%10)]);
 }
}
else {
 /* Handle Ones */
 if((*(aNum+1) != ' ') || (*(aNum+1) != '0'))
 strcat(word, numnames[(atoi(aNum)%10)]);
}

return(word);
}

/* End section for class library support functions */
/* **** */

SOM_Scope void SOMLINK \
payroll_checkpoint_check(
 payroll_check *somSelf, Environment *ev)
{
 int x, end;
 char *point, *dolls;
 char amt[10] = {' '};
 char day[14];
 time_t temp;
 struct tm *timeptr;

 payroll_checkData *somThis =
 payroll_checkGetData(somSelf);
 payroll_checkMethodDebug(
 "payroll_check",
 "payroll_checkpoint_check");
 temp = time(NULL);
 timeptr = localtime(&temp);
 strftime(day, sizeof(day)-1, "%b %d, %Y", timeptr);
 sprintf(amt, "%8.2f", _amount);
 point = strchr(amt, '.');
 printf("*****\n"
 "*****\n"
 "* Demo, Corp. "
 "\n"
 "* TechNo, NY 54321 "
 "\n"
 " Date: %s *\n"
 "* "
 "\n"
 "* Pay to the "
 "\n"
 " "
 "\n"
 "* order of %s",
 day, _get_pay_to_name(somSelf,ev));
 end = (47 - strlen(_get_pay_to_name(somSelf,ev)));
 for(x = 0; x <= end; x++)
 printf(" ");
 printf(" $ %8.2f *\n", _amount);
 *point = '\0';
 dolls = aNumtoWord(amt);
 dolls[0] = (char) toupper((int) dolls[0]);
 printf("* %sand %s/100 ", dolls, (point+1));
 end = (52 - strlen(dolls));
}

```

Figure 8-5 (Part 2 of 6). "Payroll" class library implementation code.

Figure 8-5 (Part 3 of 6). “Payroll” class library implementation code.

```

SOM_Scope void SOMLINK \
 payroll_personsomUninit(
 payroll_person *somSelf)
{
 payroll_personData *somThis =
 payroll_personGetData(somSelf);
 payroll_personMethodDebug(
 "payroll_person",
 "payroll_personsomUninit");

 /* clean up storage to avoid a memory leak */
 if(somThis->name)
 SOMFree(somThis->name);

 if(somThis->address)
 SOMFree(somThis->address);

 if(somThis->home_phone)
 SOMFree(somThis->home_phone);

 payroll_person_parent_SOMObject_somUninit(
 somSelf);
}

SOM_Scope void SOMLINK \
 payroll_tax_payersomUninit(
 payroll_tax_payer *somSelf)
{
 payroll_tax_payerData *somThis =
 payroll_tax_payerGetData(somSelf);
 payroll_tax_payerMethodDebug(
 "payroll_tax_payer",
 "payroll_tax_payersomUninit");

 /* clean up storage to avoid a memory leak */
 if(somThis->ss_or_tax_number)
 SOMFree(somThis->ss_or_tax_number);

 payroll_tax_payer_parent_SOMObject_somUninit(
 somSelf);
}
/*
 * abstract method
 */

SOM_Scope void SOMLINK \
 base_calc_pay(
 payroll_employee *somSelf, Environment *ev)
{
 payroll_employeeData *somThis =
 payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug(
 "payroll_employee",
 "base_calc_pay");
}

```

Figure 8-5 (Part 4 of 6). “Payroll” class library implementation code.

```

SOM_Scope void SOMLINK \
base_hire(
 payroll_employee *somSelf, Environment *ev)
{
 char buf[100];
 payroll_employeeData *somThis =
 payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug(
 "payroll_employee",
 "base_hire");

 printf("Enter name: \n");
 __set_name(somSelf, ev,
 strcpy(SOMMalloc(strlen(gets(buf))+1), buf));

 printf("Enter SS Number: \n");
 __set_ss_or_tax_number(somSelf, ev,
 strcpy(SOMMalloc(strlen(gets(buf))+1), buf));

 if(__get_health_benefits(somSelf, ev)) {
 printf("Enter Annual Wage: \n");
 __set_annual_salary(somSelf, ev, atof(gets(buf)));
 }
 else {
 printf("Enter Hourly Wage: \n");
 __set_hourly_rate(somSelf, ev, atof(gets(buf)));
 }
}

SOM_Scope void SOMLINK \
base_fire(
 payroll_employee *somSelf, Environment *ev)
{
 payroll_employeeData *somThis =
 payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug(
 "payroll_employee",
 "base_fire");

/* This method is intentionally left blank. It is reserved for
 * future expansion of the employee class.
 */
}

SOM_Scope void SOMLINK \
base_promote(
 payroll_employee *somSelf, Environment *ev)
{
 payroll_employeeData *somThis =
 payroll_employeeGetData(somSelf);
 payroll_employeeMethodDebug(
 "payroll_employee",
 "base_promote");
/* This method is intentionally left blank. It is reserved for
 * future expansion of the employee class.
 */
}

SOM_Scope void SOMLINK \
s_calc_pay(
 payroll_salary_employee *somSelf, Environment *ev)
{
 payroll_salary_employeeData *somThis =
 payroll_salary_employeeGetData(somSelf);
 payroll_salary_employeeMethodDebug(
 "payroll_salary_employee",
 "s_calc_pay");
}

```

Figure 8-5 (Part 5 of 6). “Payroll” class library implementation code.

```

/* calculate weekly pay and print check */
_set_check_obj(somSelf, ev, payroll_pay_checkNew());
_print_paycheck(_get_check_obj(somSelf, ev), ev,
 _get_name(somSelf, ev),
 (_annual_salary / 52),
 _get_health_benefits(somSelf, ev),
 _get_ss_or_tax_number(somSelf, ev));
_somFree(_get_check_obj(somSelf, ev));
}

SOM_Scope void SOMLINK \
 h_calc_pay(
 payroll_hourly_employee *somSelf, Environment *ev)
{
 payroll_hourly_employeeData *somThis =
 payroll_hourly_employeeGetData(somSelf);
 payroll_hourly_employeeMethodDebug(
 "payroll_hourly_employee",
 "h_calc_pay");

 /* calculate weekly pay and print check */
 _set_check_obj(somSelf, ev, payroll_pay_checkNew());
 _print_paycheck(_get_check_obj(somSelf, ev), ev,
 _get_name(somSelf, ev),
 (_hourly_rate * _hours_worked),
 _get_health_benefits(somSelf, ev),
 _get_ss_or_tax_number(somSelf, ev));
 _somFree(_get_check_obj(somSelf, ev));
}

```

Figure 8-5 (Part 6 of 6). "Payroll" class library implementation code.

## Steps 5 and 6: Compile the Implementation Code with the DLL Option, Prelink and Link

**Compiling the Object Deck:** You would now compile this code, which is found in *sommvs.SGOSSMPC.C(PAYROLL)*, through the C compiler using the DLL option on the compiler command invocation. (Use the EXPORTALL option to export all the functions and variables in the DLL.)

**Note:** For C++, the DLL option is not required. Because C++ is an object-oriented language, it automatically creates the stubs required to create a DLL.

Alternatively, you can identify specific functions and variables to be exported by using the #pragma export directive in the DLL source file. The syntax of this directive is:

```
#pragma export (name)
```

where *name* is the name of the function or variable to be exported.

After completing the C compile step to create your object deck, your next steps are to prelink and link this object deck to create a DLL load module.

**Prelinking the object deck:** The prelink JCL step must identify the name of a dataset to receive the definition side deck (SYSDEFSD) generated by the MVS C/C++ Language Support feature on OS/390, using the OS/390 Language Environment. The definition side deck is identified in JCL with the label SYSDEFSD as shown in the following example:

```
//SYSDEFSD DD DSNAME=dataset_stem(member) ,DISP=SHR
```

**Linking the object deck:** Link-edit the prelinked object deck. The output of the prelinker is an updated object deck that contains additional CSECTs to resolve the names and offsets of the DLL's functions and variables. This object deck is used as input to the linkage editor which in turn creates the DLL load module.

**Note:** This load module is not independently executable, and it must be reentrant (for C and C++ it does not contain a *MAIN* function). However, functions and variables exported by this load module are callable and accessible by application programs.

Refer to *sommvs.SGOSJCL(GOS1PAY)*, which shows the C compiler JCL with the EXPORTALL option specified on the C compiler command invocation and the prelink and link JCL to create the DLL load module:

## Running Your Client Program

We will now proceed to implement **Task 2** from Table 8-1 on page 8-3 by discussing steps 1, 2, 3 and 4.

- **Step 1:** Create the application code.
- **Step 2:** Compile the application with the DLL compiler option.
- **Step 3:** Prelink and link-edit the application.
- **Step 4:** Go (run the application program)

The following section, walks you through these four steps:

### Step 1: Create the Application Code

Let's create the application code that will access the "Payroll" DLL load module so that we can run the methods named *calc\_pay* and *hire*. For C and C++ programs, be sure to include a function prototype for the functions that reside in DLLs. These can be explicitly coded or coded in a header file to be included by a #include directive.

Figure 8-6 on page 8-33 is the application code, found in *sommvs.SGOSSMPC.C(PAYMAIN)*, to use the functions that reside in the class library DLL.

```

??=ifdef _COMPILER_VER
 ??=pragma filetag ("IBM-1047")
??=endif

#pragma nosequence nomargins

#include <stdio.h>
#include <stdlib.h>
#include "payroll.h"

/* Link list element containing employee objects */
struct emp_db_entry {
 payroll_employee *emp;
 struct emp_db_entry *next;
};

void add_to_db(struct emp_db_entry **db, payroll_employee *emp) {
/*
 * description: This function is used to add an employee object
 * ('emp') to the employee link list ('db').
 * side effects: The link list is modified by adding the new
 * entry to the top of the list.
 */
 struct emp_db_entry *temp = SOMMalloc(sizeof(struct emp_db_entry));

 temp->next = *db;
 temp->emp = emp;
 *db = temp;
}

int main(int argc, char *argv[]) {
 int count;
 int action = 1;
 char line[20];
 struct emp_db_entry *employee_db = NULL;
 struct emp_db_entry *temp;
 payroll_employee *w_emp;
 Environment *ev = somGetGlobalEnvironment();

 /* prime database with two employees (1 salary and 1 hourly) */
 w_emp = payroll_hourly_employeeNew();
 add_to_db(&employee_db, w_emp);
 __set_name(w_emp, ev, "John Baker");
 __set_ss_or_tax_number(w_emp, ev, "123-45-6789");
 __set_hourly_rate(w_emp, ev, 4.60);
 __set_hours_worked(w_emp, ev, 40);
 __set_health_benefits(w_emp, ev, 0);
}

```

*Figure 8-6 (Part 1 of 2). Client application code which calls the “Payroll” DLL load module.*

```

w_emp = payroll_salary_employeeNew();
add_to_db(&employee_db, w_emp);
__set_name(w_emp, ev, "Jackie Mustard");
__set_ss_or_tax_number(w_emp, ev, "233-65-4321");
__set_annual_salary(w_emp, ev, 76000.00);
__set_health_benefits(w_emp, ev, 1);

printf("Enter operation: 0-Exit\n"
 " 1-Hire Hourly\n"
 " 2-Hire Salary\n"
 " 3-Time Keeping\n"
 " 4-Print Paychecks\n");
action = atoi(gets(line));
while(action) {
 switch(action) {
 case 1 : printf("Hire Hourly Employee\n");
 w_emp = payroll_hourly_employeeNew();
 add_to_db(&employee_db, w_emp);
 /* hourly employees do not get health benefits */
 __set_health_benefits(w_emp, ev, 0);
 _hire(w_emp, ev);
 break;
 case 2 : printf("Hire Salary Employee\n");
 w_emp = payroll_salary_employeeNew();
 add_to_db(&employee_db, w_emp);
 /* salary employees get health benefits */
 __set_health_benefits(w_emp, ev, 1);
 _hire(w_emp, ev);
 break;
 case 3 : printf("Performing Time Keeping\n");
 for(temp=employee_db; temp; temp=temp->next) {
 w_emp = temp->emp;
 /* check if hourly employee */
 if(!(__get_health_benefits(w_emp, ev))) {
 printf("Enter the hours for %s: \n",
 __get_name(w_emp, ev));
 __set_hours_worked(w_emp, ev, atoi(gets(line)));
 }
 }
 break;
 case 4 : printf("Printing Pay Checks\n");
 for(temp=employee_db; temp; temp=temp->next) {
 w_emp = temp->emp;
 _calc_pay(w_emp, ev);
 }
 break;
 } /* end switch */
 printf("Enter operation: 0-Exit\n"
 " 1-Hire Hrly\n"
 " 2-Hire Salary\n"
 " 3-Time Keeping\n"
 " 4-Print Paychecks\n");
 action = atoi(gets(line));
} /* end while */

/* free employee objects */
printf("Cleaning up Employee DataBase\n");
for(temp=employee_db; temp; temp=temp->next) {
 _somFree(temp->emp);
}

return(0);
}

```

Figure 8-6 (Part 2 of 2). Client application code which calls the "Payroll" DLL load module.

## **Steps 2 and 3: Compile the Application with the DLL Compiler Option, Prelink and Link**

***Compiling the application with the DLL compiler option:*** Take the application created in Step 2 and compile it with the DLL compiler option.

### **Notes:**

1. For C++, the DLL option is not required. Because C++ is an object-oriented language, it automatically creates the stubs required to create a DLL.
2. Do not specify the EXPORTALL option for the compile of DLL applications. The EXPORTALL option is only used in building DLLs, not in accessing them.

***Prelinking and Linking the Application:*** In the prelink JCL procedure, be sure to specify an INCLUDE control card for the definition side decks of each DLL used by the application.

Link-edit the prelinked application program to create the application load module.

Refer to *sommvs.SGOSJCL(GOS1PAY)* for the JCL used to compile, prelink and link the application created in Step 2:

We have now:

- Designed a class library named “Payroll”
- Created the IDL for its classes
- SOM compiled the output and created the implementation templates
- Updated the implementation template with C method code
- Created a DLL load module by compiling, prelinking and linking the completed implementation template and
- Created a client application which accesses the “Payroll” DLL load module.

Let's now create the job to set up the environment to run the client application.

## **Step 4: Go (Run the Client Application Calling the “Payroll” Class)**

Whereas the *OS/390 SOMobjects: Getting Started* showed how to run an example in the batch environment, this example will be run from TSO using a REXX exec.

Before you can run the “Payroll” client application program which is named **PAYMAIN**, you have to set up your environment. Since the **PAYMAIN** program is an online interactive program, we will use a REXX exec to set up our environment. This REXX exec is in Figure 8-7 on page 8-36.

```

/* REXX */
/*

 /* Provides setup to run the PAYROLL example in TSO foreground */
 /*
 /-----/
trace 'o'; /* Turn off REXX tracing */
hlq = 'SOMMVS' /* high level qualifier for SOM */
dll="SOM.LOAD"; /* SOMMVS sample DLLs */
ld="'"hlq".SGOSLOAD"'; /* SOMMVS DLLs */
cee="CEE.SCEERUN"; /* Language Environment RTL */
somenv = "'''hlq".SGOSPROF(GOSENV)"'; /* SOM environment dataset */

queue "TSOLIB RESET"

/* Allocate the loadlibs */
say "Allocate "cee", "ld", "dll
say " "
queue "FREE FI(SOMLLIB)"
queue "ALLOC FI(SOMLLIB) DA("cee", "ld", "dll") SHR REU";

/* Allocate the SOM environment. */
say "Allocate "somenv
say " "
queue "FREE FI(SOMENV)"
queue "ALLOC FI(SOMENV) DA("SOMENV") SHR REU";

/* Activate SOMLLIB */
queue "TSOLIB ACTIVATE FILE(SOMLLIB)"
queue "TSOLIB DISPLAY"

exit

```

Figure 8-7. "Payroll" REXX exec to set up your online interactive environment.

After running the above REXX exec, you can enter the following command on your TSO Ready command line and your "Payroll" interactive program will proceed:



PAYMAIN

## Building and Running an Application from the TSO Environment

Previous examples showed how to build and access a non-distributed application in the batch environment. It is also possible to build and access a sample in the TSO environment. This can be done using a REXX exec. A sample REXX exec for building the "PAYROLL" sample application can be found in *sommvs.SGOSEXEC(GOSRPAY)*.

---

## Setting Up and Running Non-distributed Vehicle in Different OS/390 Environments

There are various OS/390 environments that your applications can run in. This section will describe various environments you can use in OS/390 and how to setup and run non-distributed examples in them.

The following are the three environments (with steps and examples) that can be run in OS/390 and each example gives the dataset location in the product for each of those environments.

- “Building and Running a Non-Distributed Application in the Batch Environment”
- “Building and Running a Non-Distributed Application from the OpenEdition Shell Environment”
- “Building and Running a Non-Distributed Application from the TSO Environment” on page 8-39.

## **Building and Running a Non-Distributed Application in the Batch Environment**

See *OS/390 SOMobjects: Getting Started* which will take you through the basic steps of building the non-distributed “Vehicle” example running in the batch environment.

## **Building and Running a Non-Distributed Application from the OpenEdition Shell Environment**

The following example demonstrates a non-distributed SOM example of “Vehicle” running in the OS/390 OpenEdition environment.

The datasets where the OS/390 OpenEdition scripts can be found are *sommvs.SGOSMISC(GOSAUTSP)* and *sommvs.SGOSMISC(GOSDSETU)*.

**Note:** See the *OS/390 SOMobjects Configuration and Administration Guide* for information on how to setup your OS/390 OpenEdition environment. Once your OS/390 OpenEdition environment has been set up, you can run the following example.

### **Building a Non-Distributed Application from the OpenEdition Shell Environment**

The following are the class library builder steps to build a non-distributed “Vehicle” example in the OS/390 OpenEdition environment:

#### 1. Get the IDL

The IDL can be found in *sommvs.SGOSSMPI.IDL(CAR)* and *sommvs.SGOSSMPI.IDL(VEHICLE)*.

```
tso OPUT "'$OMMVS.SGOSSMPI.IDL(VEHICLE)' '$PWD/vehicle.idl'"
tso OPUT "'$OMMVS.SGOSSMPI.IDL(CAR)' '$PWD/car.idl'"
```

#### 2. Tailor the environment variables

For more information, see Appendix A, “Setting up Configuration Files” on page A-1.

```
export SOMENV="$PWD/somenv.ini"
export SOM_TraceLevel=0
export LIBPATH=$PWD
export PATH=$PWD:$PATH:
export _C89_EXTRA_ARGS=1
```

**Note:** The above environment variable information should go into your *.profile*.

3. Ensure that **SMOE=YES** is in your configuration file (somenv.ini) by entering the following:

```
oedit somenv.ini
```

Check your [somc] stanza for the following:

```
[somc]
; Denotes the stanza used by the OS/390 SOMObjects compiler.
;
; SMOE=YES
;
; SMOE is used by the compiler to decide whether it was invoked
; under the OE shell or not. You must set this when in OE.
```

4. SOM Compile

The SOM compile can be found in *sommvs.SGOSMISC(GOSAUTSP)*.

```
sc -sh:ih vehicle.idl
sc -sh:ih car.idl
```

5. Update the Interface Repository

**Note:** Not needed for this sample, but would be needed if the application does not do a NewClass and the dllname is different than the class name.

6. Update the C implementation code.

For this sample program, we have created the C implementation templates, and the following OS/390 OpenEdition commands (found in *sommvs.SGOSMISC(GOSDSETU)*) will copy them:

```
tso OPUT "'SOMMVS.SGOSSMPC.C(VEHICLE)' '$PWD/vehicle.c'"
tso OPUT "'SOMMVS.SGOSSMPC.C(CAR)' '$PWD/car.c'"
```

The following OS/390 OpenEdition commands will update the initial C implementation templates:

```
sc -sc vehicle.idl
sc -sc car.idl
```

7. C Compile, Prelink, Link

This can be found in *sommvs.SGOSMISC(GOSAUTSP)*.

```
c89 -ovehicle -Wc,dll,expo -Wl,dll -I. -I"//'SOMMVS.SGOSH.H'" \
vehicle.c "://"SOMMVS.SGOSIMP(GOSSOMK)"
c89 -ocar -Wc,dll,expo -Wl,dll -I. -I"//'SOMMVS.SGOSH.H'" \
car.c vehicle.x "://"SOMMVS.SGOSIMP(GOSSOMK)"
```

## Running a Non-Distributed Application from the OpenEdition Shell Environment

The following are the client steps to running a non-distributed “Vehicle” example in the OS/390 OpenEdition environment:

1. Decide which objects to use
2. C code

This can be found in *sommvs.SGOSMISC(GOSDSETU)*.

```
tso OPUT "'SOMMVS.SGOSSMPC.C(DRVCAR)' '$PWD/drvcar.c'"
```

3. C Compile, Prelink, Link

This can be found in *sommvs.SGOSMISC(GOSAUTSP)*.

```
c89 -odrvcar -Wc,dll -I . -I // 'SOMMVS.SGOSH.H' \
drvcar.c vehicle.x car.x // 'SOMMVS.SGOSIMP(GOSSOMK)'"
```

4. The SOM subsystem should be started.

See *OS/390 SOMobjects Configuration and Administration Guide* for a description of how to start the SOM subsystem.

5. Start the client by entering:

```
drvcar 4
```

## Building and Running a Non-Distributed Application from the TSO Environment

The following is a non-distributed SOM example of “Vehicle” running in the TSO environment with a REXX exec.

The dataset where this REXX exec can be found is *sommvs.SGOSEXEC(GOSRAUTO)*.

**Note:** Before you run *sommvs.SGOSEXEC(GOSRAUTO)* you must run the REXX exec *sommvs.SGOSEXEC(GOSSETUP)*, which will set up your TSO environments for running the GOSRAUTO exec.

The following steps are included and commented in the REXX exec.

**Note:** Have the SOM subsystem up. See *OS/390 SOMobjects Configuration and Administration Guide* for more information.

## Building a Non-Distributed Application from the TSO Environment

1. Get the IDL

This IDL can be found in

*sommvs.SGOSSMPI.IDL(CAR)*  
*sommvs.SGOSSMPI.IDL(VEHICLE)*

See *sommvs.SGOSEXEC(GOSRAUTO)* for more information.

2. SOM Compile

See *sommvs.SGOSEXEC(GOSRAUTO)* for more information.

3. Update the C implementation template

For this sample program, we have updated the C implementation templates, which can be found in:

*sommvs.SGOSSMPC.C(CAR)*  
*sommvs.SGOSSMPC.C(VEHICLE)*

4. C Compile

See *sommvs.SGOSEXEC(GOSRAUTO)* for more information.

5. Prelink/link

See *sommvs.SGOSEXEC(GOSRAUTO)* for more information.

## Running a Non-Distributed Application from the TSO Environment

1. Decide which objects to use

2. Write C code

This code can be found in

*sommvs.SGOSSMPC.C(DRVCAR)*

3. C Compile

See *sommvs.SGOSEXEC(GOSRAUTO)* for more information.

4. Pre-link/link

See *sommvs.SGOSEXEC(GOSRAUTO)* for more information.

5. Run the client application

See *sommvs.SGOSEXEC(GOSRAUTO)* for more information.

---

## Summarized Non-Distributed Examples

The following sections list the major steps needed to run the following non-distributed examples:

- “Animals”
- “Text Processing” on page 8-42
- “Hello” on page 8-44

## Animals

Table 8-2 on page 8-41 shows the steps and default data set names needed to run “Animals” as a non-distributed SOMobjects client/server application.

The JCL in *sommvs.SGOSJCL(GOS1/VP)* contains all the steps necessary to build and run the “Animals” class in a non-distributed environment.

**Note:** The column labeled “Default Data Set Names” are the data set names as shipped with the product. *sommvs* is the high level qualifier we refer to in these default data sets. Your system administrator may have modified the high level qualifier or the entire names of these default data sets on your system.

Table 8-2. Building and Running “Animals” in a Non-Distributed Environment

| Step                                                                  | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Default Data Set Names                                                |
|-----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------|
| <b>Building “Animals” in a Non-Distributed Environment</b>            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                       |
| 1.                                                                    | SOM compile the IDL. <ul style="list-style-type: none"> <li>— Copy the IDL (<i>sommvs.SGOSSMPI.IDL(ANIMALS)</i>) into your own private data set.</li> <li>— Copy the SOM compile JCL from <i>sommvs.SGOSJCL(GOS1IVP)</i> into your own private data set and point to the above IDL with your SCANIM card.</li> <li>— Modify the names (if necessary) in your copy of the above IDL dataset and the JCL in <i>sommvs.SGOSJCL(GOS1IVP)</i>, and point to the above IDL.</li> </ul>                                                                                                                                                                                                                                                                | <i>sommvs.SGOSSMPI.IDL(ANIMALS)</i><br><i>sommvs.SGOSJCL(GOS1IVP)</i> |
| 2.                                                                    | C Compile. <ul style="list-style-type: none"> <li>— Copy <i>sommvs.SGOSSMPC.C(ANIMALS)</i> into your own private data set.</li> <li>— Modify the name (if necessary) in your copy of <i>sommvs.SGOSSMPC.C(ANIMALS)</i>, and the JCL in <i>sommvs.SGOSJCL(GOS1IVP)</i> (especially the “ANIMALS” step needed for the C compile). Refer to the notes in <i>sommvs.SGOSJCL(GOS1IVP)</i> for more information. Ensure that the INFILe has the correct name in it.               <p><b>Note:</b> Normally when creating a new application, you would modify the generated C template before compiling the program. For this sample program, however, we have done this modification for you so that you can now compile the program.</p> </li> </ul> | <i>sommvs.SGOSSMPC.C(ANIMALS)</i><br><i>sommvs.SGOSJCL(GOS1IVP)</i>   |
| 3.                                                                    | Prelink/Linkedit. <ul style="list-style-type: none"> <li>— Modify the JCL in your copy of <i>sommvs.SGOSJCL(GOS1IVP)</i> and any other parts of the “ANIMALPL” step needed for the prelink and link steps.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | <i>sommvs.SGOSJCL(GOS1IVP)</i>                                        |
| <b>Running “Animals” as a Client in a Non-Distributed Environment</b> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                       |
| 1.                                                                    | Start the client. <ul style="list-style-type: none"> <li>— Copy <i>sommvs.SGOSSMPC.C(ANMAIN)</i> into your own private data set.</li> <li>— Modify your copies of <i>sommvs.SGOSSMPC.C(ANMAIN)</i> and <i>sommvs.SGOSJCL(GOS1IVP)</i> to C compile the “Animals” application, prelink, link and run the main program. Refer to the notes in <i>sommvs.SGOSJCL(GOS1IVP)</i> for more information.</li> <li>— Submit your modified JCL to build the “Animals” class and run it as a client.</li> </ul>                                                                                                                                                                                                                                            | <i>sommvs.SGOSSMPC.C(ANMAIN)</i><br><i>sommvs.SGOSJCL(GOS1IVP)</i>    |

## Text Processing

Table 8-3 shows the steps and default data set names needed to run the “Text Processing” example as a non-distributed SOMobjects client/server application.

The JCL in *sommvs.SGOSJCL(GOS1TP)* contains all the steps necessary to build and run the “Text Processing” class in a non-distributed environment.

**Note:** The column labeled “Default Data Set Names” are the data set names as shipped with the product. *sommvs* is the high level qualifier we refer to in these default data sets. Your system administrator may have modified the high level qualifier or the entire names of these default data sets on your system.

Table 8-3 (Page 1 of 3). Building and Running “Text Processing” in a Non-Distributed Environment

| Step                                                               | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Default Data Set Names                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Building “Text Processing” in a Non-Distributed Environment</b> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 1.                                                                 | <p>SOM compile the IDL.</p> <p>— Copy the following datasets:<br/><i>sommvs.SGOSSMPI.IDL(BLEP)</i><br/><i>sommvs.SGOSSMPI.IDL(BT)</i><br/><i>sommvs.SGOSSMPI.IDL(COLBLK)</i><br/><i>sommvs.SGOSSMPI.IDL(EP)</i><br/><i>sommvs.SGOSSMPI.IDL(FM)</i><br/><i>sommvs.SGOSSMPI.IDL(FOOTER)</i><br/><i>sommvs.SGOSSMPI.IDL(HEADER)</i><br/><i>sommvs.SGOSSMPI.IDL(IPEP)</i><br/><i>sommvs.SGOSSMPI.IDL(LINK)</i><br/><i>sommvs.SGOSSMPI.IDL(LL)</i><br/><i>sommvs.SGOSSMPI.IDL(PAGE)</i><br/><i>sommvs.SGOSSMPI.IDL(SUEP)</i><br/><i>sommvs.SGOSSMPI.IDL(TEXTLINE)</i><br/><i>sommvs.SGOSSMPI.IDL(TXTEP)</i><br/><i>sommvs.SGOSSMPI.IDL(TPWORD)</i><br/><i>sommvs.SGOSSMPI.IDL(WORD)</i><br/><i>sommvs.SGOSJCL(GOS1TP)</i></p> <p>into your own private data sets.</p> <p>— Copy the SOM compile JCL from <i>sommvs.SGOSJCL(GOS1TP)</i> into your own private data set and point to the above IDL.</p> <p>— Modify the names (if necessary) in your copies of the above IDL datasets and the JCL in <i>sommvs.SGOSJCL(GOS1TP)</i>, and point to the above IDL within the “SOM Compile the TP Class IDL” section of your data set.</p> | <i>sommvs.SGOSSMPI.IDL(BLEP)</i><br><i>sommvs.SGOSSMPI.IDL(BT)</i><br><i>sommvs.SGOSSMPI.IDL(COLBLK)</i><br><i>sommvs.SGOSSMPI.IDL(EP)</i><br><i>sommvs.SGOSSMPI.IDL(FM)</i><br><i>sommvs.SGOSSMPI.IDL(FOOTER)</i><br><i>sommvs.SGOSSMPI.IDL(HEADER)</i><br><i>sommvs.SGOSSMPI.IDL(IPEP)</i><br><i>sommvs.SGOSSMPI.IDL(LINK)</i><br><i>sommvs.SGOSSMPI.IDL(LL)</i><br><i>sommvs.SGOSSMPI.IDL(PAGE)</i><br><i>sommvs.SGOSSMPI.IDL(SUEP)</i><br><i>sommvs.SGOSSMPI.IDL(TEXTLINE)</i><br><i>sommvs.SGOSSMPI.IDL(TXTEP)</i><br><i>sommvs.SGOSSMPI.IDL(TPWORD)</i><br><i>sommvs.SGOSSMPI.IDL(WORD)</i><br><i>sommvs.SGOSJCL(GOS1TP)</i> |

Table 8-3 (Page 2 of 3). Building and Running “Text Processing” in a Non-Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Default Data Set Names                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2.   | <p>C Compile.</p> <ul style="list-style-type: none"> <li>— Copy the following .C datasets:</li> </ul> <p style="margin-left: 20px;"><i>sommvs.SGOSSMPC.C(BLEP)</i><br/> <i>sommvs.SGOSSMPC.C(BT)</i><br/> <i>sommvs.SGOSSMPC.C(COLBLK)</i><br/> <i>sommvs.SGOSSMPC.C(EP)</i><br/> <i>sommvs.SGOSSMPC.C(FM)</i><br/> <i>sommvs.SGOSSMPC.C(FOOTER)</i><br/> <i>sommvs.SGOSSMPC.C(HADER)</i><br/> <i>sommvs.SGOSSMPC.C(IPEP)</i><br/> <i>sommvs.SGOSSMPC.C(LINK)</i><br/> <i>sommvs.SGOSSMPC.C(LL)</i><br/> <i>sommvs.SGOSSMPC.C(PAGE)</i><br/> <i>sommvs.SGOSSMPC.C(SUEP)</i><br/> <i>sommvs.SGOSSMPC.C(TEXTLINE)</i><br/> <i>sommvs.SGOSSMPC.C(TXTEP)</i><br/> <i>sommvs.SGOSSMPC.C(TPWD)</i><br/> <i>sommvs.SGOSSMPC.C(WRD)</i><br/> <i>sommvs.SGOSSMPC.C(HELP)</i></p> <p style="margin-left: 20px;">into your own private datasets.</p> <ul style="list-style-type: none"> <li>— Modify the names (if necessary) in your copies of the above .C datasets and the JCL in <i>sommvs.SGOSJCL(GOS1TP)</i>, and point to the above .C datasets within the “C Compile the class implementations” section of your data set. Refer to the notes in <i>sommvs.SGOSJCL(GOS1TP)</i> for more information.</li> </ul> <p><b>Note:</b> Normally when creating a new application, you would modify the generated C template before compiling the program. For this sample program, however, we have done this modification for you so that you can now compile the program.</p> | <i>sommvs.SGOSSMPC.C(BLEP)</i><br><i>sommvs.SGOSSMPC.C(BT)</i><br><i>sommvs.SGOSSMPC.C(COLBLK)</i><br><i>sommvs.SGOSSMPC.C(EP)</i><br><i>sommvs.SGOSSMPC.C(FM)</i><br><i>sommvs.SGOSSMPC.C(FOOTER)</i><br><i>sommvs.SGOSSMPC.C(HADER)</i><br><i>sommvs.SGOSSMPC.C(IPEP)</i><br><i>sommvs.SGOSSMPC.C(LINK)</i><br><i>sommvs.SGOSSMPC.C(LL)</i><br><i>sommvs.SGOSSMPC.C(PAGE)</i><br><i>sommvs.SGOSSMPC.C(SUEP)</i><br><i>sommvs.SGOSSMPC.C(TEXTLINE)</i><br><i>sommvs.SGOSSMPC.C(TXTEP)</i><br><i>sommvs.SGOSSMPC.C(TPWD)</i><br><i>sommvs.SGOSSMPC.C(WRD)</i><br><i>sommvs.SGOSSMPC.C(HELP)</i><br><i>sommvs.SGOSJCL(GOS1TP)</i> |
| 3.   | <p>Prelink/Linkedit.</p> <ul style="list-style-type: none"> <li>— Modify the JCL in your copy of <i>sommvs.SGOSJCL(GOS1TP)</i> and any other parts of the “TP” prelink/link step needed for the prelink and link of “Text Processing”.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | <i>sommvs.SGOSJCL(GOS1TP)</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

#### Running “Text Processing” as a Client in a Non-Distributed Environment

Table 8-3 (Page 3 of 3). Building and Running “Text Processing” in a Non-Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | Default Data Set Names                                                            |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| 1.   | <p>Start the client.</p> <ul style="list-style-type: none"> <li>— Copy <i>sommvs.SGOSSMPC.C(TPMAIN)</i> into your own private data set.</li> <li>— Modify your copies of <i>sommvs.SGOSSMPC.C(TPMAIN)</i> and <i>sommvs.SGOSJCL(GOS1TP)</i> to C compile the “Text Processing” application. Refer to the notes in <i>sommvs.SGOSJCL(GOS1TP)</i> for more information.</li> <li>— Make a copy of the SGOSSMPD.DAT data set that was shipped with SOMobjects. In the TPMAIN member that you copied from SGOSSMPC.C, change SGOSSMPD.DAT to the name of the new data set. (Note, you do not specify a high level qualifier.)</li> <li>— Prelink, link and run the main program.</li> <li>— Submit your modified JCL to build the “Text Processing” class and run it as a client.</li> </ul> | <i>sommvs.SGOSSMPC.C(TPMAIN)</i><br><i>sommvs.SGOSJCL(GOS1TP)</i><br>SGOSSMPD.DAT |

## Hello

Table 8-4 shows the steps and default data set names needed to run “Hello” as a non-distributed SOMobjects client/server application. This sample is in C++.

The JCL in *sommvs.SGOSJCL(GOS1IVPP)* contains all the steps necessary to build and run the “Hello” class in a non-distributed environment.

**Note:** The column labeled “Default Data Set Names” are the data set names as shipped with the product. *sommvs* is the high level qualifier we refer to in these default data sets. Your system administrator may have modified the high level qualifier or the entire names of these default data sets on your system.

Table 8-4 (Page 1 of 2). Building and Running “Hello” in a Non-Distributed Environment

| Step                                                     | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | Default Data Set Names                                               |
|----------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| <b>Building “Hello” in a Non-Distributed Environment</b> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                      |
| 1.                                                       | <p>SOM compile the IDL.</p> <ul style="list-style-type: none"> <li>— Copy the IDL (<i>sommvs.SGOSSMPI.IDL(HELLO)</i>) into your own private data set.</li> <li>— Copy the SOM compile JCL from <i>sommvs.SGOSJCL(GOS1IVPP)</i> into your own private data set and point to the above IDL with the “SCHELO” step.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOS1IVPP)</i> and any other parts of the JCL needed for your environment.</li> </ul> | <i>sommvs.SGOSSMPI.IDL(HELLO)</i><br><i>sommvs.SGOSJCL(GOS1IVPP)</i> |

Table 8-4 (Page 2 of 2). Building and Running "Hello" in a Non-Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Default Data Set Names                                              |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------|
| 2.   | <p>C++ Compile.</p> <ul style="list-style-type: none"> <li>— Copy <i>sommvs.SGOSMPX.CXX(HELLO)</i> into your own private dataset.</li> <li>— Copy the SOM compile JCL from <i>sommvs.SGOSJCL(GOS1IVPP)</i> into your own personal dataset.</li> <li>— Modify your copies of <i>sommvs.SGOSMPX.CXX(HELLO)</i>, and <i>sommvs.SGOSJCL(GOS1IVPP)</i> for the C++ compile). Ensure that the "HELLOCM" step has the correct name in it. Refer to the notes in <i>sommvs.SGOSJCL(GOS1IVPP)</i> for more information.</li> </ul> <p><b>Note:</b> Normally when creating a new application, you would modify the generated CXX template before compiling the program. For this sample program, however, we have done this modification for you so that you can now compile the program.</p> | <i>sommvs.SGOSMPX.CXX(HELLO)</i><br><i>sommvs.SGOSJCL(GOS1IVPP)</i> |
| 3.   | <p>Prelink/Linkedit.</p> <ul style="list-style-type: none"> <li>— Modify the JCL in your copy of <i>sommvs.SGOSJCL(GOS1IVPP)</i> and any other parts of the "HELLOPL" step needed for the prelink and link steps.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | <i>sommvs.SGOSJCL(GOS1IVPP)</i>                                     |

#### Running "Hello" as a Client in a Non-Distributed Environment

|    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                    |
|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|
| 1. | <p>Start the client.</p> <ul style="list-style-type: none"> <li>— Copy <i>sommvs.SGOSMPX.CXX(MAIN)</i> into your own private data set.</li> <li>— Modify your copies of <i>sommvs.SGOSMPX.CXX(MAIN)</i> and <i>sommvs.SGOSJCL(GOS1IVPP)</i> to C++ compile the "Hello" application, prelink, link and run the main program. Refer to the notes in <i>sommvs.SGOSJCL(GOS1IVPP)</i> for more information.</li> <li>— Submit your modified JCL to build the "Hello" class and run it as a client.</li> </ul> | <i>sommvs.SGOSMPX.CXX(MAIN)</i><br><i>sommvs.SGOSJCL(GOS1IVPP)</i> |
|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|

You have now been able to build and access non-distributed classes in various environments with SOMobjects on OS/390.

The next chapter will describe examples using distributed SOMobjects (DSOM).



---

## Chapter 9. Distributed SOMobjects (DSOM) Examples

This chapter will discuss distributed SOMobjects (DSOM) examples and includes the following:

- “Setting Up and Running Distributed SOMobjects (DSOM) Vehicle in Different OS/390 Environments”

In this section, the following OS/390 environments will be used with the “Vehicle” example:

- “Building and Running a Distributed Application in the Batch Environment” on page 9-2
- “Building and Running a Distributed Application from the OpenEdition Shell Environment” on page 9-2
- “Building and Running a Distributed Application from the TSO Environment” on page 9-4

- “Summarized Distributed Examples” on page 9-5

These summarized distributed examples include:

- “Stack” on page 9-5
- “SSStack” on page 9-9
- “Vehicle” on page 9-11
- “Animals” on page 9-14
- “Phone” on page 9-18

---

### Setting Up and Running Distributed SOMobjects (DSOM) Vehicle in Different OS/390 Environments

There are various OS/390 environments that you can run in. This section will describe the following environments you can use in OS/390 and how to setup and run examples in them.

The following are the three environments (with steps and examples) that can be run in OS/390 and each example gives the dataset location in the product for each of those environments.

- “Building and Running a Distributed Application in the Batch Environment” on page 9-2
- “Building and Running a Distributed Application from the OpenEdition Shell Environment” on page 9-2
- “Building and Running a Distributed Application from the TSO Environment” on page 9-4
- “Summarized Distributed Examples” on page 9-5

**Note:** Before you can begin running any of these examples, the SOMobjects environment needs to be configured on your system. Consult with your system administrator to ensure that the SOMobjects environment has been configured. For more information on how to configure the SOMobjects environment, see the *OS/390 SOMobjects Configuration and Administration Guide*.

**Note:** Some of these examples show that **somInit** and **somUninit** are overridden. These methods now execute under the overall control of the **somDefaultInit**

method and the **somDestruct** method. When you, in your program, use the **somDefaultInit** method and the **somDestruct** method instead of the **somInit** and **somUninit** methods, then you must override them as well. See “Multiple Inheritance” on page 2-5 for more information.

## Building and Running a Distributed Application in the Batch Environment

See *OS/390 SOMobjects: Getting Started* which will take you through the basic steps of building the distributed “Vehicle” example running in the batch environment. Further details on the programming issues and techniques involved can be found in “DSOM Tutorial” on page 7-7.

The remainder of this chapter will discuss how to build and run distributed examples in various execution environments.

## Building and Running a Distributed Application from the OpenEdition Shell Environment

The following example demonstrates a distributed SOMobjects (DSOM) example of “Vehicle” running in the OS/390 OpenEdition environment.

The datasets where the OS/390 OpenEdition scripts can be found are *sommvs.SGOSMISC(GOSDAUSP)* and *sommvs.SGOSMISC(GOSDSETU)*.

**Note:** See the *OS/390 SOMobjects Configuration and Administration Guide* for information on how to setup your OS/390 OpenEdition environment, WLM environment and your SOMobjects server. Once your OS/390 OpenEdition environment has been set up, you can run the following example.

The following are the steps to running a distributed “Vehicle” example in the OS/390 OpenEdition environment:

- “Vehicle” distributed class library builder steps:

1. Tailor the environment variables

```
export SOMENV="$PWD/somenv.ini"
export SOM_TraceLevel=0
export LIBPATH=$PWD
export PATH=$PWD:$PATH:
export _C89_EXTRA_ARGS=1
oedit somenv.ini (Ensure SMOE=YES)
```

2. Get the IDL

```
tso OPUT "'SOMMVS.SGOSSMPI.IDL(DVEHICLE)' '$PWD/dvehicle.idl'"
tso OPUT "'SOMMVS.SGOSSMPI.IDL(DCAR)' '$PWD/dcar.idl'"
```

3. SOM Compile

```
sc -sh:ih dvehicle.idl
sc -sh:ih dcar.idl
```

4. Update the Interface Repository

```
oedit somenv.ini (Ensure dvehicle.idl is at end of SOMIR)
sc -sir -u dvehicle.idl
sc -sir -u dcar.idl
```

## 5. Update the C implementation template

```
tso OPUT "'SOMMVS.SGOSSMPC.C(DVEHICLE)' '$PWD/dvehicle.c'"
tso OPUT "'SOMMVS.SGOSSMPC.C(DCAR)' '$PWD/dcar.c'"
sc -sc dvehicle.idl
sc -sc dcar.idl
```

## 6. C Compile, Prelink, Link

```
c89 -odvehicle -Wc,dll,expo -Wl,dll -I. -I"/'SOMMVS.SGOSH.H'" \
dvehicle.c //"'SOMMVS.SGOSIMP(GOSSOMD,GOSSOMK)'"'
c89 -odcar -Wc,dll,expo -Wl,dll -I. -I"/'SOMMVS.SGOSH.H'" \
dcar.c dvehicle.x //"'SOMMVS.SGOSIMP(GOSSOMD,GOSSOMK)'"'
```

## 7. Update the Implementation Repository, register the server with WLM and the security server.

A system administrator normally handles these tasks (see *OS/390 SOMobjects Configuration and Administration Guide* for more information). The following are typical OS/390 OpenEdition commands that your system administrator can use to update the Implementation Repository.

```
regimpl -A -i CARSERVER1
regimpl -a -i CARSERVER1 -c Dvehicle -c Car
```

- “Vehicle” distributed class client steps:

1. Decide which objects to use
2. C code

```
tso OPUT "'SOMMVS.SGOSSMPC.C(DRVCARD)' '$PWD/drvcard.c'"
```

## 3. C Compile, Prelink, Link

```
c89 -odrvcards -Wc,dll -I. -I"/'SOMMVS.SGOSH.H'" \
drvcards.c dvehicle.x dcar.x \
//"'SOMMVS.SGOSIMP(GOSSOMD,GOSSOMK,GOSSOMTC,GOSNMNAM)'"'
```

## 4. The SOM subsystem should be started.

See *OS/390 SOMobjects Configuration and Administration Guide* for a description of how to start the SOM subsystem.

## 5. Start the server

```
somdsrv -a CARSERVER1
```

See *OS/390 SOMobjects Configuration and Administration Guide* for more information.

6. Start the client

```
drvcard 4
```

## Building and Running a Distributed Application from the TSO Environment

The following example demonstrates a distributed SOM example of “Vehicle” running in the TSO environment with REXX execs.

**Note:** Have the SOM subsystem up. See *OS/390 SOMobjects Configuration and Administration Guide* for more information.

The datasets where the IDLs can be found are:

- *sommvs.SGOSSMPI.IDL(DCAR)*.
- *sommvs.SGOSSMPI.IDL(DVEHICLE)*.

The datasets where the updated C implementation templates can be found are:

- *sommvs.SGOSSMPC.C(DCAR)*
- *sommvs.SGOSSMPC.C(DVEHICLE)*

The dataset where the client code can be found is  
*sommvs.SGOSSMPC.C(DRVCARD)*

The datasets where the REXX execs can be found are:

- *sommvs.SGOSEEXEC(GOSSETUP)*
- *sommvs.SGOSEEXEC(GOSRDVEH)*
- *sommvs.SGOSEEXEC(GOSRDCAR)*
- *sommvs.SGOSEEXEC(GOSRDRVC)*
- *sommvs.SGOSEEXEC(GOSRDAUT)*

The following is the order to run the REXX execs for the distributed Vehicle example:

1. From the ready message, run the following to set up your TSO environment:

```
ex 'SOMMVS.SGOSEEXEC(GOSSETUP)'
```

2. From the ready message, run the following to SOM compile, C compile, prelink, and link to create the DVEHICLE DLL.

```
ex 'SOMMVS.SGOSEEXEC(GOSRDVEH)'
```

3. From the ready message, run the following to SOM compile, C compile, prelink, and link to create the DCAR DLL.

```
ex 'SOMMVS.SGOSEXEC(GOSRDCAR)'
```

4. Update the Implementation Repository with your server and classes.

See the *OS/390 SOMobjects Configuration and Administration Guide* for more information on updating the Implementation Repository.

5. From the ready message, run the following to C compile, prelink, and link to create the client DLL as DRVCARD.

```
ex 'SOMMVS.SGOSEXEC(GOSRDRVC)'
```

6. From the ready message, run the client.

```
ex 'SOMMVS.SGOSEXEC(GOSRDAUT)'
```

---

## Summarized Distributed Examples

The following sections — discuss the distributed “Stack”, “SStack”, “Vehicle”, “Animals” and “Phone” examples which can be found in the SOMobjects product.

The datasets for the following examples include the files necessary to create distributed server objects for these examples. The IDL, .h, .c, and object files can be found under the PDS qualifier *sommvs.SGOSSMP\** in the datasets you received. Also, sample JCL may be found in *sommvs.SGOSJCL*.

The rest of this section points out the steps and datasets needed to run the following examples:

- “Stack”
- “SStack” on page 9-9
- “Vehicle” on page 9-11
- “Animals” on page 9-14
- “Phone” on page 9-18

## Stack

Table 9-1 shows the steps and default data set names needed to run “Stack” as a distributed SOMobjects (DSOM) client/server application.

**Note:** The column labeled “Default Data Set Names” are the data set names as shipped with the product. *sommvs* is the high level qualifier we refer to in these default data sets. Your system administrator may have modified the high level qualifier or the entire names of these default data sets on your system.

---

Table 9-1 (Page 1 of 4). Building and Running “Stack” in a Distributed Environment

| Step                                          | Action | Default Data Set Names |
|-----------------------------------------------|--------|------------------------|
| Building “Stack” in a Distributed Environment |        |                        |

Table 9-1 (Page 2 of 4). Building and Running "Stack" in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | Default Data Set Names                                               |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| 1.   | <p>Ensure that your SOMobjects environment has been configured.</p> <ul style="list-style-type: none"> <li>— Register the "Stack" server with WLM.</li> <li>— Register the "Stack" server with the Security server.</li> </ul> <p><b>Note:</b> Consult your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | Not applicable                                                       |
| 2.   | <p>SOM compile the IDL.</p> <ul style="list-style-type: none"> <li>— Copy the IDL (<i>sommvs.SGOSSMPI.IDL(STACK)</i>) into your own private data set.</li> <li>— Copy the SOM compile JCL from <i>sommvs.SGOSJCL(GOSSMPSC)</i> into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPSC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPSC)</i>).</li> </ul> <p><b>Notes:</b></p> <ol style="list-style-type: none"> <li>1. The SAMPLIB symbolic points to the data set group that will contain the sample headers being built. These data sets must already exist before this JCL can be successful.</li> <li>2. The SOMLIB symbolic points to the data set group that contain the IBM supplied system load modules.</li> <li>3. The SOMENV DD statement points to a data set that has the SMINCLUDE file definitions in it.</li> <li>4. Set STACK, SSTACK, ANIMAL and AUTO to either           <ul style="list-style-type: none"> <li>1 - you want to perform the som compile for the sample</li> <li>0 - you do NOT want to perform the som compile for the sample</li> </ul> </li> </ol> <p>Make the following update to your copy of <i>sommvs.SGOSJCL(GOSSMPSC)</i> to SOM compile only the "Stack" IDL:</p> <pre>// SET STACK=1 // SET SSTACK=0 // SET ANIMAL=0 // SET AUTO=0</pre> <p>Note that this sample JCL contains the SOM compile for all the distributed sample IDLs other than "Phone".</p> <p>Submit <i>sommvs.SGOSJCL(GOSSMPSC)</i>.</p> | <i>sommvs.SGOSSMPI.IDL(STACK)</i><br><i>sommvs.SGOSJCL(GOSSMPSC)</i> |

Table 9-1 (Page 3 of 4). Building and Running “Stack” in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Default Data Set Names                                                                                 |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| 3.   | <p>Update the Interface Repository (IR).</p> <ul style="list-style-type: none"> <li>— Copy the JCL (found in <i>sommvs.SGOSJCL(GOSSMPIR)</i>) for generating the IR for “Stack” into your own private data set.</li> </ul> <p><b>Note:</b> The IR has already been generated for your use and is available in <i>sommvs.SGOSIRSM</i>. This sample JCL also contains extra steps beyond the needs of this “Stack” sample. You need to use only the “Stack” step for this sample.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | <i>sommvs.SGOSJCL(GOSSMPIR)</i><br><i>sommvs.SGOSIRSM</i>                                              |
| 4.   | <p>C Compile.</p> <ul style="list-style-type: none"> <li>— If you intend to modify the C code, copy the C compile JCL (<i>sommvs.SGOSJCL(GOSSMPCC)</i>) and the C implementation templates (<i>sommvs.SGOSSMPC.C(STACK)</i> and <i>sommvs.SGOSSMPC.C(CLIENT)</i>) into your own private data sets.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPCC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPCC)</i>).</li> </ul> <p><b>Note:</b> Normally when creating a new application, you would modify the generated C template before compiling the program. For this sample program, however, we have done this modification for you so that you can now compile the program. This program is actually two programs, one for the “Stack” server (found in your copy of <i>sommvs.SGOSSMPC.C(STACK)</i>), and one for the client (“client”) (found in your copy of <i>sommvs.SGOSSMPC.C(CLIENT)</i>). The “client” program runs the “Stack” example.</p> <ul style="list-style-type: none"> <li>— Submit your modified C compile JCL after making the following update to your copy of <i>sommvs.SGOSJCL(GOSSMPCC)</i> to C compile only “Stack”:</li> </ul> <pre>// SET STACK=1 // SET SSTACK=0 // SET ANIMAL=0 // SET AUTO=0</pre> | <i>sommvs.SGOSJCL(GOSSMPCC)</i><br><i>sommvs.SGOSSMPC.C(STACK)</i><br><i>sommvs.SGOSSMPC.C(CLIENT)</i> |

Table 9-1 (Page 4 of 4). Building and Running "Stack" in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Default Data Set Names          |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 5.   | <p>Prelink/Linkedit.</p> <ul style="list-style-type: none"> <li>— Copy the prelink/link JCL (<i>sommvs.SGOSJCL(GOSSMPLK)</i>) into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPLK)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPLK)</i>).</li> <li>— Submit your copy of <i>sommvs.SGOSJCL(GOSSMPLK)</i> to linkedit the executable code into your load library. If you want to linkedit just the "Stack" and "client" programs, modify the JCL variables as follows:</li> </ul> <pre>// SET STACK=1 // SET SSTACK=0 // SET ANIMAL=0 // SET AUTO=0</pre> | <i>sommvs.SGOSJCL(GOSSMPLK)</i> |
| 6.   | <p>Start the SOM subsystem.</p> <ul style="list-style-type: none"> <li>— A sample SOM subsystem PROC is shipped in <i>sommvs.SGOSJCL(GOS1DSOM)</i>. Contact your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                               | <i>sommvs.SGOSJCL(GOS1DSOM)</i> |
| 7.   | <p>Start the server (optional).</p> <ul style="list-style-type: none"> <li>— You may start the server manually or you can let the server start automatically as the client requests processing for that server. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> for a description of starting a DSOM server.</li> </ul>                                                                                                                                                                                                                                                                                                                                     | Not applicable                  |

#### Running "Stack" as a Client in a Distributed Environment

|    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                 |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 1. | <p>Start the client.</p> <ul style="list-style-type: none"> <li>— Copy the client program (<i>sommvs.SGOSJCL(GOSXQCLI)</i>) into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSXQCLI)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSXQCLI)</i>).</li> <li>— Submit your modified JCL to start the client.</li> </ul> <p><b>Note:</b> This client program will run the "Stack" sample program and will send requests to and receive data back from the server. As with the server JCL, you should modify environment variables by changing the SOMENV profile DD specification or by changing the contents of the SOMENV profile data set. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> and Appendix A, "Setting up Configuration Files" on page A-1 for more information on environment variables.</p> | <i>sommvs.SGOSJCL(GOSXQCLI)</i> |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|

## SStack

Table 9-2 shows the steps and default data set names needed to run “SStack” as a distributed SOMobjects (DSOM) client/server application.

**Note:** Your system administrator may have modified the high level qualifier or the entire names of these default data sets on your system.

Table 9-2 (Page 1 of 3). Building and Running “SStack” in a Distributed Environment

| Step                                                  | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | Default Data Set Names                                               |
|-------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| <b>Building “SStack” in a Distributed Environment</b> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                      |
| 1.                                                    | <p>Ensure that your SOMobjects environment has been configured.</p> <ul style="list-style-type: none"> <li>— Register the “SStack” server with WLM.</li> <li>— Register the “SStack” server with the Security server.</li> </ul> <p><b>Note:</b> Consult your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Not applicable                                                       |
| 2.                                                    | <p>SOM compile the IDL.</p> <ul style="list-style-type: none"> <li>— Copy the IDL (<i>sommvs.SGOSSMPI.IDL(STACK)</i>) into your own private data set.</li> <li>— Copy the SOM compile JCL from <i>sommvs.SGOSJCL(GOSSMPSC)</i> into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPSC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPSC)</i>).</li> <li>— Make the following update to your copy of <i>sommvs.SGOSJCL(GOSSMPSC)</i> to SOM compile only the “SStack” IDL:</li> <pre>// SET STACK=0 // SET SSTACK=1 // SET ANIMAL=0 // SET AUTO=0</pre> <p>Note that this sample JCL contains the SOM compile for all the distributed sample IDLs other than “Phone”.</p> <li>— Submit <i>sommvs.SGOSJCL(GOSSMPSC)</i>.</li> </ul> | <i>sommvs.SGOSSMPI.IDL(STACK)</i><br><i>sommvs.SGOSJCL(GOSSMPSC)</i> |
| 3.                                                    | <p>Update the Interface Repository (IR).</p> <ul style="list-style-type: none"> <li>— Copy the JCL (found in <i>sommvs.SGOSJCL(GOSSMPIR)</i>) for generating the IR for “SStack” into your own private data set.</li> </ul> <p><b>Note:</b> The IR has already been generated for your use and is available in <i>sommvs.SGOSIRSM</i>. This sample JCL also contains extra steps beyond the needs of this “SStack” sample. You need to use only the “SStack” step for this sample.</p>                                                                                                                                                                                                                                                                                                                                                               | <i>sommvs.SGOSJCL(GOSSMPIR)</i><br><i>sommvs.SGOSIRSM</i>            |

Table 9-2 (Page 2 of 3). Building and Running "SStack" in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Default Data Set Names                                                                                  |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| 4.   | <p>C Compile.</p> <ul style="list-style-type: none"> <li>— If you intend to modify the C code, copy the C compile JCL (<i>sommvs.SGOSJCL(GOSSMPCC)</i>) and the C implementation templates (<i>sommvs.SGOSSMPC.C(SSTACK)</i> and <i>sommvs.SGOSSMPC.C(SCIENT)</i>) into your own private data sets.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPCC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPCC)</i>).</li> <li><b>Note:</b> Normally when creating a new application, you would modify the generated C template before compiling the program. For this sample program, however, we have done this modification for you so that you can now compile the program. This program is actually two programs, one for the "SStack" server (found in your copy of <i>sommvs.SGOSSMPC.C(SSTACK)</i>), and one for the client (found in your copy of <i>sommvs.SGOSSMPC.C(SCIENT)</i>). The "client" program runs the "SStack" example.</li> <li>— Submit your modified C compile JCL after making the following update to your copy of <i>sommvs.SGOSJCL(GOSSMPCC)</i> to C compile only "SStack":</li> </ul> <pre>// SET STACK=0 // SET SSTACK=1 // SET ANIMAL=0 // SET AUTO=0</pre> | <i>sommvs.SGOSJCL(GOSSMPCC)</i><br><i>sommvs.SGOSSMPC.C(SSTACK)</i><br><i>sommvs.SGOSSMPC.C(SCIENT)</i> |
| 5.   | <p>Prelink/Linkedit.</p> <ul style="list-style-type: none"> <li>— Copy the prelink/link JCL (<i>sommvs.SGOSJCL(GOSSMPLK)</i>) into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPLK)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPLK)</i>).</li> <li>— Submit your copy of <i>sommvs.SGOSJCL(GOSSMPLK)</i> to linkedit the executable code into your load library. If you want to linkedit just the "SStack" and "scient" programs, modify the JCL variables as follows:</li> </ul> <pre>// SET STACK=0 // SET SSTACK=1 // SET ANIMAL=0 // SET AUTO=0</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | <i>sommvs.SGOSJCL(GOSSMPLK)</i>                                                                         |

Table 9-2 (Page 3 of 3). Building and Running “SStack” in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                      | Default Data Set Names          |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 6.   | <p>Start the SOM subsystem.</p> <ul style="list-style-type: none"> <li>— A sample SOM subsystem PROC is shipped in <i>sommvs.SGOSJCL(GOS1DSOM)</i>. Contact your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</li> </ul>                                           | <i>sommvs.SGOSJCL(GOS1DSOM)</i> |
| 7.   | <p>Start the server (optional).</p> <ul style="list-style-type: none"> <li>— You may start the server manually or you can let the server start automatically as the client requests processing for that server. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> for a description of starting a DSOM server.</li> </ul> | Not applicable                  |

#### Running “SStack” as a Client in a Distributed Environment

|    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                 |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 1. | <p>Start the client.</p> <ul style="list-style-type: none"> <li>— Copy the client program (<i>sommvs.SGOSJCL(GOSXQSCL)</i>) into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSXQSCL)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSXQSCL)</i>).</li> <li>— Submit your modified JCL to start the client.</li> </ul> <p><b>Note:</b> This client program will run the “SStack” sample program and will send requests to and receive data back from the server. As with the server JCL, you should modify environment variables by changing the SOMENV profile DD specification or by changing the contents of the SOMENV profile data set. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> and Appendix A, “Setting up Configuration Files” on page A-1 for more information on environment variables.</p> | <i>sommvs.SGOSJCL(GOSXQSCL)</i> |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|

## Vehicle

Table 9-3 shows the steps and default data set names needed to run “Vehicle” as a distributed SOMobjects (DSOM) client/server application.

**Note:** Your system administrator may have modified the high level qualifier or the entire names of these default data sets on your system.

Table 9-3 (Page 1 of 4). Building and Running “Vehicle” in a Distributed Environment

| Step                                                   | Action | Default Data Set Names |
|--------------------------------------------------------|--------|------------------------|
| <b>Building “Vehicle” in a Distributed Environment</b> |        |                        |

Table 9-3 (Page 2 of 4). Building and Running "Vehicle" in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | Default Data Set Names                                                                                      |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| 1.   | <p>Ensure that your SOMobjects environment has been configured.</p> <ul style="list-style-type: none"> <li>— Register the "Vehicle" server with WLM.</li> <li>— Register the "Vehicle" server with the Security server.</li> </ul> <p><b>Note:</b> Consult your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | Not applicable                                                                                              |
| 2.   | <p>SOM compile the IDL.</p> <ul style="list-style-type: none"> <li>— Copy the IDLs (<i>sommvs.SGOSSMPI.IDL(DVEHICLE)</i>) and (<i>sommvs.SGOSSMPI.IDL(DCAR)</i>) into your own private data sets.</li> <li>— Copy the SOM compile JCL from <i>sommvs.SGOSJCL(GOSSMPSC)</i> into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPSC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPSC)</i>).</li> <li>— Make the following update to your copy of <i>sommvs.SGOSJCL(GOSSMPSC)</i> to SOM compile only the "Vehicle" IDL:</li> <pre>// SET STACK=0 // SET SSTACK=0 // SET ANIMAL=0 // SET AUTO=1</pre> <p>Note that this sample JCL contains the SOM compile for all the distributed sample IDLs other than "Phone".</p> <li>— Submit <i>sommvs.SGOSJCL(GOSSMPSC)</i>.</li> </ul> | <i>sommvs.SGOSSMPI.IDL(DVEHICLE)</i><br><i>sommvs.SGOSSMPI.IDL(DCAR)</i><br><i>sommvs.SGOSJCL(GOSSMPSC)</i> |
| 3.   | <p>Update the Interface Repository (IR).</p> <ul style="list-style-type: none"> <li>— Copy the JCL (found in <i>sommvs.SGOSJCL(GOSSMPIR)</i>) for generating the IR for "Vehicle" into your own private data set.</li> </ul> <p><b>Note:</b> The IR has already been generated for your use and is available in <i>sommvs.SGOSIRSM</i>. This sample JCL also contains extra steps beyond the needs of this "Vehicle" sample. You need to use only the "Vehicle" step for this sample.</p>                                                                                                                                                                                                                                                                                                                                                                                                         | <i>sommvs.SGOSJCL(GOSSMPIR)</i><br><i>sommvs.SGOSIRSM</i>                                                   |

Table 9-3 (Page 3 of 4). Building and Running “Vehicle” in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | Default Data Set Names                                                                                                              |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| 4.   | <p>C Compile.</p> <ul style="list-style-type: none"> <li>— If you intend to modify the C code, copy the C compile JCL (<i>sommvs.SGOSJCL(GOSSMPCC)</i>) and the C implementation templates (<i>sommvs.SGOSSMPC.C(DVEHICLE)</i>, <i>sommvs.SGOSSMPC.C(DCAR)</i> and <i>sommvs.SGOSSMPC.C(DRVCARD)</i>) into your own private data sets.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPCC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPCC)</i>).</li> <li>— <b>Note:</b> Normally when creating a new application, you would modify the generated C template before compiling the program. For this sample program, however, we have done this modification for you so that you can now compile the program. This program is actually three programs, one for the “Vehicle” server, (found in your copy of <i>sommvs.SGOSSMPC.C(DVEHICLE)</i>), one for the “Car” server (found in your copy of <i>sommvs.SGOSSMPC.C(DCAR)</i>), and one for the “client” (found in your copy of <i>sommvs.SGOSSMPC.C(DRVCARD)</i>). The “client” program runs the “Vehicle” example.</li> <li>— Submit your modified C compile JCL after making the following update to your copy of <i>sommvs.SGOSJCL(GOSSMPCC)</i> to C compile only “Vehicle”:</li> </ul> <pre>// SET STACK=0 // SET SSTACK=0 // SET ANIMAL=0 // SET AUTO=1</pre> | <i>sommvs.SGOSJCL(GOSSMPCC)</i> <i>sommvs.SGOSSMPC.C(DVEHICLE)</i> <i>sommvs.SGOSSMPC.C(DCAR)</i> <i>sommvs.SGOSSMPC.C(DRVCARD)</i> |
| 5.   | <p>Prelink/Linkedit.</p> <ul style="list-style-type: none"> <li>— Copy the prelink/link JCL (<i>sommvs.SGOSJCL(GOSSMPLK)</i>) into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPLK)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPLK)</i>).</li> <li>— Submit your copy of <i>sommvs.SGOSJCL(GOSSMPLK)</i> to linkedit the executable code into your load library. If you want to linkedit just the “Vehicle” and “client” programs, modify the JCL variables as follows:</li> </ul> <pre>// SET STACK=0 // SET SSTACK=0 // SET ANIMAL=0 // SET AUTO=1</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | <i>sommvs.SGOSJCL(GOSSMPLK)</i>                                                                                                     |

Table 9-3 (Page 4 of 4). Building and Running “Vehicle” in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                      | Default Data Set Names          |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 6.   | <p>Start the SOM subsystem.</p> <ul style="list-style-type: none"> <li>— A sample SOM subsystem PROC is shipped in <i>sommvs.SGOSJCL(GOS1DSOM)</i>. Contact your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</li> </ul>                                           | <i>sommvs.SGOSJCL(GOS1DSOM)</i> |
| 7.   | <p>Start the server (optional).</p> <ul style="list-style-type: none"> <li>— You may start the server manually or you can let the server start automatically as the client requests processing for that server. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> for a description of starting a DSOM server.</li> </ul> | Not applicable                  |

#### Running “Vehicle” as a Client in a Distributed Environment

|    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                 |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 1. | <p>Start the client.</p> <ul style="list-style-type: none"> <li>— Copy the client program (<i>sommvs.SGOSJCL(GOSXQSCL)</i>) into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSXQSCL)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSXQSCL)</i>).</li> <li>— Edit the program name in <i>sommvs.SGOSJCL(GOSXQSCL)</i> to start the DRVCARD program. For example:</li> <pre>//DRVCARD EXEC PGM=DRVCARD</pre> <li>— Submit your modified JCL to start the client.</li> </ul> <p><b>Note:</b> This client program will run the “Vehicle” sample program and will send requests to and receive data back from the server. As with the server JCL, you should modify environment variables by changing the SOMENV profile DD specification or by changing the contents of the SOMENV profile data set. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> and Appendix A, “Setting up Configuration Files” on page A-1 for more information on environment variables. See <i>OS/390 SOMobjects: Getting Started</i> for a detailed distributed example of “Vehicle”.</p> | <i>sommvs.SGOSJCL(GOSXQSCL)</i> |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|

## Animals

Table 9-4 on page 9-15 shows the steps and default data set names needed to run “Animals” as a distributed SOMobjects (DSOM) client/server application.

**Note:** Your system administrator may have modified the high level qualifier or the entire names of these default data sets on your system.

Table 9-4 (Page 1 of 4). Building and Running "Animals" in a Distributed Environment

| Step                                                   | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | Default Data Set Names                                                                                                                                                                                                   |
|--------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Building "Animals" in a Distributed Environment</b> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                                                                                                                                                                                                                          |
| 1.                                                     | <p>Ensure that your SOMobjects environment has been configured.</p> <ul style="list-style-type: none"> <li>— Register the "Animals" server with WLM.</li> <li>— Register the "Animals" server with the Security server.</li> </ul> <p><b>Note:</b> Consult your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Not applicable                                                                                                                                                                                                           |
| 2.                                                     | <p>SOM compile the IDLs.</p> <ul style="list-style-type: none"> <li>— Copy the IDLs (<i>sommvs.SGOSSMPI.IDL(DANIMAL)</i>, <i>sommvs.SGOSSMPI.IDL(DBDOG)</i>, <i>sommvs.SGOSSMPI.IDL(DDOG)</i>, <i>sommvs.SGOSSMPI.IDL(DLDOG)</i> and <i>sommvs.SGOSSMPI.IDL(DLOC)</i>) into your own private data sets.</li> <li>— Copy the SOM compile JCL from <i>sommvs.SGOSJCL(GOSSMPSC)</i> into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPSC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPSC)</i>).</li> <li>— Make the following update to your copy of <i>sommvs.SGOSJCL(GOSSMPSC)</i> to SOM compile only the "Animals" IDL:</li> </ul> <pre>// SET STACK=0 // SET SSTACK=0 // SET ANIMAL=1 // SET AUTO=0</pre> <p>Note that this sample JCL contains the SOM compile for all the distributed sample IDLs other than "Phone".</p> <ul style="list-style-type: none"> <li>— Submit <i>sommvs.SGOSJCL(GOSSMPSC)</i>.</li> </ul> | <i>sommvs.SGOSSMPI.IDL(DANIMAL)</i><br><i>sommvs.SGOSSMPI.IDL(DBDOG)</i><br><i>sommvs.SGOSSMPI.IDL(DDOG)</i><br><i>sommvs.SGOSSMPI.IDL(DLDOG)</i><br><i>sommvs.SGOSSMPI.IDL(DLOC)</i><br><i>sommvs.SGOSJCL(GOSSMPSC)</i> |
| 3.                                                     | <p>Update the Interface Repository (IR).</p> <ul style="list-style-type: none"> <li>— Copy the JCL (found in <i>sommvs.SGOSJCL(GOSSMPIR)</i>) for generating the IR for "Animals" into your own private data set.</li> </ul> <p><b>Note:</b> The IR has already been generated for your use and is available in <i>sommvs.SGOSIRSM</i>. This sample JCL also contains extra steps beyond the needs of this "Animals" sample. You need to use only the "Animals" step for this sample.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | <i>sommvs.SGOSJCL(GOSSMPIR)</i><br><i>sommvs.SGOSIRSM</i>                                                                                                                                                                |

Table 9-4 (Page 2 of 4). Building and Running "Animals" in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | Default Data Set Names                                                                                                                                                                                                                                                                   |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4.   | <p>C Compile.</p> <ul style="list-style-type: none"> <li>— If you intend to modify the C code, copy the C compile JCL (<i>sommvs.SGOSJCL(GOSSMPCC)</i>) and the C implementation templates (<i>sommvs.SGOSSMPC.C(ANITEST)</i>, <i>sommvs.SGOSSMPC.C(DANIMAL)</i>, <i>sommvs.SGOSSMPC.C(DBDOG)</i>, <i>sommvs.SGOSSMPC.C(DDOG)</i>, <i>sommvs.SGOSSMPC.C(DLDOG)</i>, <i>sommvs.SGOSSMPC.C(DLOC)</i> and <i>sommvs.SGOSSMPC.C(TESTSVR)</i>) into your own private data sets.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPCC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPCC)</i>).</li> <li><b>Note:</b> Normally when creating a new application, you would modify the generated C template before compiling the program. For this sample program, however, we have done this modification for you so that you can now compile the program. This program is actually two programs. One for the "Animals" server, found in your copy of <i>sommvs.SGOSSMPC.C(TESTSVR)</i>, and one for the "Animals" client, found in your copy of <i>sommvs.SGOSSMPC.C(ANITEST)</i>.</li> <li>— Submit your modified C compile JCL after making the following update to your copy of <i>sommvs.SGOSJCL(GOSSMPCC)</i> to C compile only "Animals":</li> </ul> <pre>// SET STACK=0 // SET SSTACK=0 // SET ANIMAL=1 // SET AUTO=0</pre> | <i>sommvs.SGOSJCL(GOSSMPCC)</i><br><i>sommvs.SGOSSMPC.C(ANITEST)</i><br><i>sommvs.SGOSSMPC.C(DANIMAL)</i><br><i>sommvs.SGOSSMPC.C(DBDOG)</i><br><i>sommvs.SGOSSMPC.C(DDOG)</i><br><i>sommvs.SGOSSMPC.C(DLDOG)</i><br><i>sommvs.SGOSSMPC.C(DLOC)</i><br><i>sommvs.SGOSSMPC.C(TESTSVR)</i> |

Table 9-4 (Page 3 of 4). Building and Running “Animals” in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | Default Data Set Names          |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 5.   | <p>Prelink/Linkedit.</p> <ul style="list-style-type: none"> <li>— Copy the prelink/link JCL (<i>sommvs.SGOSJCL(GOSSMPLK)</i>) into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMPLK)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMPLK)</i>).</li> <li>— Submit your copy of <i>sommvs.SGOSJCL(GOSSMPLK)</i> to linkedit the executable code into your load library.</li> </ul> <p><b>Notes:</b></p> <ol style="list-style-type: none"> <li>1. The SAMPLIB symbolic points to the data set group that will contain the sample headers being built. These data sets must already exist before this JCL can be successful.</li> <li>2. The SOMLIB symbolic points to the data set group that contain the IBM supplied system load modules.</li> <li>3. The SOMENV DD statement points to a data set that has the SMINCLUDE file definitions in it.</li> <li>4. Set STACK, SSTACK, ANIMAL and AUTO to either           <ul style="list-style-type: none"> <li>1 - you want to perform the som compile for the sample</li> <li>0 - you do NOT want to perform the som compile for the sample</li> </ul> </li> </ol> <p>If you want to linkedit just the “Animals” programs, modify the JCL variables as follows:</p> <pre>// SET STACK=0 // SET SSTACK=0 // SET ANIMAL=1 // SET AUTO=0</pre> | <i>sommvs.SGOSJCL(GOSSMPLK)</i> |
| 6.   | <p>Start the SOM subsystem.</p> <ul style="list-style-type: none"> <li>— A sample SOM subsystem PROC is shipped in <i>sommvs.SGOSJCL(GOS1DSOM)</i>. Contact your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | <i>sommvs.SGOSJCL(GOS1DSOM)</i> |
| 7.   | <p>Start the server (optional).</p> <ul style="list-style-type: none"> <li>— You may start the server manually or you can let the server start automatically as the client requests processing for that server. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> for a description of starting a DSOM server.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | Not applicable                  |

#### Running “Animals” as a Client in a Distributed Environment

Table 9-4 (Page 4 of 4). Building and Running “Animals” in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | Default Data Set Names          |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 1.   | <p>Start the client.</p> <ul style="list-style-type: none"> <li>— Copy the client program (<i>sommvs.SGOSJCL(GOSXQANI)</i>) into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSXQANI)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSXQANI)</i>).</li> <li>— Submit your modified JCL to start the client.</li> </ul> <p><b>Note:</b> This client program will run the “Animals” sample program and will send requests to and receive data back from the server. As with the server JCL, you should modify environment variables by changing the SOMENV profile DD specification or by changing the contents of the SOMENV profile data set. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> and Appendix A, “Setting up Configuration Files” on page A-1 for more information on environment variables. See <i>OS/390 SOMobjects: Getting Started</i> for a detailed distributed example of “Vehicle”.</p> | <i>sommvs.SGOSJCL(GOSXQANI)</i> |

## Phone

Table 9-5 shows the steps and default data set names needed to run “Phone” as a distributed SOMobjects (DSOM) client/server application.

**Note:** Your system administrator may have modified the high level qualifier or the entire names of these default data sets on your system.

This sample is written in C++ and is discussed in more detail in the “Persistent Object Service” chapter in *OS/390 SOMobjects Object Services*.

Table 9-5 (Page 1 of 5). Building and Running “Phone” in a Distributed Environment

| Step                                                 | Action                                                                                                                                                                                                                                                                                                                                                                                | Default Data Set Names |
|------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|
| <b>Building “Phone” in a Distributed Environment</b> |                                                                                                                                                                                                                                                                                                                                                                                       |                        |
| 1.                                                   | <p>Ensure that your SOMobjects environment has been configured.</p> <ul style="list-style-type: none"> <li>— Register the “Phone” server with WLM.</li> <li>— Register the “Phone” server with the Security server.</li> </ul> <p><b>Note:</b> Consult your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</p> | Not applicable         |

Table 9-5 (Page 2 of 5). Building and Running "Phone" in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | Default Data Set Names                                               |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| 2.   | <p>SOM compile the IDL.</p> <ul style="list-style-type: none"> <li>— Copy the IDL (<i>sommvs.SGOSSMPI.IDL(PHONE)</i>) into your own private data set.</li> <li>— Copy the SOM compile JCL from <i>sommvs.SGOSJCL(GOSSMXSC)</i> into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMXSC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMXSC)</i>).</li> <li>— Make the following update to your copy of <i>sommvs.SGOSJCL(GOSSMXSC)</i> to SOM compile only the "SStack" IDL:</li> <pre>// SET STACK=0 // SET SSTACK=0 // SET ANIMAL=0 // SET PHONE=1</pre> <li>— Submit <i>sommvs.SGOSJCL(GOSSMXSC)</i>.</li> </ul> | <i>sommvs.SGOSSMPI.IDL(PHONE)</i><br><i>sommvs.SGOSJCL(GOSSMXSC)</i> |
| 3.   | <p>Update the Interface Repository (IR).</p> <ul style="list-style-type: none"> <li>— Copy the JCL (found in <i>sommvs.SGOSJCL(GOSSMPIR)</i>) for generating the IR for "Phone" into your own private data set.</li> </ul> <p><b>Note:</b> The IR has already been generated for your use and is available in <i>sommvs.SGOSIRSM</i>. This sample JCL also contains extra steps beyond the needs of this "Phone" sample. You need to use only the "Phone" step for this sample.</p>                                                                                                                                                                                                                                                 | <i>sommvs.SGOSJCL(GOSSMPIR)</i><br><i>sommvs.SGOSIRSM</i>            |

Table 9-5 (Page 3 of 5). Building and Running "Phone" in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | Default Data Set Names                                                                                       |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| 4.   | <p>C++ Compile.</p> <ul style="list-style-type: none"> <li>— If you intend to modify the C++ code, copy the C++ compile JCL (<i>sommvs.SGOSJCL(GOSSMXCC)</i>) and the C++ implementation templates (<i>sommvs.SGOSSMPX.CXX(PHONE)</i> and <i>sommvs.SGOSSMPX.CXX(PHONMAIN)</i>) into your own private data sets.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMXCC)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMXCC)</i>).</li> <li><b>Note:</b> Normally when creating a new application, you would modify the generated C template before compiling the program. For this sample program, however, we have done this modification for you so that you can now compile the program. This program is actually two programs, one for the "Phone" server (found in your copy of <i>sommvs.SGOSSMPX.CXX(PHONE)</i>), and one for the client (found in your copy of <i>sommvs.SGOSSMPX.CXX(PHONMAIN)</i>). The "client" program runs the "Phone" example.</li> <li>— Submit your copy of <i>sommvs.SGOSJCL(GOSSMXCC)</i> to C++ compile "Phone":</li> </ul> | <i>sommvs.SGOSJCL(GOSSMXCC)</i><br><i>sommvs.SGOSSMPX.CXX(PHONE)</i><br><i>sommvs.SGOSSMPX.CXX(PHONMAIN)</i> |

Table 9-5 (Page 4 of 5). Building and Running “Phone” in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Default Data Set Names          |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 5.   | <p>Prelink/Linkedit.</p> <ul style="list-style-type: none"> <li>— Copy the prelink/link JCL (<i>sommvs.SGOSJCL(GOSSMXLK)</i>) into your own private data set.</li> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSSMXLK)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSSMXLK)</i>).</li> <li>— Submit your copy of <i>sommvs.SGOSJCL(GOSSMXLK)</i> to linkedit the executable code into your load library.</li> </ul> <p><b>Notes:</b></p> <ol style="list-style-type: none"> <li>1. The SAMPLIB symbolic points to the data set group that will contain the sample headers being built. These data sets must already exist before this JCL can be successful.</li> <li>2. The SOMLIB symbolic points to the data set group that contain the IBM supplied system load modules.</li> <li>3. The SOMENV DD statement points to a data set that has the SMINCLUDE file definitions in it.</li> <li>4. Set STACK, SSTACK, ANIMAL and PHONE to either           <ul style="list-style-type: none"> <li>1 - you want to perform the som compile for the sample</li> <li>0 - you do NOT want to perform the som compile for the sample</li> </ul> </li> </ol> <p>If you want to linkedit just the “Phone” program, modify the JCL variables as follows:</p> <pre>// SET STACK=0 // SET SSTACK=0 // SET ANIMAL=0 // SET PHONE=1</pre> | <i>sommvs.SGOSJCL(GOSSMXLK)</i> |
| 6.   | <p>Start the SOM subsystem.</p> <ul style="list-style-type: none"> <li>— A sample SOM subsystem PROC is shipped in <i>sommvs.SGOSJCL(GOS1DSOM)</i>. Contact your system administrator or see the <i>OS/390 SOMobjects Configuration and Administration Guide</i> for more information.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | <i>sommvs.SGOSJCL(GOS1DSOM)</i> |
| 7.   | <p>Start the server (optional).</p> <ul style="list-style-type: none"> <li>— You may start the server manually or you can let the server start automatically as the client requests processing for that server. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> for a description of starting a DSOM server.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | Not applicable                  |

#### Running “Phone” as a Client in a Distributed Environment

Table 9-5 (Page 5 of 5). Building and Running "Phone" in a Distributed Environment

| Step | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | Default Data Set Names          |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| 1.   | <p>Start the client.</p> <ul style="list-style-type: none"> <li>— Copy the client program (<i>sommvs.SGOSJCL(GOSXQSCL)</i>) into your own private data set and edit the program name to start the PHONMAIN program. For example:</li> <pre>//PHONMAIN EXEC PGM=PHONMAIN</pre> <li>— Modify the job card in your copy of <i>sommvs.SGOSJCL(GOSXQSCL)</i> and any other parts of the JCL needed for your environment (refer to the notes in <i>sommvs.SGOSJCL(GOSXQSCL)</i>).</li> <li>— Submit your modified JCL to start the client.</li> </ul> <p><b>Note:</b> This client program will run the "Phone" sample program and will send requests to and receive data back from the server. As with the server JCL, you should modify environment variables by changing the SOMENV profile DD specification or by changing the contents of the SOMENV profile data set. See <i>OS/390 SOMobjects Configuration and Administration Guide</i> and Appendix A, "Setting up Configuration Files" on page A-1 for more information on environment variables.</p> | <i>sommvs.SGOSJCL(GOSXQSCL)</i> |

You have now been able to build and access both non-distributed and distributed classes in various environments in OS/390.

The next chapter will demonstrate the language neutrality of SOMobjects by using examples in C, C++ and COBOL.

---

## Chapter 10. Language Neutrality with SOMobjects: Examples

This chapter demonstrates the language neutrality of SOMobjects by using examples in C, C++ and COBOL.

Object technology on OS/390 includes support for IBM C, C++, and COBOL programming languages. Using the C++ Direct-to-SOM (DTS) feature, a C++ programmer is able to generate SOM objects directly from C++ programs, without hand-coding IDL. Object-oriented language extensions to COBOL on OS/390 are also available, and provide the ability to generate objects directly from COBOL programs, again without a requirement to hand-code SOM IDL.

One of the benefits of SOMobjects is its programming language neutrality. SOMobjects preserves the key OO technology characteristics of encapsulation, inheritance, and polymorphism without requiring that the class implementer and the class user work in the same programming language.

Examples of this language neutrality are:

- Creating a C++ class (using Direct-to-SOM (DTS)) that is used by a C client
- A class written in COBOL inheriting from and extending a class written in C++
- Client programs written in the C++ and COBOL languages invoking methods in both the C++ and COBOL classes.

**Note:** Before you can begin running any of these examples, the SOMobjects environment needs to be configured on your system. Consult with your system administrator to ensure that the SOMobjects environment has been configured. For more information on how to configure the SOMobjects environment, see the *OS/390 SOMobjects Configuration and Administration Guide*.

|  
| **Note:** Some of these examples show that **somInit** and **somUninit** are overridden.  
| These methods now execute under the overall control of the **somDefaultInit**  
| method and the **somDestruct** method. When you, in your program, use the  
| **somDefaultInit** method and the **somDestruct** method instead of the **somInit** and  
| **somUninit** methods, then you must override them as well. See "Multiple  
| Inheritance" on page 2-5 for more information.

---

## Four Scenarios

This section uses four scenarios to illustrate the concept of language neutrality:

- Scenario 1: A C client program using DTS C++ classes
- Scenario 2: A COBOL client program using DTS C++ classes
- Scenario 3: A COBOL client program using a COBOL class that inherits from the C++ classes.
- Scenario 4: A C++ client program using Scenario 3's COBOL class that inherits from the C++ classes.

For details about performing the individual process steps, see the appropriate programming language documentation.

We used the following levels of products to produce these scenarios:

- OS/390 Version 1 Release 3 SOMobjects
- OS390 V1R3.0 C/C++ (C/C++ Version 3, Release 3)
- IBM COBOL for OS/390 and VM Version 2 Release 1 (DLL support driver)
- OS390 V1R3.0 Language Environment (LE Version 1, Release 7)

The class library for the scenarios consists of three C++ classes which inherit from the SOMObject class provided with SOMobjects. The C++ classes provide the capability to create, update, and print entries for a phone book. A COBOL class inherits one of the C++ classes, overrides the method to print a single phone book entry, and adds a method to print all the entries.

Interface information about SOMobjects classes is made available to the COBOL compiler through the interface repository (IR). Interface repository entries are created by compiling the IDL file for a class using the SOM compiler with its *ir* emitter.

Interface information about SOMobjects classes is made available to the C++ compiler by including header files (.hh) into the source programs, as is usual for C++ application development.

**Note:** In all of the scenarios, the following applies:

*userprfx* is the user defined prefix under which these sample scenarios are installed.

*somprfx* is the high level qualifier under which the SOMobjects product has been installed.

## Scenario 1: C Client Program Using DTS C++ Classes

In the first scenario, a C client program uses classes built using the Direct-to-SOM support in the C++ compiler. The following figure shows the C++ classes used by the C client in this scenario.

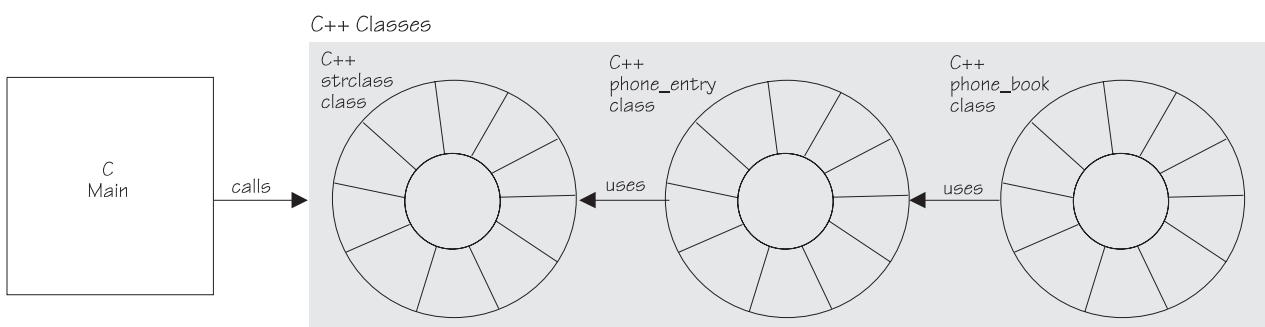


Figure 10-1 on page 10-3, with text below, shows the steps needed to complete this scenario.

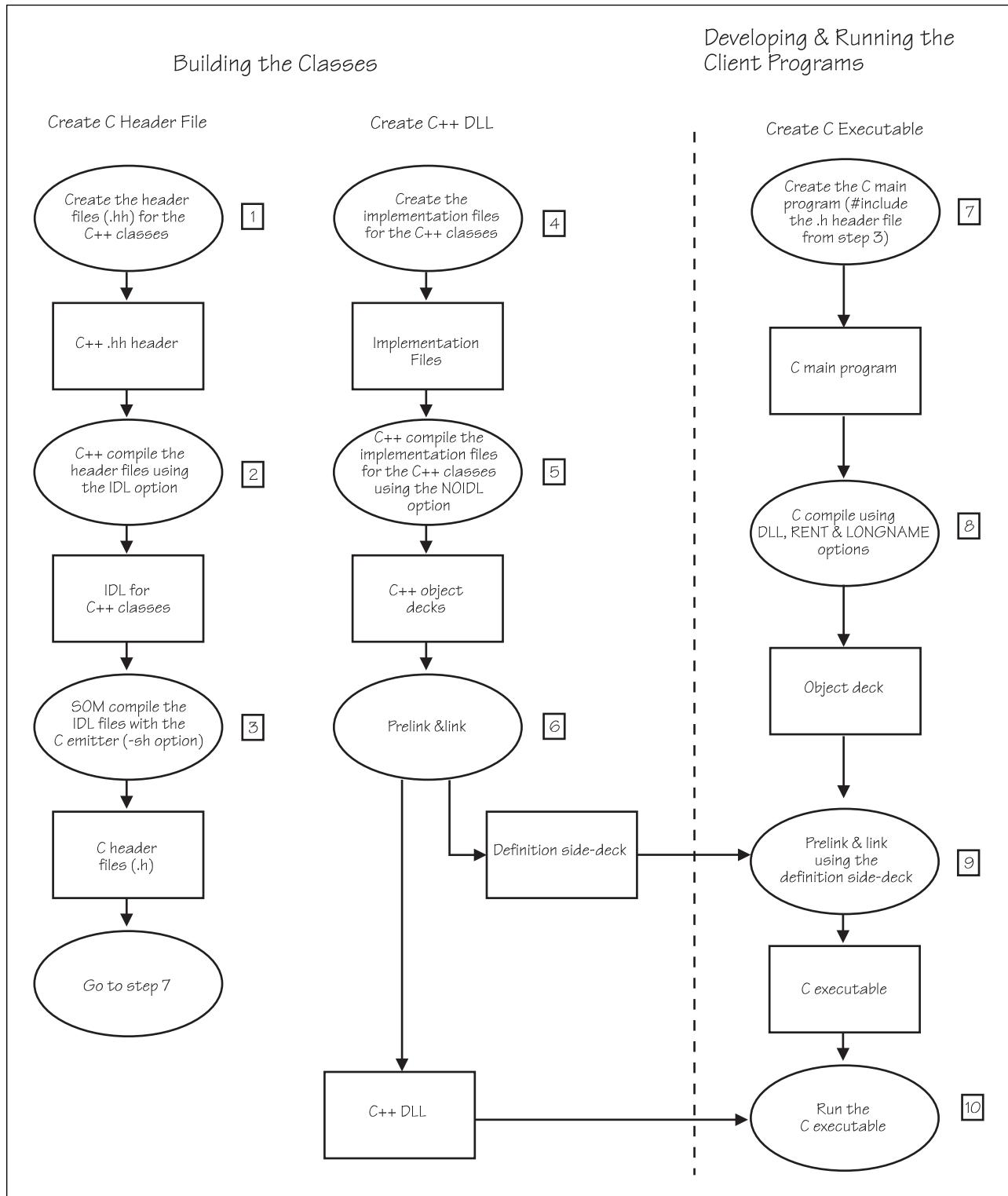


Figure 10-1. C program using DTS C++ classes.

## Building the Classes

In order for SOM objects to be shared across languages, IDL is required to communicate the class interface to the other languages. The easiest method to create the IDL is to use compilers which support Direct-to-SOM (DTS).

### Create the Header Files

**Step 1:** Create the header files (.hh) for the C++ classes, using appropriate #pragmas and options for DTS support. The pbook, pentry and mystring header files are contained in Figure 10-2, Figure 10-3 on page 10-5 and Figure 10-4 on page 10-6.

```
#ifndef PBOOK_H
#define PBOOK_H

#include "pentry.hh"

// turn on implicit SOM mode
#pragma SOMAsDefault(on)

// turn off mangling of C++ names
#pragma SOMNoMangling(on)

// define phone book class
class phone_book {

 // prevent mangling of class name
 #pragma SOMClassName(*,"phone_book")

 // use this pragma to generate an include of the IDL defining the
 // phone_entry interface
 #pragma SOMIDLTypes(*,phone_entry)

private:
 phone_entry *current;
 phone_entry *head;
 int total_nodes;

public:
 // make all functions virtual, except for constructor and
 // destructor, to allow them to be overridden by subclasses
 virtual phone_entry*
 *search(CHOICE type,char *input1,char *input2 = NULL);
 virtual phone_entry* add(char *, char *, char *, char *, char *);
 virtual void load(char *);
 virtual void save(char *);
 virtual phone_entry *erase();
 virtual phone_entry *next();
 virtual phone_entry *prev();
 virtual phone_entry *get_head();
 virtual void reset();
 virtual void display_entry(phone_entry*);

 phone_book();
 ~phone_book();
};

// restore to the previous state
#pragma SOMAsDefault(pop)
#pragma SOMNoMangling(pop)
#endif
```

Figure 10-2. C++ pbook header file for Scenario 1.

```

#ifndef PENTRY_H
#define PENTRY_H

#include "mystring.hh"

// turn on implicit SOM mode
#pragma SOMAsDefault(on)

// turn off mangling of C++ names
#pragma SOMNoMangling(on)

// use all lower case for the enumerators
// (#pragma SOMNoMangling(ON) does not turn off mangling of this type
// of data)
enum CHOICE {find_name, find_first, find_last, find_phone,
 find_addr, find_com};

// define phone entry class
class phone_entry {

 // prevent mangling of class name
 #pragma SOMClassName(*,"phone_entry")

 // use this pragma to generate an include of the IDL defining the
 // strclass interface
 #pragma SOMIDLTypes(*,strclass)

private:
 strclass *l_name;
 strclass *f_name;
 strclass *phone;
 strclass *address;
 strclass *comment;
 phone_entry *before;
 phone_entry *after;

public:
 // make all functions virtual, except for constructor and
 // destructor, to allow them to be overridden by subclasses
 virtual void set_lname(char *);
 virtual void set_fname(char *);
 virtual void set_phone(char *);
 virtual void set_address(char *);
 virtual void set_comment(char *);
 virtual void set_before(phone_entry *);
 virtual void set_after(phone_entry *);
 virtual phone_entry* get_before();
 virtual phone_entry* get_after();
 virtual int sub_entry(CHOICE type, char *input);
 virtual int compare_entry(CHOICE, char *);
 virtual void get_entry(char *, char *, char *, char *, char *);

 phone_entry(char *,char *,char *, char *,char *);
 phone_entry();
 ~phone_entry();

 // give a unique name to the overloaded constructor to allow it to
 // be accessed by other languages
 #pragma SOMMethodName(\
 phone_entry(char *,char *,char *, char *,char *),"phone_entry_init")
};

// restore to the previous state
#pragma SOMAsDefault(pop)
#pragma SOMNoMangling(pop)

#endif

```

*Figure 10-3. C++ pantry header file for Scenario 1.*

```

#ifndef MYSTRING_H
#define MYSTRING_H

// turn on implicit SOM mode
#pragma SOMAsDefault(on)

// turn off mangling of C++ names
#pragma SOMNoMangling(on)

// define string class
class strclass {

 // prevent mangling of class name
 #pragma SOMClassName(*,"strclass")

private:
 char *str;

public:
 // make all functions virtual, except for constructor and
 // destructor, to allow them to be overridden by subclasses
 virtual int sub_str(char *);
 virtual int compare_str(char *);
 virtual int replace_str(char *);
 virtual void get_str(char *);
 virtual void upper (char *);

 strclass(char *);
 strclass();
 ~strclass();

 // give a unique name to the overloaded constructor to allow it to
 // be accessed by other languages
 #pragma SOMMethodName(strclass(char),"strclass_init")
};

// restore to the previous state
#pragma SOMAsDefault(pop)
#pragma SOMNoMangling(pop)

#endif

```

*Figure 10-4. C++ mystring header file for Scenario 1.*

**Step 2:** C++ compile the header files for the C++ classes, using the IDL option, obtaining the IDL files for the C++ classes. Figure 10-5 on page 10-7 contains the JCL that does this compile.

```

//S1STEP2 JOB <JOB CARD PARAMETERS>
//ORDER JCLLIB ORDER=(CBC.SCBCPRC)
//***** SCENARIO 1 STEP 2: CREATE IDL FROM .HH HEADERS FOR C++ CLASSES ****
//*
//*
//** COMPILER: C++
//** COMPILE OPTIONS: IDL SEARCH
//** INPUT: .HH HEADER FILES IN 'userprfx.hh'
//** OUTPUT: IDL IN 'userprfx.IDL'
//***** -----
//-----GENERATE IDL FOR PBOOK-----
//COMP1 EXEC CBCC,
// LIBPRFX=CEE,LNGPRFX=CBC,
// CPARM='IDL OPTFILE(DD:OPTION)',
// INFILe=userprfx.hh(PBOOK),
// OUTFILE=NULLFILE
//COMPILE.OPTION DD DATA,DLM='/>
SEARCH('userprfx.+','CEE.SCEEH.+',
 'CBC.SCLBH.+','somprfx.SGOSHH.+',
 'somprfx.SGOSH.+','somprfx.SGOSXH.+')
/>
//COMPILE.SYSUT15 DD DSN=userprfx.IDL(PBOOK),DISP=SHR
//-----GENERATE IDL FOR PENTRY-----
//COMP2 EXEC CBCC,
// LIBPRFX=CEE,LNGPRFX=CBC,
// CPARM='IDL OPTFILE(DD:OPTION)',
// INFILe=userprfx.hh(PENTRY),
// OUTFILE=NULLFILE
//COMPILE.OPTION DD DATA,DLM='/>
SEARCH('userprfx.+','CEE.V1R5M0.SCEEH.+',
 'CBC.SCLBH.+','somprfx.SGOSHH.+',
 'somprfx.SGOSH.+','somprfx.SGOSXH.+')
/>
//COMPILE.SYSUT15 DD DSN=userprfx.IDL(PENTRY),DISP=SHR
//-----GENERATE IDL FOR MYSTRING-----
//COMP3 EXEC CBCC,
// LIBPRFX=CEE,LNGPRFX=CBC,
// CPARM='IDL OPTFILE(DD:OPTION)',
// INFILe=userprfx.hh(MYSTRING),
// OUTFILE=NULLFILE
//COMPILE.OPTION DD DATA,DLM='/>
SEARCH('userprfx.+','CEE.SCEEH.+',
 'CBC.SCLBH.+','somprfx.SGOSHH.+',
 'somprfx.SGOSH.+','somprfx.SGOSXH.+')
/>
//COMPILE.SYSUT15 DD DSN=userprfx.IDL(MYSTRING),DISP=SHR

```

Figure 10-5. JCL to compile the header files for the C++ classes in Scenario 1.

**Step 3:** SOM compile the generated IDL files with the C emitter (e.g. sc with -sh option) to create C header files (.h) for the classes. The following are the TSO line commands using the SOM compiler to generate the C header files for the three classes in Scenario 1:

```

sc -V -sh -D_private_ 'userprfx.idl(strclass)'
sc -V -sh -D_private_ 'userprfx.idl(pbook)'
sc -V -sh -D_private_ 'userprfx.idl(pentry)'

```

## Create C++ DLL

**Step 4:** Create (with editor) the implementation files for the C++ classes, coding #include statements for the .hh header files created in Step 1. The pbook, pentry and mystring implementation files for Scenario 1 are contained in Figure 10-6 on page 10-9, Figure 10-7 on page 10-13 and Figure 10-8 on page 10-15.

```

#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include "pbook.hh"

// export the required 3 symbols for the SOM class phone_book
#pragma export(phone_bookClassData)
#pragma export(phone_bookCClassData)
#pragma export(phone_bookNewClass)

phone_entry *phone_book::search(CHOICE type,char *input1,char *input2) {
 phone_entry *start = current;
 phone_entry *return_val = NULL;
 int result;

 switch(type) {
 case find_name:
 // search for full name
 // input1 is last name, input2 is first name
 start = head;

 do {
 result = start->compare_entry(find_last,input1);

 if (result == 0) {
 result = start->compare_entry(find_first,input2);
 if (result == 0) {
 current = start;
 return_val = start;
 break;
 }
 }
 }

 if (result > 0) start = start->get_after();

 if ((result < 0) || (start == head)) {
 current = start->get_before();
 return_val = NULL;
 break;
 }
 } while (start != head);
 break;

 case find_last:
 do {
 result = start->compare_entry(find_last,input1);
 if (result == 0) {
 current = start;
 return_val = start;
 break;
 }
 start = start->get_after();
 } while (start != current);
 break;

 case find_first:
 do {
 result = start->compare_entry(find_first,input1);
 if (result == 0) {
 current = start;
 return_val = start;
 break;
 }
 }
 }
}

```

Figure 10-6 (Part 1 of 4). Implementation file pbook for Scenario 1.

```

 start = start->get_after();
 } while (start != current);
 break;
case find_phone:
do {
 result = start->compare_entry(find_phone,input1);
 if(result == 0) {
 current = start;
 return_val = start;
 break;
 }
 start = start->get_after();
} while (start != current);
break;

case find_addr:
do {
 result = start->sub_entry(find_addr,input1);
 if(result == 0) {
 current = start;
 return_val = start;
 break;
 }
 start = start->get_after();
} while (start != current);
break;

case find_com:
do {
 result = start->sub_entry(find_com,input1);
 if(result == 0) {
 current = start;
 return_val = start;
 break;
 }
 start = start->get_after();
} while (start != current);
break;
}
return(return_val);
}

phone_entry *phone_book::add(char *lname, char *fname,
char *num,char *addr, char *com) {
char input;
phone_entry *newentry, *temp;
newentry = new phone_entry(lname, fname, num, addr , com);

reset();
temp = search(find_name,lname,fname);

if (temp != NULL) {
 temp = current->get_before();
 total_nodes--;
 erase();
}
// if entry not found, add after current, so set temp to current
else temp = current;

if(temp -> get_after() == head) head ->set_before(newentry);
(temp->get_after())->set_before(newentry);

newentry->set_after(temp->get_after());
newentry->set_before(temp);
}

```

Figure 10-6 (Part 2 of 4). Implementation file pbook for Scenario 1.

```

temp->set_after(newentry);

current = newentry;
total_nodes++;
return(current);
}
void phone_book::load(char *file) {
 fstream infile;
 char buffer[80];
 char lname[16], fname[16], num[12], addr[81], com[81];

 infile.open(file,ios::in);
 if (!infile)
 return;

 while (infile.getline(buffer,80)) {
 sscanf(buffer, "%s %s %s", lname,fname,num);
 infile.getline(addr,80);
 infile.getline(com,80);
 add(lname,fname,num,addr,com);
 }

 infile.close();
}

void phone_book::save(char *file) {
 fstream outfile;
 char buffer[80];
 char lname[16], fname[16], num[12], addr[81], com[81];

 phone_entry *start = head->get_after();

 outfile.open(file, ios::out);
 if (!outfile)
 perror("couldn't save to file");

 do {
 start->get_entry(lname,fname, num, addr, com);
 outfile << lname << " ";
 outfile << fname << " ";
 outfile << num << endl;
 outfile << addr << endl;
 outfile << com << endl;
 start = start->get_after();
 } while(start != head);

 outfile.close();
}

phone_entry *phone_book::erase()
{
 phone_entry *temp;
 temp = current;
 (current -> get_before()) -> set_after(current->get_after());
 (current -> get_after()) -> set_before(current -> get_before());
 current = temp -> get_before();
 if (current == head)
 current = current -> get_before();
 return(current);
}

phone_entry *phone_book::next()
{
 current = current -> get_after();
 if (current == head) {

```

*Figure 10-6 (Part 3 of 4). Implementation file pbook for Scenario 1.*

```

 current = head -> get_after();
 }
 return(current);
}

phone_entry *phone_book::prev()
{
 current = current -> get_before();
 if (current == head) {
 current = head -> get_before();
 }
 return(current);
}

phone_entry *phone_book::get_head() {
 return(head);
}

void phone_book::reset()
{
 current = head;
}

void phone_book::display_entry(phone_entry* entry)
{
 char last[16], first[16], number[12], addr[80], com[80];
 entry->get_entry(last,first,number,addr,com);
 printf("%s, %s <%s>\n",last, first, number);
 printf("%s\n",addr);
 printf("%s\n",com);
}

// phone_book constructor loads data from the file OOILC.DATA
phone_book::phone_book() {
 head = new phone_entry(" "," ","0000000000"," ",
 " PHONE BOOK IS EMPTY");
 current = new phone_entry(" "," "," "," "," ");
 head->set_before(head);
 head->set_after(head);
 load("OOILC.DATA");
 reset();
}

// phone_book destructor saves data to the file OOILC.DATA
phone_book::~phone_book() {
 save("OOILC.DATA");
 delete head;
}

```

*Figure 10-6 (Part 4 of 4). Implementation file pbook for Scenario 1.*

```

#include <stdio.h>
#include "mystring.hh"
#include "pentry.hh"

void phone_entry::set_lname(char *lname)
{
 l_name->replace_str(lname);
}

void phone_entry::set_fname(char *fname)
{
 f_name->replace_str(fname);
}

void phone_entry::set_phone(char *number)
{
 phone->replace_str(number);
}

void phone_entry::set_address(char *addr)
{
 address->replace_str(addr);
}

void phone_entry::set_comment(char *info)
{
 comment->replace_str(info);
}

void phone_entry::set_before(phone_entry *prev)
{
 before = prev;
}

void phone_entry::set_after(phone_entry *next)
{
 after = next;
}

phone_entry * phone_entry::get_before()
{
 return(before);
}

phone_entry * phone_entry::get_after()
{
 return(after);
}

int phone_entry::sub_entry(CHOICE type, char *input)
{
 int result;
 switch(type) {
 case find_phone:
 result = phone->sub_str(input);
 break;
 case find_addr:
 result = address->sub_str(input);
 break;
 case find_com:
 result = comment->sub_str(input);
 break;
 }
 return(result);
}

```

*Figure 10-7 (Part 1 of 2). Implementation file pentry for Scenario 1.*

```

int phone_entry::compare_entry(CHOICE type, char *input)
{
 int result;
 switch(type) {
 case find_last:
 result = l_name->compare_str(input);
 break;
 case find_first:
 result = f_name->compare_str(input);
 break;
 case find_phone:
 result = phone->compare_str(input);
 break;
 case find_addr:
 result = address->compare_str(input);
 break;
 case find_com:
 result = comment->compare_str(input);
 break;
 }
 return(result);
}

void phone_entry::get_entry(char *last, char* first, char *number,
 char *addr, char* com)
{
 l_name->get_str(last);
 f_name->get_str(first);
 phone->get_str(number);
 address->get_str(addr);
 comment->get_str(com);
}

phone_entry::phone_entry(char *lname, char *fname, char *number,
 char *addr, char *com) {
 l_name = new strclass(lname);
 f_name = new strclass(fname);
 phone = new strclass(number);
 address = new strclass(addr);
 comment = new strclass(com);
}

phone_entry::phone_entry() {
 l_name = new strclass;
 f_name = new strclass;
 phone = new strclass;
 address = new strclass;
 comment = new strclass;
}

phone_entry::~phone_entry() {
 delete l_name;
 delete f_name;
 delete phone;
 delete address;
 delete comment;
}

```

*Figure 10-7 (Part 2 of 2). Implementation file pentry for Scenario 1.*

```

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "mystring.hh"

int strclass::sub_str(char *substr)
{
 upper(substr);
 if (strstr(str,substr) != NULL)
 return(0);
 else
 return(1);
}

int strclass::compare_str(char *input)
{
 upper(input);
 if (strcmp(input,str) > 0)
 return(1);
 else if (strcmp(input,str) < 0)
 return(-1);
 else
 return(0);
}

int strclass::replace_str(char *newstr)
{
 delete str;
 upper(newstr);
 int len = strlen(newstr);
 str = new char [len + 1];
 if (strcpy(str,newstr) != NULL)
 return(1);
 else
 return(0);
}

void strclass::get_str(char *input)
{
 strcpy(input,str);
}

void strclass::upper(char *input)
{
 int len = strlen(input);
 for (int i=0; i <= len; i++)
 {
 input[i] = toupper(input[i]);
 }
}

```

*Figure 10-8 (Part 1 of 2). Implementation file mystring for Scenario 1.*

```
strclass::strclass(char *input)
{
 int len = strlen(input);
 str = new char [len+1];

 upper(input);
 strcpy(str,input);
}
strclass::strclass()
{
 str = new char [1];
 strcpy(str,"");
}
strclass::~strclass()
{
 delete str;
}
```

Figure 10-8 (Part 2 of 2). Implementation file mystring for Scenario 1.

**Step 5:** C++ compile the implementation files for the C++ classes, using the NOIDL option, which is the default, obtaining the object decks for the C++ classes. Figure 10-9 on page 10-17 contains the JCL that does this compile.

```

//S1STEP5 JOB <JOB CARD PARAMETERS>
//ORDER JCLLIB ORDER=(CBC.SCBCPRC)
//***** SCENARIO 1 STEP 5: COMPILE THE SOURCE FOR THE C++ CLASSES ****
//*
//* COMPILER: C++
//* COMPILE OPTIONS: SEARCH
//* 'NIDL' IS USED BY DEFAULT
//* INPUT: 'userprfx.CPP'
//* OUTPUT: 'userprfx.OBJ'
//***** -----
//*-----COMPILE PBOOK-----
//COMP1 EXEC CBCC,
// LIBPRFX=CEE,
// LNGPRFX=CBC,
// CPARM='OPTFILE(DD:OPTION)',
// INFILE=userprfx.CPP(PBOOK)
//COMPILE.OPTION DD DATA,DLM='/>
SEARCH('userprfx.+','CEE.SCEEH.+',
 'CBC.SCLBH.+','somprfx.SGOSHH.+',
 'somprfx.SGOSH.+','somprfx.SGOSXH.+')
/>
//COMPILE.SYSLIN DD DSN=userprfx.OBJ(PBOOK),DISP=SHR
//*-----COMPILE PENTRY-----
//COMP2 EXEC CBCC,
// LIBPRFX=CEE.V1R5M0.SPE,
// LNGPRFX=CBC,
// CPARM='OPTFILE(DD:OPTION)',
// INFILE=userprfx.CPP(PENTRY)
//COMPILE.OPTION DD DATA,DLM='/>
SEARCH('userprfx.+','CEE.SCEEH.+',
 'CBC.SCLBH.+','somprfx.SGOSHH.+',
 'somprfx.SGOSH.+','somprfx.SGOSXH.+')
/>
//COMPILE.SYSLIN DD DSN=userprfx.OBJ(PENTRY),DISP=SHR
//*-----COMPILE MYSTRING-----
//COMP3 EXEC CBCC,
// LIBPRFX=CEE.V1R5M0.SPE,
// LNGPRFX=CBC,
// CPARM='OPTFILE(DD:OPTION)',
// INFILE=userprfx.CPP(MYSTRING)
//COMPILE.OPTION DD DATA,DLM='/>
SEARCH('userprfx.+','CEE.SCEEH.+',
 'CBC.SCLB3H.+','somprfx.SGOSHH.+',
 'somprfx.SGOSH.+','somprfx.SGOSXH.+')
/>
//COMPILE.SYSLIN DD DSN=userprfx.OBJ(MYSTRING),DISP=SHR

```

Figure 10-9. JCL to compile the implementation files for the C++ classes in Scenario 1.

**Step 6:** Prelink and link to create a C++ DLL and a corresponding definition side-deck. Figure 10-10 on page 10-18 contains the JCL that does the prelink and link to create the C++ DLL and the corresponding definition side-deck.

```

//S1STEP6 <JOB CARD PARAMETERS>
//ORDER JCLLIB ORDER=(CBC.SCBCPRC)
//*****SCENARIO 1 STEP 6: PRELINK AND LINK DLL FOR C++ CLASSES*****
//*
//** INPUT: 'userprfx.OBJ' */
//** OUTPUT: DLL IN 'userprfx.LOAD(PBOOK)' */
//** DEFINITION SIDE-DECK IN 'userprfx.SYSDEFSD(PBOOK)' */
//*****LINK1 EXEC CBCL,
// LIBPRFX=CEE,CLBPRFX=CBC,
// INFILE=userprfx.OBJ(PBOOK),
// OUTFILE='userprfx.LOAD(PBOOK),DISP=OLD'
//PLKED.SYSIN2 DD DATA,DLM='>
INCLUDE OBJECT(MYSTRING)
INCLUDE OBJECT(PENTRY)
INCLUDE IMPORT(GOSSOMK)
NAME PBOOK(R)
/>
//PLKED.IMPORT DD DSN=somprfx.SGOSIMP,DISP=SHR
//PLKED.OBJECT DD DSN=userprfx.OBJ,DISP=SHR
//PLKED.SYSDEFSD DD DSN=userprfx.SYSDEFSD(PBOOK),DISP=SHR

```

*Figure 10-10. JCL to prelink and link to create the C++ DLL and the corresponding definition side-deck in Scenario 1.*

For more information on the creation of a DLL, see Chapter 5, “Building Class Libraries” on page 5-1.

## Developing and Running the Client Program

### Create C Executable

**Step 7:** Create (with editor) the C main program, coding #include statements for the .h header files created in Step 3. Figure 10-11 on page 10-19 contains the C main program for Scenario 1:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "pentry.h"
#include "pbook.h"

main()
{
 phone_book mine;
 phone_entry temp;
 phone_entry output = NULL;

 char input[2], lname[16], fname[16], number[11], addr[81], com[81];
 int look_ahead;
 int int_input;

 /* set up environment variable */
 Environment *ev = somGetGlobalEnvironment();

 /* call the default object constructor */
 mine = phone_bookNew();

 do {
 printf("\n***** *\n");
 printf("SOM Multi-Language Phone Book \n");
 printf(".....brought to you by C using C++ methods. \n\n");
 printf("1 - Search for an entry \n");
 printf("2 - Go to the next entry \n");
 printf("3 - Go to the previous entry \n");
 printf("4 - Add an entry \n");
 printf("5 - Delete an entry \n");
 printf("Q - Quit the program \n\n");

 /* show current entry, which is the result of a search, a new */
 /* entry added, etc. */
 if(output != NULL) {
 printf("CURRENT ENTRY: \n");
 _display_entry(mine,ev,output);
 }

 printf("\n Enter a number or 'Q' to quit. \n");
 scanf("%s",input);
 int_input = atoi(input);

 switch(int_input) {
 /* search */
 case 1:
 printf("\n CHOICE 1 \n");
 printf("1. Search for a last name \n");
 printf("2. Search for a first name \n");
 printf("3. Search for a phone number \n");
 printf("4. Search for part of a address \n");
 printf("5. Search for part of a comment \n\n");
 scanf("%s",input);
 int_input = atoi(input);

 switch(int_input) {
 /* search for last name */
 case 1:
 printf("enter a last name to be searched \n");
 scanf("%s",lname);
 temp = _search(mine,ev,find_last,lname,NULL);
 if(temp == NULL)
 printf("*** NO OTHER ENTRY FOUND *** \n");
 else
 output = temp;
 break;
 }
 }
}

```

Figure 10-11 (Part 1 of 3). C main program for Scenario 1.

```

/* search for first name */
case 2:
 printf("enter a first name to be searched \n");
 scanf("%s",fname);
 temp = _search(mine,ev,find_first, fname,NULL);
 if(temp == NULL)
 printf("*** NO OTHER ENTRY FOUND *** \n");
 else
 output = temp;
 break;

/* search for phone number */
case 3:
 printf("enter a phone number to be searched \n");
 scanf("%s",number);
 temp = _search(mine,ev,find_phone,number,NULL);
 if(temp == NULL)
 printf("*** NO OTHER ENTRY FOUND *** \n");
 else
 output = temp;
 break;

/* search for address */
case 4:
 printf("enter an address to searched \n");
 scanf("%s",addr);
 temp = _search(mine,ev,find_addr,addr,NULL);
 if(temp == NULL)
 printf("*** NO OTHER ENTRY FOUND *** \n");
 else
 output = temp;
 break;

/* search for comment */
case 5:
 printf("enter a comment to searched \n");
 scanf("%s",addr);
 /* must pass NULL as the last parameter */
 /* SOM doesn't permit the use of a default value */
 temp = _search(mine,ev,find_addr,com,NULL);
 if(temp == NULL)
 printf("*** NO OTHER ENTRY FOUND *** \n");
 else
 output = temp;
 break;
}

/* go to next entry */
case 2:
 printf("\n CHOICE 2 \n");
 output = _next(mine,ev);
 break;

/* go to previous entry */
case 3:
 printf("\n CHOICE 3 \n");
 output = _prev(mine,ev);
 break;

```

*Figure 10-11 (Part 2 of 3). C main program for Scenario 1.*

```

/* add an entry */
case 4:
 printf("\n CHOICE 4 \n");
 printf("Please enter the following in the given order \n");
 printf("first name (maximum 15 characters) \n");
 printf("last name (maximum 15 characters) \n");
 printf("phone number (nnn-mmmm) \n");
 printf("address (maximum 80 characters) \n");
 printf("comment (maximum 80 characters) \n");

 scanf("%s %s %s", fname, lname, number);
 /* if next char is \n, ignore it */
 if ((look_ahead=getchar()) != '\n') ungetc(look_ahead,stdin);
 gets(addr);
 gets(com);

 output = _add(mine,ev,lname,fname,number,addr,com);
 break;

/* erase current entry */
case 5:
 printf("\n CHOICE 5\n");
 printf("ERASING current entry!! \n");
 output = _erase(mine,ev);
 break;
}
} while(strcmp(input,"Q") && strcmp(input,"q"));

/* call destructor and delete object */
_somFree(mine);
}

```

Figure 10-11 (Part 3 of 3). C main program for Scenario 1.

**Step 8:** C compile the C main program (using the DLL, RENT and LONGNAME options) to create the object deck for the C program. Figure 10-12 contains the JCL to C compile the C main program for Scenario 1.

```

//S1STEP8 JOB <JOB CARD PARAMETERS>
//ORDER JCLLIB ORDER=(CBC.SCBCPRC)
//*****SCENARIO 1 STEP 8: COMPILE THE SOURCE FOR THE C MAIN PROGRAM*****
//*
//** COMPILER: C
//** COMPILE OPTIONS: DLL LONGNAME RENT
//** INPUT: 'userprfx.C'
//** OUTPUT: 'userprfx.OBJ'
//** USES: .H HEADER FILES FOR THE C++ CLASSES IN 'userprfx.H'
//*
//*/
//COMP1 EXEC EDCC,
// LIBPRFX=CEE,LNGPRFX=CBC,
// CPARM='DLL LONGNAME RENT',
// INFILE='userprfx.C(MAIN)',
// OUTFILE='userprfx.OBJ(MAIN),DISP=SHR'
//COMPILE.SYSLIB DD DSN=userprfx.H,DISP=SHR
// DD DSN=CEE.SCEEH.H,DISP=SHR
// DD DSN=CBC.SCLBH.H,DISP=SHR
// DD DSN=somprfx.SGOSH.H,DISP=SHR

```

Figure 10-12. C compile JCL of the main program for Scenario 1.

**Step 9:** Prelink and link the object deck for the C main program with the definition side-deck for the C++ DLL.

Figure 10-13 on page 10-22 contains the JCL to prelink and link the object deck for the C main program for Scenario 1.

```
//S1STEP9 JOB <JOB CARD PARAMETERS>
//ORDER JCLLIB ORDER=(CBC.SCBCPRC)
//*****SCENARIO 1 STEP 9: PRELINK AND LINK THE C MAIN PROGRAM ****
//*
//** INPUT: 'userprfx.OBJ(MAIN)' *
//** OUTPUT: 'userprfx.LOAD(MAIN)' *
//** USES: SIDE-DECK FOR THE C++ DLL IN 'userprfx.SYSDEFSD' *
//*****LINK1 EXEC CBCL,
// LIBPRFX=CEE,
// CLBPRFX=CBC,
// INFILE=userprfx.OBJ(MAIN),
// OUTFILE='userprfx.LOAD(MAIN),DISP=OLD'
//PLKED.SYSIN2 DD DATA,DLM='>
INCLUDE IMPORT1(PBOOK)
INCLUDE IMPORT2(GOSSOMK)
NAME MAIN(R)
/>
//PLKED.IMPORT1 DD DSN=userprfx.SYSDEFSD,DISP=SHR
//PLKED.IMPORT2 DD DSN=somprfx.SGOSIMP,DISP=SHR
//PLKED.OBJECT DD DSN=userprfx.OBJ,DISP=SHR
```

Figure 10-13. C prelink and link JCL of the object deck for Scenario 1.

**Step 10:** Run the C executable, including the C++ DLL in the GO.STEPLIB concatenation.

Figure 10-14 on page 10-23 contains the JCL to run the C executable for Scenario 1.

```

//S1STEP10 JOB <JOB CARD PARAMETERS>
//ORDER JCLLIB ORDER=(CBC.SCBCPRC)
//***** *****/
/* SCENARIO 1 STEP 10: RUN THE C MAIN PROGRAM */
/* */
/* INPUT: MAIN PROGRAM IN 'userprfx.LOAD(MAIN)' */
/* USES: C++ CLASSES DLL IN 'userprfx.LOAD(PBOOK)' */
/* */
/* THE INPUT DATA FOR THE PROGRAM IS PROVIDED USING SYSIN. */
/* */
/* THE PROGRAM STORES THE PHONE BOOK AS OOILC.DATA. THIS DATA IN */
/* THIS FILE IS LOADED EACH TIME THE PROGRAM IS RUN, IF THE FILE */
/* EXISTS. */
//***** *****/
//RUNJOB EXEC PGM=MAIN,REGION=4M
//STEPLIB DD DSN=userprfx.LOAD,DISP=SHR
// DD DSN=CEE.SCEERUN,DISP=SHR
// DD DSN=CBC.SCLBDLL,DISP=SHR
// DD DSN=somprfx.SGOSLOAD,DISP=SHR
//SOMENV DD DSN=somprfx.SGOSPROF(GOSENV),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSIN DD *
4
Sylvester Cat 800WDisney
Disney World, Orlando, FL 12345-6789
Large tuxedo-tabby cat who loves to chase small yellow tweety birds.
4
Tweetie Bird 1115551234
Disney World, California 99887-6655
Sylvester's favorite food.
3
3
2
1 1 Bird
1 3 800WDisney
5
2
2
Q
/*

```

*Figure 10-14. C executable JCL for Scenario 1.*

The output of the RUNJOB step for Scenario 1 is in Figure 10-15 on page 10-24.

```

1
* * * * *
OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
- Enter a number or 'Q' to quit.
0 CHOICE 4
Please enter the following in the given order
first name (maximum 15 characters)
last name (maximum 15 characters)
phone number (nnn-mmmm)
address (maximum 80 characters)
comment (maximum 80 characters)
-* * * * *
OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENt ENTRY:
CAT, SYLVESTER <800WDISNEY>
DISNEY WORLD, ORLANDO, FL 12345-6789
LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.
0 Enter a number or 'Q' to quit.
0 CHOICE 4
Please enter the following in the given order
first name (maximum 15 characters)
last name (maximum 15 characters)
phone number (nnn-mmmm)
address (maximum 80 characters)
comment (maximum 80 characters)
-* * * * *
OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENt ENTRY:
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.
0 Enter a number or 'Q' to quit.
0 CHOICE 3
-* * * * *

```

Figure 10-15 (Part 1 of 4). Output of the RUNJOB step for Scenario 1.

```

OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENT ENTRY:
CAT, SYLVESTER <800WDISNEY>
DISNEY WORLD, ORLANDO, FL 12345-6789
LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.
0 Enter a number or 'Q' to quit.
0 CHOICE 3
* * * * *
OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENT ENTRY:
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.
0 Enter a number or 'Q' to quit.
0 CHOICE 2
* * * * *
OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENT ENTRY:
CAT, SYLVESTER <800WDISNEY>
DISNEY WORLD, ORLANDO, FL 12345-6789
LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.
0 Enter a number or 'Q' to quit.
0 CHOICE 1
01. Search for a last name
2. Search for a first name
3. Search for a phone number
4. Search for part of a address
5. Search for part of a comment
-enter a last name to be searched
* * * * *

```

*Figure 10-15 (Part 2 of 4). Output of the RUNJOB step for Scenario 1.*

```

OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENT ENTRY:
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.
0 Enter a number or 'Q' to quit.
0 CHOICE 1
01. Search for a last name
2. Search for a first name
3. Search for a phone number
4. Search for part of a address
5. Search for part of a comment
-enter a phone number to be searched
-* * * * * * * * * * * *
OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENT ENTRY:
CAT, SYLVESTER <800WDISNEY>
DISNEY WORLD, ORLANDO, FL 12345-6789
LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.
0 Enter a number or 'Q' to quit.
0 CHOICE 5
ERASING current entry!!
-* * * * * * * * * * * *
OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENT ENTRY:
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.
0 Enter a number or 'Q' to quit.
0 CHOICE 2
-* * * * * * * * * * * *
OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENT ENTRY:
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.
0 Enter a number or 'Q' to quit.
0 CHOICE 2

```

Figure 10-15 (Part 3 of 4). Output of the RUNJOB step for Scenario 1.

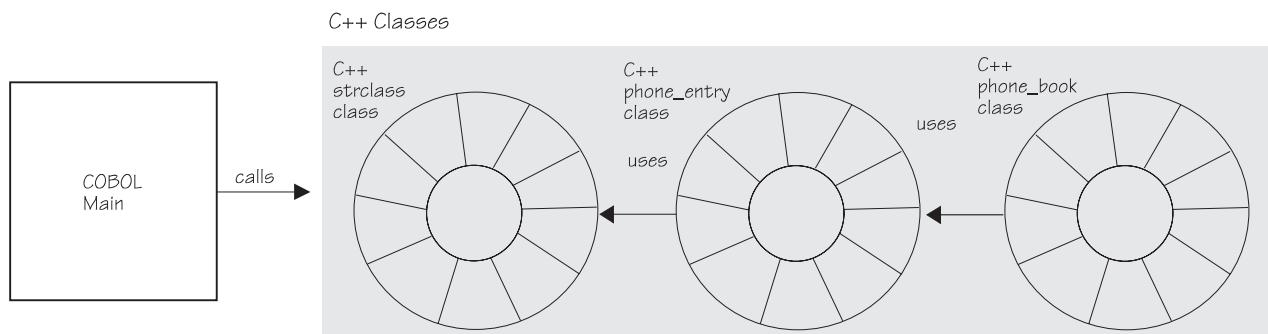
```

OSOM Multi-Language Phone Book
.....brought to you by C using C++ methods.
01 - Search for an entry
2 - Go to the next entry
3 - Go to the previous entry
4 - Add an entry
5 - Delete an entry
Q - Quit the program
0CURRENT ENTRY:
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.
0 Enter a number or 'Q' to quit.
```

Figure 10-15 (Part 4 of 4). Output of the RUNJOB step for Scenario 1.

## Scenario 2: COBOL Program Using DTS C++ Classes

In the second scenario, a COBOL main program invokes C++ class methods. Note that the creation of the C++ classes in this scenario (and subsequent scenarios) is the same as that of scenario 1.



## Building the Classes

To build the C++ classes:

### Create the Header Files and IR Entries

**Step 1:** Create (with editor) the header files (.hh) for the C++ classes, using appropriate #pragmas and options for DTS support. (See Figure 10-4 on page 10-6, Figure 10-2 on page 10-4 and Figure 10-3 on page 10-5.)

**Step 2:** C++ compile the header files for the C++ classes, using the IDL option, obtaining the IDL files for the C++ classes. (See Figure 10-5 on page 10-7.)

**Step 3:** SOM compile the generated IDL files with the *ir* emitter (e.g. sc with *-usir* option) to create IR entries for the classes. Figure 10-16 on page 10-28 contains the JCL to SOM compile, with the *usir* option, the IDL files for Scenario 2.

```

//S2STEP3 JOB <JOB CARD PARAMETERS>
//*-----
//SC PROC INDSN='',
// MEM='',
// SCPARMS='',
// SOMPRF='somprfx',
// LEPRF='CEE'
//SOMC EXEC PGM=SC,REGION=80M,
// PARM=('&SCPARMS ''&INDSN.(&MEM.)''')
//STEPLIB DD DSN=&LEPRF..SCEERUN,DISP=SHR
// DD DSN=&SOMPRF..SGOSLOAD,DISP=SHR
//SOMENV DD DUMMY
//SYSPRINT DD SYSOUT=*
// PEND
//*-----
//SC1 EXEC SC,
// MEM='MYSTRING',
// INDSN='userprfx.IDL',
// SCPARMS='-usir'
//SOMENV DD DSN=userprfx.SOMENV(mysomprf),DISP=SHR
//*-----
//SC2 EXEC SC,
// MEM='PENTRY',
// INDSN='userprfx.IDL',
// SCPARMS='-usir'
//SOMENV DD DSN=userprfx.SOMENV(mysomprf),DISP=SHR
//*-----
//SC3 EXEC SC,
// MEM='PBOOK',
// INDSN='userprfx.IDL',
// SCPARMS='-usir'
//SOMENV DD DSN=userprfx.SOMENV(mysomprf),DISP=SHR
//

```

*Figure 10-16. JCL to SOM compile the IDL files for Scenario 2.*

### Create the C++ Implementation Files

**Step 4:** Create (with editor) the implementation files for the C++ classes. (See Figure 10-8 on page 10-15, Figure 10-6 on page 10-9, Figure 10-7 on page 10-13.)

**Step 5:** C++ compile the implementation files for the C++ classes, using the NOIDL option, obtaining the object decks for the C++ classes. (See Figure 10-9 on page 10-17.)

**Step 6:** Prelink and link to create a C++ DLL and a corresponding definition side-deck. Figure 10-10 on page 10-18 contains the JCL that does the prelink and link to create the C++ DLL and the corresponding definition side-deck.

## Developing and Running the Client Program

### Create COBOL Executable

**Step 7:** Create (with editor) the COBOL main program.

Figure 10-17 on page 10-29 contains the COBOL main program for Scenario 2.

```

CBL APOST,NOCMPR2,PGMNAME(LONGMIXED)
Identification Division.
Program-Id. 'CobMain1'.
Environment Division.
Configuration Section.
Repository.
 Class PhoneEntry is 'phone_entry'
 Class PhoneBook is 'phone_book'.
Data Division.
Working-Storage Section.
01 peObj usage object reference PhoneEntry value null global.
01 pbObj usage object reference PhoneBook value null global.
01 envPtr usage pointer value null global.
01 response-input.
 02 filler pic x value '4'.
 02 filler pic x value '2'.
 02 filler pic x value '3'.
 02 filler pic x value '1'.
 02 filler pic x value '3'.
 02 filler pic x value '2'.
 02 filler pic x value '5'.
 02 filler pic x value '3'.
 02 filler pic x value '2'.
 02 filler pic x value 'Q'.
01 response-table redefines response-input.
 02 response-entry occurs 10 times.
 03 response-item pic x.
01 ndx pic 99 value 1.
01 response pic x.
 88 one value '1'.
 88 two value '2'.
 88 three value '3'.
 88 four value '4'.
 88 five value '5'.
 88 quit value 'Q'.
01 add-parameters.
 02 lname pic x(16) value z'Cat'.
 02 fname pic x(16) value z'Sylvester'.
 02 pnumb pic x(12) value z'1800WDisney'.
 02 addrs pic x(80)
 value z'Disney World, Orlando, FL 12345-6789'.
 02 commit pic x(80)
 value z'Large tuxedo-tabby cat who loves to chase
 'small yellow tweety birds.'.
01 search-parameters.
 02 scode pic x(1) value X'02'.
 02 s1str pic x(80) value z'Cat'.
 02 s2str pic x(80) value z'Sylvester'.
Linkage Section.
01 Environment-structure.
 02 major pic 9(9) binary.
 88 NO-EXCEPTION value 0.
 88 USER-EXCEPTION value 1.
 88 SYSTEM-EXCEPTION value 2.
 02 pic x(12).
Procedure Division.
 call 'somGetGlobalEnvironment' returning envPtr.
 set address of environment-structure to envPtr.
 invoke PhoneBook 'somNew' returning pbObj.

 perform Get-Response.

 perform until quit
 evaluate true
 when one
 call 'CobSearch1' using by content scode
 s1str
 s2str

```

Figure 10-17 (Part 1 of 3). COBOL main program for Scenario 2.

```

 if peObj = null
 display 'Phone entry not found.'
 else
 invoke pbObj 'display_entry'
 using by value envPtr peObj
 end-if
 when two
 invoke pbObj 'next'
 using by value envPtr
 returning peObj
 invoke pbObj 'display_entry'
 using by value envPtr peObj
 when three
 invoke pbObj 'prev'
 using by value envPtr
 returning peObj
 invoke pbObj 'display_entry'
 using by value envPtr peObj
 when four
 call 'CobAdd1' using by content lname
 fname
 pnumb
 addrs
 commt
 invoke pbObj 'display_entry'
 using by value envPtr peObj
 when five
 invoke pbObj 'erase'
 using by value envPtr
 returning peObj
 end-evaluate

 set peObj to null
 perform Get-Response
 end-perform.

 stop run.

Get-Response.
 move response-item (ndx) to response.
 add 1 to ndx.

Identification Division.
Program-Id. 'CobAdd1'.
Data Division.
Linkage Section.
01 lname pic x(16).
01 fname pic x(16).
01 pnumb pic x(12).
01 addrs pic x(80).
01 commt pic x(80).
Procedure Division using lname fname pnumb addrs commt.
 invoke pbObj 'add'
 using by value envPtr
 address of lname
 address of fname
 address of pnumb
 address of addrs
 address of commt
 returning peObj.
 exit program.
End Program 'CobAdd1'.

```

Figure 10-17 (Part 2 of 3). COBOL main program for Scenario 2.

```

Identification Division.
Program-Id. 'CobSearch1'.
Data Division.
Linkage Section.
01 scode pic x(1).
01 s1str pic x(16).
01 s2str pic x(16).
Procedure Division using scode s1str s2str.
 invoke pbObj 'search'
 using by value envPtr
 scode
 address of s1str
 address of s2str
 returning peObj.
 exit program.
End Program 'CobSearch1'.

End Program 'CobMain1'.

```

*Figure 10-17 (Part 3 of 3). COBOL main program for Scenario 2.*

**Step 8:** Compile the COBOL main program to create the object deck for the COBOL program.

The COBOL compile may optionally specify the TYPECHK option - if specified, the COBOL program will be type checked against the interfaces of the C++ classes that are in the interface repository, to validate that the usage of the classes is correct, i.e. that the method invocations are valid and have correct parameter lists. The JCL for the this compile step is combined with JCL in STEP 9 and is found in Figure 10-18 on page 10-32.

**Step 9:** Prelink and link the object deck for the COBOL main program with the definition side-deck for the C++ DLL. Figure 10-18 on page 10-32 contains the JCL to do both steps 8 and 9 for Scenario 2.

```

//S2STEP8 JOB <JOB CARD PARAMETERS>
//PROCLIB JCLLIB ORDER=(IGV.V1R3M0.SIGYPROC)
//*
//SETSOM1 SET SOMPRF='somprfx'
//SETCC SET CCPRF='CBC'
//SETLE SET LEPRF='CEE'
//SETSRC SET SRCPRF='userprfx.COBOL'
//SETOBJ SET OBJPRF='userprfx.OBJ'
//SETIMP SET IMPPRF='userprfx.SYSDEFSD'
//SETLOAD SET LOADPRF='userprfx.LOAD'
//SETSOM2 SET MYSOMPRF='userprfx.SOMPROF'
//*
//RUNJOB EXEC IGYWCPLG,REGION=56M,GOPGM=COBMAIN1,
// PARM.COBOL='RENT,PGMNAME(LM),TEST,DLL,TYPECHK',
// PARM.LKED='LIST,XREF,LET,MAP'
//COBOL.SYSIN DD DSN=&SRCPRF.(COBMAIN1),DISP=SHR
//COBOL.SYSLIN DD DSN=&OBJPRF.(COBMAIN1),DISP=SHR
//COBOL.STEPLIB DD
// DD DSN=&SOMPRF..SGOSLOAD,DISP=SHR
// DD DSN=&LEPRF..SCEERUN,DISP=SHR
//COBOL.SOMENV DD DSN=&MYSOMPRF.(mysomprf),DISP=SHR
//PLKED.STEPLIB DD DSN=&LEPRF..SCEERUN,DISP=SHR
//PLKED.SYSLIB DD DSN=&LEPRF..SCEECPP,DISP=SHR
// DD DSN=&CCPRF..SCLBCPP,DISP=SHR
//SYSIN DD DATA,DLM='/>
INCLUDE OBJECT(COBMAIN1)
INCLUDE IMPORT(PBOOK)
INCLUDE IMPORT(GOSSOMK)
NAME COBMAIN1(R)
/>
//*
//SYSOUT DD SYSOUT=*
//SYSDEFSD DD DSN=&IMPPRF.(COBMAIN1),DISP=SHR
//IMPORT DD DSN=&IMPPRF.,DISP=SHR
// DD DSN=&SOMPRF..SGOSIMP,DISP=SHR
//OBJECT DD DSN=&OBJPRF.,DISP=SHR
//LKED.SYSLMOD DD DSN=&LOADPRF.(COBMAIN1),DISP=SHR
//GO.SYSOUT DD SYSOUT=H
//GO.STEPLIB DD DSN=&LEPRF..SCEERUN,DISP=SHR
// DD DSN=&SOMPRF..SGOSLOAD,DISP=SHR
// DD DSN=&LOADPRF.,DISP=SHR
//GO.SOMENV DD DSN=&MYSOMPRF.,DISP=SHR
//
```

Figure 10-18. Compile the COBOL main program, prelink and link the C++ definition side decks and run the program for Scenario 2.

The output of the RUNJOB step for Scenario 2 is in Figure 10-19 on page 10-33.

```

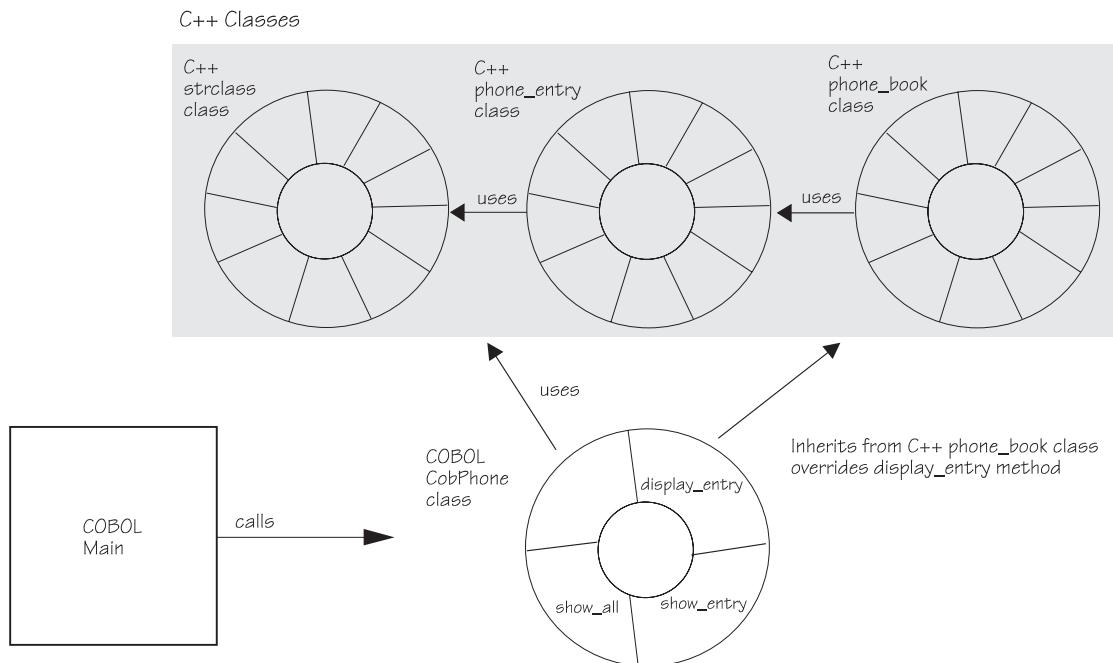
1
CAT, SYLVESTER <1800WDISNEY>
DISNEY WORLD, ORLANDO, FL 12345-6789
LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.
CAT, SYLVESTER <1800WDISNEY>
DISNEY WORLD, ORLANDO, FL 12345-6789
LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.
CAT, SYLVESTER <1800WDISNEY>
DISNEY WORLD, ORLANDO, FL 12345-6789
LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.
CAT, SYLVESTER <1800WDISNEY>
DISNEY WORLD, ORLANDO, FL 12345-6789
LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.
BIRD, TWEETIE <1115551234>
DISNEY WORLD, CALIFORNIA 99887-6655
SYLVESTER'S FAVOURITE FOOD.

```

Figure 10-19. Output of the RUNJOB step for Scenario 2.

### Scenario 3: COBOL Program Using COBOL Class that Inherits from C++ Classes

The third scenario is a COBOL main program using a COBOL class that inherits C++ classes and overrides a C++ class method.



## **Building the Classes**

To build the C++ and COBOL classes:

### **Create the Header Files and IR Entries**

**Step 1:** Create (with editor) the header files (.hh) for the C++ classes. (See Figure 10-4 on page 10-6, Figure 10-2 on page 10-4 and Figure 10-3 on page 10-5.)

**Step 2:** C++ compile the header files for the C++ classes, using the IDL option, obtaining the IDL files for the C++ classes. (See Figure 10-5 on page 10-7.)

**Step 3:** SOM compile the generated IDL files with the *ir* emitter (e.g. sc with the *-usir* option) to create IR entries for the classes. (See Figure 10-16 on page 10-28.)

### **Create the C++ Implementation Files**

**Step 4:** Create (with editor) the implementation files for the C++ classes, using appropriate #pragmas and options for DTS support. (See Figure 10-8 on page 10-15, Figure 10-6 on page 10-9, Figure 10-7 on page 10-13.)

**Step 5:** C++ compile the implementation files for the C++ classes, using the NOIDL option, obtaining the object decks for the C++ classes. (See Figure 10-9 on page 10-17.)

**Step 6:** Create (with editor) the COBOL subclass. Specify the C++ base class(es) in the INHERITS clause in the COBOL class definition.

Figure 10-20 on page 10-35 contains the COBOL subclass specifying the C++ base class(es) for Scenario 3.

```

CBL nocmpr2,pgmname(longmixed),apost

* Class definition for CobMain2

Identification Division.
Class-id. CobPhone inherits PhoneBook.
Environment Division.
Configuration Section.
Repository.
 Class CobPhone is 'CobPhone'
 Class PhoneBook is 'phone_book'
 Class PhoneEntry is 'phone_entry'.
Data Division.
Working-Storage Section.
01 envPtr pointer.
Procedure Division.

 Identification Division.
 Method-Id. 'somInit' override.
 Procedure Division.
 call 'somGetGlobalEnvironment' returning envPtr.
 exit method.
 End Method 'somInit'.

 Identification Division.
 Method-Id. 'somUninit' override.
 Procedure Division.
 set envPtr to null.
 exit method.
 End Method 'somUninit'.

 Identification Division.
 Method-Id. 'show_all'.
 Data Division.
 Local-Storage Section.
 01 peObj object reference PhoneEntry value null.
 01 tailObj object reference.
 01 parameters.
 02 lname pic x(16).
 02 fname pic x(16).
 02 pnumb pic x(12).
 02 addrs pic x(80).
 02 commt pic x(80).
 Procedure Division.
 display spaces.
 display '*****'
 display '*** PHONE BOOK ***'
 display '-----'
 invoke self 'reset'
 using by value envPtr.
 invoke self 'prev'
 using by value envPtr
 returning tailObj.
 invoke self 'reset'
 using by value envPtr.
 perform until peObj = tailObj
 invoke self 'next'
 using by value envPtr
 returning peObj
 initialize parameters
 invoke self 'show_entry'
 using by value peObj
 by reference lname fname pnumb addrs commt
 end-perform.
 exit method.
 End Method 'show_all'.

```

Figure 10-20 (Part 1 of 2). COBOL subclass for Scenario 3.

```

Identification Division.
Method-Id. 'show_entry'.
Data Division.
Local-Storage Section.
01 l-len pic 99 value 0.
01 f-len pic 99 value 0.
01 p-len pic 99 value 0.
01 a-len pic 99 value 0.
01 c-len pic 99 value 0.
Linkage Section.
01 lname pic x(16).
01 fname pic x(16).
01 pnumb pic x(12).
01 addrs pic x(80).
01 commt pic x(80).
01 peObj object reference PhoneEntry.
Procedure Division
 using by value peObj
 by reference lname fname pnumb addrs commt.
 invoke peObj 'get_entry'
 using by value envPtr
 address of lname
 address of fname
 address of pnumb
 address of addrs
 address of commt.
 inspect lname tallying l-len
 for characters before initial X'00'.
 inspect fname tallying f-len
 for characters before initial X'00'.
 inspect pnumb tallying p-len
 for characters before initial X'00'.
 inspect addrs tallying a-len
 for characters before initial X'00'.
 inspect commt tallying c-len
 for characters before initial X'00'.
 display space.
 display ' Name: ' fname(1:f-len) space lname(1:l-len).
 display ' Phone: ' pnumb(1:p-len).
 display 'Address: ' addrs(1:a-len).
 display 'Comment: ' commt(1:c-len).
 exit method.
End Method 'show_entry'.

Identification Division.
Method-Id. 'display_entry' override.
Data Division.
Local-Storage Section.
01 lname pic x(16) value spaces.
01 fname pic x(16) value spaces.
01 pnumb pic x(12) value spaces.
01 addrs pic x(80) value spaces.
01 commt pic x(80) value spaces.
Linkage Section.
01 peObj object reference PhoneEntry.
01 envPtr pointer.
Procedure Division using by value envPtr peObj.
 display spaces.
 display '*****'
 display '*** PHONE ENTRY ***'
 display '-----'
 invoke self 'show_entry'
 using by value peObj
 by reference lname fname pnumb addrs commt
 exit method.
End Method 'display_entry'.

End Class CobPhone.

```

Figure 10-20 (Part 2 of 2). COBOL subclass for Scenario 3.

**Step 7:** Compile the COBOL subclass, using the IDLGEN, DLL and EXPORTALL options obtaining the:

- object deck for the COBOL subclass
- IDL files for the COBOL subclass.

The COBOL compile may optionally specify the TYPECHK option - if specified, the COBOL subclass will be type checked to validate that the usage of the C++ classes is correct.

Figure 10-21 contains the JCL that compiles the COBOL subclass for Scenario 3.

```
//S3STEP7 JOB <JOB CARD PARAMETERS>
//PROCLIB JCLLIB ORDER=IGY.V1R3M0.SIGYPROC
//-----
//SETSON SET SOMPRF='somprfx'
//SETLE SET LEPRF='CEE'
//SETSRC SET SRCPRF='userprfx.COBOL'
//SETOBJ SET OBJPRF='userprfx.OBJ'
//SETIDL SET IDLPRF='userprfx.IDL'
//SETMYP SET MYPRF='userprfx.SOMPROF'
//-----
//COMPCLS EXEC IGYWC,REGION=56M,
// PARM.COBOL='RENT,PGMNAME(LM),DLL,IDLGEN,EXPORTALL'
//COBOL.SYSIN DD DSN=&SRCPRF.(COBPHONE),DISP=SHR
//COBOL.SYSLIN DD DSN=&OBJPRF.(COBPHONE),DISP=SHR
//COBOL.STEPLIB DD
// DD DSN=&SOMPRF..SGOSLOAD,DISP=SHR
// DD DSN=&LEPRF..SCEERUN,DISP=SHR
//COBOL.SOMENV DD DSN=&MYPRF.(mysomprf),DISP=SHR
//COBOL.SYSDL DD DSN=&IDLPRF.(COBPHONE),DISP=SHR
```

Figure 10-21. JCL to compile the COBOL subclass for Scenario 3.

**Step 8:** Create the COBOL subclass DLL (Cobphone). This DLL is comprised of the COBOL subclass object deck and the C++ classes object decks. Figure 10-22 on page 10-38 contains the JCL that builds the "COBPHONE" DLL for Scenario 3.

```

//S3STEP8 JOB <JOB CARD PARAMETERS>
//PROCLIB JCLLIB ORDER=(IGV.V1R3M0.SIGYPROC)
//-----
//SETSOM1 SET SOMPRF='somprfx'
//SETCC SET CCPRF='CBC, '
//SETLE SET LEPRF='CEE'
//SETSRC SET SRCPRF='userprfx.COBOL'
//SETOBJ SET OBJPRF='userprfx.OBJ'
//SETIMP SET IMPPRF='userprfx.SYSDEFSD'
//SETLOAD SET LOADPRF='userprfx.LOAD'
//SETSOM2 SET MYSOMPRF='userprfx.SOMPROF'
//-----
//MAKEDLL EXEC IGYWPL,REGION=56M,
// PARM LKED='LIST,XREF,LET,MAP'
//PLKED.STEPLIB DD DSN=&LEPRF..SCEERUN,DISP=SHR
//PLKED.SYSLIB DD DSN=&LEPRF..SCEECPP,DISP=SHR
// DD DSN=&CCPRF..SCLBCPP,DISP=SHR
//SYSIN DD DATA,DLM='/>
INCLUDE OBJECT(COBPHONE)
INCLUDE OBJECT(PBOOK)
INCLUDE OBJECT(PENTRY)
INCLUDE OBJECT(MYSTRING)
INCLUDE IMPORT(GOSSOMK)
NAME COBPHONE(R)
/>
//SYSOUT DD SYSOUT=*
//SYSDEFSD DD DSN=&IMPPRF.(COBPHONE),DISP=SHR
//OBJECT DD DSN=&OBJPRF.,DISP=SHR
//IMPORT DD DSN=&IMPPRF.,DISP=SHR
// DD DSN=&SOMPRF..SGOSIMP,DISP=SHR
//LKED.SYSLMOD DD DSN=&LOADPRF.(COBPHONE),DISP=SHR
/*

```

Figure 10-22. JCL to build the COBOL subclass DLL (Cobphone) which includes the C++ classes object decks for Scenario 3.

**Step 9:** SOM compile the COBOL-generated IDL file with the *ir* emitter (e.g. sc with the *-usir* option) to create an IR entry for the COBOL subclass.

**Note:** The record length requirement for COBOL generated IDL is <= 255. To avoid record length errors, it is good practice to allocate the COBOL IDL dataset in a different PDS than the C++ generated IDL dataset. Be sure to include any required IDL files on the SMINCLUDE include environment variable. Figure 10-23 on page 10-39 contains the JCL that SOM compiles the COBOL-generated IDL file with the *usir* option for Scenario 3.

```

//S3STEP9 JOB <JOB CARD PARAMETERS>
//-----*
//SC PROC INDSN='',
// MEM='',
// SCPARMS='',
// SOMPRF='somprfx',
// LEPRF='CEE'
//SOMC EXEC PGM=SC,REGION=80M,
// PARM=('&SCPARMS ''&INDSN.(&MEM.)''')
//STEPLIB DD DSN=&LEPRF..SCEERUN,DISP=SHR
// DD DSN=&SOMPRF..SGOSLOAD,DISP=SHR
//SOMENV DD DUMMY
//SYSPRINT DD SYSOUT=*
// PEND
//-----*
//SCCOB EXEC SC,
// MEM='COBPHONE',
// INDSN='userprfx.IDL',
// SCPARMS='-usir'
//SOMENV DD DSN=userprfx.SOMENV(mysomprf),DISP=SHR
//

```

Figure 10-23. JCL to SOM compile the COBOL-generated IDL file for Scenario 3.

## Developing and Running the Client Program

### Create COBOL Executable

**Step 10:** Create (with editor) the COBOL main program.

Figure 10-24 on page 10-40 contains the COBOL main program for Scenario 3.

```

CBL APOST,NOCMPR2,PGMNAME(LONGMIXED)
Identification Division.
Program-Id. 'CobMain2'.
Environment Division.
Configuration Section.
Repository.
 Class CobPhone is 'CobPhone'
 Class PhoneEntry is 'phone_entry'.
Data Division.
Working-Storage Section.
01 peObj object reference PhoneEntry value null global.
01 pbObj object reference CobPhone value null global.
01 envPtr usage pointer value null global.
01 response-input.
 02 filler pic x value '6'.
 02 filler pic x value '2'.
 02 filler pic x value '4'.
 02 filler pic x value '6'.
 02 filler pic x value '4'.
 02 filler pic x value '6'.
 02 filler pic x value '1'.
 02 filler pic x value '5'.
 02 filler pic x value '3'.
 02 filler pic x value '6'.
 02 filler pic x value '2'.
 02 filler pic x value 'Q'.
01 response-table redefines response-input.
 02 response-entry occurs 12 times.
 03 response-item pic x.
01 ndx pic 99 value 1.
01 response pic x.
 88 one value '1'.
 88 two value '2'.
 88 three value '3'.
 88 four value '4'.
 88 five value '5'.
 88 six value '6'.
 88 quit value 'Q'.
01 add-parameters.
 02 lname pic x(16) value z'Cat'.
 02 fname pic x(16) value z'Sylvester'.
 02 pnumb pic x(12) value z'1800WDisney'.
 02 addrs pic x(80)
 value z'Disney World, Orlando, FL 12345-6789'.
 02 commt pic x(80)
 value z'Large tuxedo-tabby cat who loves to chase
 'small yellow tweety birds.'.
01 search-parameters.
 02 scode pic x(1) value X'02'.
 02 s1str pic x(80) value z'Cat'.
 02 s2str pic x(80) value z'Sylvester'.
Linkage Section.
01 Environment-structure.
 02 major pic 9(9) binary.
 88 NO-EXCEPTION value 0.
 88 USER-EXCEPTION value 1.
 88 SYSTEM-EXCEPTION value 2.
 02 pic x(12).
Procedure Division.
 call 'somGetGlobalEnvironment' returning envPtr.
 set address of environment-structure to envPtr.
 invoke CobPhone 'somNew' returning pbObj.

 perform Get-Response.

```

Figure 10-24 (Part 1 of 3). COBOL main program for Scenario 3.

```

perform until quit
 evaluate true
 when one
 call 'CobSearch2' using by content scode
 s1str
 s2str
 if peObj = null
 display 'Phone entry not found.'
 else
 invoke pbObj 'display_entry'
 using by value envPtr peObj
 end-if
 when two
 invoke pbObj 'next'
 using by value envPtr
 returning peObj
 invoke pbObj 'display_entry'
 using by value envPtr peObj
 when three
 invoke pbObj 'prev'
 using by value envPtr
 returning peObj
 invoke pbObj 'display_entry'
 using by value envPtr peObj
 when four
 call 'CobAdd2' using by content lname
 fname
 pnumb
 addrs
 commt
 invoke pbObj 'display_entry'
 using by value envPtr peObj
 when five
 invoke pbObj 'erase'
 using by value envPtr
 returning peObj
 when six
 invoke pbObj 'show_all'
 end-evaluate

 set peObj to null
 perform Get-Response
end-perform.

stop run.

Get-Response.
 move response-item (ndx) to response.
 add 1 to ndx.

```

Figure 10-24 (Part 2 of 3). COBOL main program for Scenario 3.

```

Identification Division.
Program-Id. 'CobAdd2'.
Data Division.
Linkage Section.
01 lname pic x(16).
01 fname pic x(16).
01 pnumb pic x(12).
01 addrs pic x(80).
01 commt pic x(80).
Procedure Division using lname fname pnumb addrs commt.
 invoke pbObj 'add'
 using by value envPtr
 address of lname
 address of fname
 address of pnumb
 address of addrs
 address of commt
 returning peObj.
 exit program.
End Program 'CobAdd2'.

Identification Division.
Program-Id. 'CobSearch2'.
Data Division.
Linkage Section.
01 scode pic x(1).
01 s1str pic x(80).
01 s2str pic x(80).
Procedure Division using scode s1str s2str.
 invoke pbObj 'search'
 using by value envPtr
 scode
 address of s1str
 address of s2str
 returning peObj.
 exit program.
End Program 'CobSearch2'.

End Program 'CobMain2'.

```

Figure 10-24 (Part 3 of 3). COBOL main program for Scenario 3.

**Step 11:** Compile the COBOL main program.

The COBOL compile may optionally specify the TYPECHK option - if specified, the COBOL program will be type checked against the interfaces of the classes that are in the interface repository, to validate that the usage of the classes is correct. The JCL for this compile step is combined with the JCL in Step 12 and is found in Figure 10-25 on page 10-43.

**Step 12:** Prelink and link the object decks for the COBOL main program with the COBOL subclass definition side deck (Cobphone definition side deck was created in the DLL build step in Step 8 above). Figure 10-25 on page 10-43 contains the JCL to do both steps 11 and 12 for Scenario 3.

```

//S3STEP11 JOB <JOB CARD PARAMETERS>
//PROCLIB JCLLIB ORDER=(IGY.V1R3M0.SIGYPROC)
/*
//SETSOM1 SET SOMPRF='somprfx'
//SETCC SET CCPRF='CBC,'
//SETLE SET LEPRF='CEE'
//SETSRC SET SRCPRF='userprfx.COBOL'
//SETOBJ SET OBJPRF='userprfx.OBJ'
//SETIMP SET IMPPRF='userprfx.SYSDEFSD'
//SETLOAD SET LOADPRF='userprfx.LOAD'
//SETSOM2 SET MYSOMPRF='userprfx.SOMPROF'
/*
//RUNJOB EXEC IGYWCPLG,REGION=56M,GOPGM=COBMAIN2,
// PARM.COBOL='RENT,PGMNAME(LM),TEST,TYPECHK',
// PARM.LKED='LIST,XREF,LET,MAP'
//COBOL.SYSIN DD DSN=&SRCPRF.(COBMAIN2),DISP=SHR
//COBOL.SYSLIN DD DSN=&OBJPRF.(COBMAIN2),DISP=SHR
//COBOL.STEPLIB DD
// DD DSN=&SOMPRF..SGOSLOAD,DISP=SHR
// DD DSN=&LEPRF..SCEERUN,DISP=SHR
//COBOL.SOMENV DD DSN=&MYSOMPRF.(mysomprf),DISP=SHR
//PLKED.STEPLIB DD DSN=&LEPRF..SCEERUN,DISP=SHR
//PLKED.SYSLIB DD DSN=&LEPRF..SCEECPP,DISP=SHR
// DD DSN=&CCPRF..SCLBCPP,DISP=SHR
//SYSIN DD DATA,DLM='/>'
INCLUDE OBJECT(COBMAIN2)
INCLUDE IMPORT(COBPHONE)
INCLUDE IMPORT(GOSSOMK)
NAME COBMAIN2(R)
*/
/*
//SYSOUT DD SYSOUT=*
//SYSDEFSD DD DSN=&IMPPRF.(COBMAIN2),DISP=SHR
//IMPORT DD DSN=&IMPPRF.,DISP=SHR
// DD DSN=&SOMPRF..SGOSIMP,DISP=SHR
//OBJECT DD DSN=&OBJPRF.,DISP=SHR
//LKED.SYSLMOD DD DSN=&LOADPRF.(COBMAIN2),DISP=SHR
//GO.SYSOUT DD SYSOUT=H
//GO.STEPLIB DD DSN=&LEPRF..SCEERUN,DISP=SHR
// DD DSN=&SOMPRF..SGOSLOAD,DISP=SHR
// DD DSN=&LOADPRF.,DISP=SHR
//GO.SOMENV DD DSN=&MYSOMPRF.,DISP=SHR
/*

```

*Figure 10-25. JCL to compile, prelink and link the COBOL main program with the Cobphone definition side deck and run job for Scenario 3.*

The output of the RUNJOB step for Scenario 3 is in Figure 10-26 on page 10-44.

1

\*\*\*\*\*  
\*\*\* PHONE BOOK \*\*\*  
-----

Name: TWEETIE BIRD  
Phone: 1115551234  
Address: DISNEY WORLD, CALIFORNIA 99887-6655  
Comment: SYLVESTER'S FAVOURITE FOOD.

\*\*\*\*\*  
\*\*\* PHONE ENTRY \*\*\*  
-----

Name: TWEETIE BIRD  
Phone: 1115551234  
Address: DISNEY WORLD, CALIFORNIA 99887-6655  
Comment: SYLVESTER'S FAVOURITE FOOD.

\*\*\*\*\*  
\*\*\* PHONE ENTRY \*\*\*  
-----

Name: SYLVESTER CAT  
Phone: 1800WDISNEY  
Address: DISNEY WORLD, ORLANDO, FL 12345-6789  
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

\*\*\*\*\*  
\*\*\* PHONE BOOK \*\*\*  
-----

Name: TWEETIE BIRD  
Phone: 1115551234  
Address: DISNEY WORLD, CALIFORNIA 99887-6655  
Comment: SYLVESTER'S FAVOURITE FOOD.

Name: SYLVESTER CAT  
Phone: 1800WDISNEY  
Address: DISNEY WORLD, ORLANDO, FL 12345-6789  
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

\*\*\*\*\*  
\*\*\* PHONE ENTRY \*\*\*  
-----

Name: SYLVESTER CAT  
Phone: 1800WDISNEY  
Address: DISNEY WORLD, ORLANDO, FL 12345-6789  
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

Figure 10-26 (Part 1 of 2). Output of the RUNJOB step for Scenario 3.

```

*** PHONE BOOK ***

Name: TWEETIE BIRD
Phone: 1115551234
Address: DISNEY WORLD, CALIFORNIA 99887-6655
Comment: SYLVESTER'S FAVOURITE FOOD.

Name: SYLVESTER CAT
Phone: 1800WDISNEY
Address: DISNEY WORLD, ORLANDO, FL 12345-6789
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

*** PHONE ENTRY ***

Name: SYLVESTER CAT
Phone: 1800WDISNEY
Address: DISNEY WORLD, ORLANDO, FL 12345-6789
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

*** PHONE ENTRY ***

Name: TWEETIE BIRD
Phone: 1115551234
Address: DISNEY WORLD, CALIFORNIA 99887-6655
Comment: SYLVESTER'S FAVOURITE FOOD.

*** PHONE BOOK ***

Name: TWEETIE BIRD
Phone: 1115551234
Address: DISNEY WORLD, CALIFORNIA 99887-6655
Comment: SYLVESTER'S FAVOURITE FOOD.

*** PHONE ENTRY ***

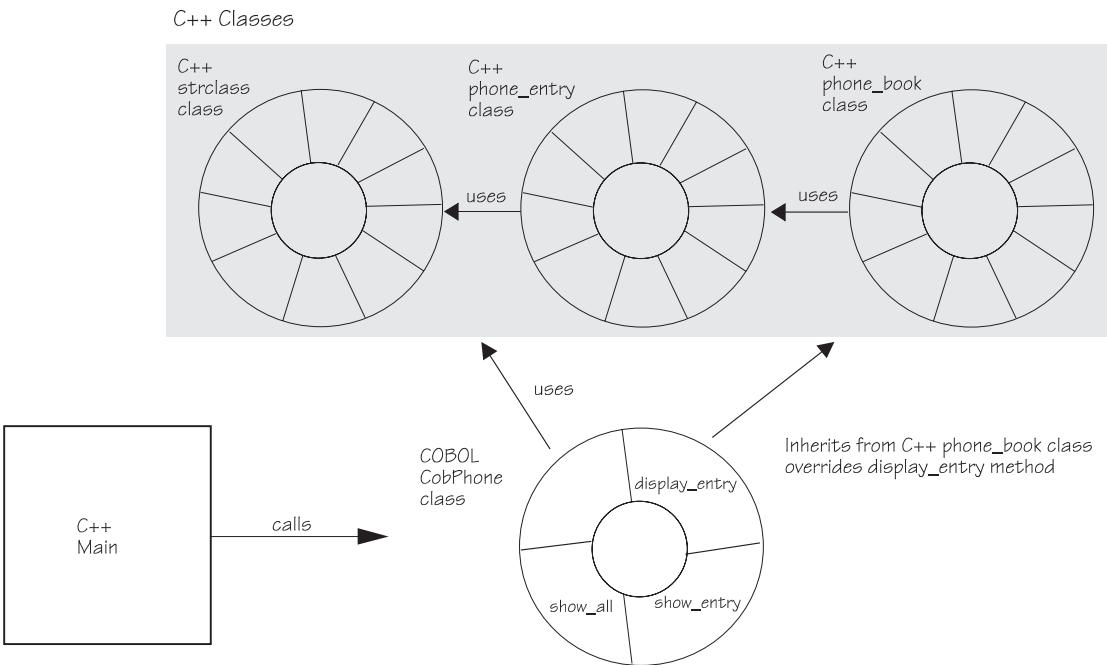
Name: TWEETIE BIRD
Phone: 1115551234
Address: DISNEY WORLD, CALIFORNIA 99887-6655
Comment: SYLVESTER'S FAVOURITE FOOD.
```

*Figure 10-26 (Part 2 of 2). Output of the RUNJOB step for Scenario 3.*

---

## Scenario 4: C++ Program Using COBOL Class that Inherits C++ Classes

The fourth scenario is a C++ main program using a COBOL class that inherits C++ classes and overrides a C++ class method.



## Building the Classes

To build the C++ and COBOL classes:

### Create the Header Files and IR Entries

**Step 1:** Create (with editor) the header files (.hh) for the C++ classes. (See Figure 10-4 on page 10-6, Figure 10-2 on page 10-4 and Figure 10-3 on page 10-5.)

**Step 2:** C++ compile the header files for the C++ classes, using the IDL option, obtaining the IDL files for the C++ classes. (See Figure 10-5 on page 10-7.)

**Step 3:** SOM compile the generated IDL files with the *ir* emitter (e.g. sc with the *-usir* option) to create IR entries for the classes. (See Figure 10-16 on page 10-28.)

### Create the C++ Implementation Files

**Step 4:** Create (with editor) the implementation files for the C++ classes, using appropriate #pragmas and options for DTS support. (See Figure 10-8 on page 10-15, Figure 10-6 on page 10-9, Figure 10-7 on page 10-13.)

**Step 5:** C++ compile the implementation files for the C++ classes, using the NOIDL option, obtaining the object decks for the C++ classes. (See Figure 10-9 on page 10-17.)

**Step 6:** Create (with editor) the COBOL subclass. Specify the C++ base class(es) in the INHERITS clause in the COBOL class definition. (See Figure 10-20 on page 10-35.)

**Step 7:** Compile the COBOL subclass, using the IDLGEN, DLL and EXPORTALL options obtaining the:

- object deck for the COBOL subclass
- IDL files for the COBOL subclass.

The COBOL compile may optionally specify the TYPECHK option - if specified, the COBOL subclass will be type checked to validate that the usage of the C++ classes is correct.

(See Figure 10-21 on page 10-37.)

**Step 8:** Create the COBOL subclass DLL (Cobphone). This DLL is comprised of the COBOL subclass object deck and the C++ classes object decks. Figure 10-22 on page 10-38 contains the JCL that builds the "COBPHONE" DLL for Scenario 3.

**Step 9:** SOM compile the COBOL-generated IDL file with the *hh* emitter to create a DTS-C++ header file CobPhone.hh for the COBOL subclass. The following TSO line command uses the SOM compiler to generate the DTS-C++ header files Cobphone for the COBOL subclass in Scenario 4.

```
sc -V -shh -mnoqualifytypes 'userprfx.idl(strclass)'
```

**Note:** The -mnoqualifytypes modifier is required to prevent the use of C-scoped names in the *hh* emitter output.

**Step 10:** Create (with editor) the C++ main program. Use an #include statement for the CobPhone.hh header file in the C++ main-program source file. Figure 10-27 on page 10-48 contains the C++ main program for Scenario 4.

```

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include "cobphone.hh"

main()
{
 CobPhone *mine;
 phone_entry *output,*temp;
 mine = new CobPhone;
 char input[1],lname[16],fname[16],number[12],addr[80],com[80];
 int x;
 output = NULL;
 do {
 cout<< "\f MAIN MENU \n"
 << "1 - Search for an entry \n"
 << "2 - Go to the next entry \n"
 << "3 - Go to the previous entry \n"
 << "4 - Add an entry \n"
 << "5 - Delete an entry \n"
 << "6 - Print phone book \n"
 << "Q - Quit the program \n" << endl;
 cin.getline(input, 80, '\n');
 x = atoi(input);
 switch(x) {
 case 1:
 cout<< "\f SEARCH MENU \n"
 << "1. Search for a last name\n"
 << "2. Search for a first name\n"
 << "3. Search for a phone number\n"
 << "4. Search for an address\n"
 << "5. Search for a comment\n \n" << endl;
 cin.getline(input, 80, '\n');
 x = atoi(input);
 switch(x) {
 case 1:
 cout << "Enter a last name to be searched" << endl;
 cin.getline(lname, 80, '\n');
 temp = mine->search(find_last,lname);
 if(temp == NULL)
 cout << "\n*** NO OTHER ENTRY FOUND ***" << endl;
 else
 output = temp;
 mine->display_entry(output);
 break;
 case 2:
 cout << "Enter a first name to be searched" << endl;
 cin.getline(fname, 80, '\n');
 temp = mine->search(find_first,fname);
 if(temp == NULL)
 cout << "\n*** NO OTHER ENTRY FOUND ***" << endl;
 else
 output = temp;
 mine->display_entry(output);
 break;
 case 3:
 cout << "Enter a phone number to be searched" << endl;
 cin.getline(number, 80, '\n');
 temp = mine->search(find_phone,number);
 if(temp == NULL)
 cout << "\n*** NO OTHER ENTRY FOUND ***" << endl;
 else
 output = temp;
 mine->display_entry(output);
 break;
 }
 }
 }
}

```

Figure 10-27 (Part 1 of 2). C++ main program for Scenario 4.

```

 case 4:
 cout << "Enter a address to searched" << endl;
 cin.getline(addr, 80, '\n');
 temp = mine->search(find_addr,addr);
 if(temp == NULL)
 cout << "\n*** NO OTHER ENTRY FOUND ***" << endl;
 else
 output = temp;
 mine->display_entry(output);
 break;
 case 5:
 cout << "Enter a comment to searched" << endl;
 cin.getline(com, 80, '\n');
 temp = mine->search(find_com,com);
 if(temp == NULL)
 cout << "\n*** NO OTHER ENTRY FOUND ***" << endl;
 else
 output = temp;
 mine->display_entry(output);
 break;
 }
 break;
case 2:
 output = mine->next();
 mine->display_entry(output);
 break;
case 3:
 output = mine->prev();
 mine->display_entry(output);
 break;
case 4:
 cout << "\f"
 << "Enter the following entries in the given order\n"
 << "first name (maximum 15 characters)\n"
 << "last name (maximum 15 characters)\n"
 << "phone number (nnn-mmmm)\n"
 << "address (maximum 80 characters)\n"
 << "comment (maximum 80 characters)\n" << endl;
 cin.getline(fname, 80, '\n');
 cin.getline(lname, 80, '\n');
 cin.getline(number, 80, '\n');
 cin.getline(addr, 80, '\n');
 cin.getline(com, 80, '\n');
 output = mine->add(lname,fname,number,addr,com);
 mine->display_entry(output);
 break;
case 5:
 cout << "ERASING current entry!!\n" << endl;
 output = mine->erase();
 break;
case 6:
 mine->show_all();
 break;
}
} while(strcmp(input,"Q") && strcmp(input,"q"));
delete mine;
}

```

Figure 10-27 (Part 2 of 2). C++ main program for Scenario 4.

## Developing and Running the Client Program

## Create C++ Executable

**Step 11:** Compile the C++ main program to obtain the object deck for the main program. Figure 10-28 contains the JCL to compile the C++ program for Scenario 4.

```
//S4STP11 JOB <JOB CARD PARAMETERS>
//PROCLIB JCLLIB ORDER=CBC.SCBCPRC
//*-----
//COMP EXEC CBCC,
// INFILE='userprfx.CPP(CPPMAIN3)',
// OUTFILE='userprfx.OBJ(CPPMAIN3),DISP=SHR',
// LNGPRFC='CBC',
// LIBPRFX='CEE',
// CREGSIZ='32M',
// CPARM='OPTFILE(DD:OPTS)'
//STEPLIB DD DSN=&LNGPRFC..SCBCCMP,DISP=SHR
// DD DSN=&LIBPRFX..SCEERUN,DISP=SHR
//COMPILE.OPTS DD *
// SEARCH('userprfx.+',
// 'CEE.SCEEH.+',
// 'CEE.SCEEH.SYS.+',
// 'CBC.SCLBH.+',
// 'CBC.SCLBSID.+',
// 'somprfx.SGOSHH.+',
// 'somprfx.SGOSH.+')
//*
```

Figure 10-28. JCL to compile the C++ main program for Scenario 4.

**Step 12:** Prelink and link the object decks for the C++ main program with the COBOL subclass definition side deck (Cobphone definition side deck was created in the DLL build step in Step 8 above) and run the program. Figure 10-29 on page 10-51 contains the JCL to do step 12 for Scenario 4.

main program object deck with the Cobphone definition side deck and run the job for Scenario 4.

```
//S4STP12 JOB <JOB CARD PARAMETERS>
//PROCLIB JCLLIB ORDER=(IGY.V1R3M0.SIGYPROC)
//-----
//SETSM1 SET SOMPRF='somprfx'
//SETCC SET CCPRF='CBC'
//SETLE SET LEPRF='CEE'
//SETSRC SET SRCPRF='userprfx.COBOL'
//SETOBJ SET OBJPRF='userprfx.OBJ'
//SETIMP SET IMPPRF='userprfx.SYSDEFSD'
//SETLOAD SET LOADPRF='userprfx.LOAD'
//SETSM2 SET MYSOMPRF='userprfx.SOMPROF'
//-----
//LINKJOB EXEC IGYWPL,REGION=56M,
// PARM.LKED='LIST,XREF,LET,MAP'
//PLKED.STEPLIB DD DSN=&LEPRF..SCEERUN,DISP=SHR
//PLKED.SYSLIB DD DSN=&LEPRF..SCEECPP,DISP=SHR
// DD DSN=&CCPRF..SCLBCPP,DISP=SHR
//SYSIN DD DATA,DLM='/>
INCLUDE OBJECT(CPPMAIN3)
INCLUDE IMPORT(COBPHONE)
INCLUDE IMPORT(GOSSOMK)
NAME CPPMAIN3(R)
/>
//*
//SYSOUT DD SYSOUT=*
//SYSDEFSD DD DSN=&IMPPRF.(CPPMAIN3),DISP=SHR
//IMPORT DD DSN=&IMPPRF.,DISP=SHR
// DD DSN=&SOMPRF..SGOSIMP,DISP=SHR
//OBJECT DD DSN=&OBJPRF.,DISP=SHR
//LKED.SYSLMOD DD DSN=&LOADPRF.(CPPMAIN3),DISP=SHR
//*****
//GO EXEC PGM=CPPMAIN3,REGION=4M
//STEPLIB DD DSN=&LEPRF..SCEERUN,DISP=SHR
// DD DSN=&SOMPRF..SGOSLOAD,DISP=SHR
// DD DSN=&LOADPRF.,DISP=SHR
//SOMENV DD DSN=&MYSOMPRF.,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSIN DD *
6
4
Sylvester
Cat
800WDisney
Disney World, Orlando, FL 12345-6789
Large tuxedo-tabby cat who loves to chase small yellow tweety birds.
6
4
Tweetie
Bird
1115551234
Disney World, California 99887-6655
Sylvester's favourite food.
6
3
3
2
1
2
Bird
1
3
```

Figure 10-29 (Part 1 of 2). JCL to prelink and link the C++

```
800WDIsney
5
6
2
2
0
//
```

*Figure 10-29 (Part 2 of 2). JCL to prelink and link the C++*

The output of the RUNJOB step for Scenario 4 is in Figure 10-30 on page 10-53.

1

\*\*\*\*\*  
\*\*\* PHONE BOOK \*\*\*

-----  
Name: TWEETIE BIRD  
Phone: 1115551234  
Address: DISNEY WORLD, CALIFORNIA 99887-6655  
Comment: SYLVESTER'S FAVOURITE FOOD.

\*\*\*\*\*  
\*\*\* PHONE ENTRY \*\*\*

-----  
Name: SYLVESTER CAT  
Phone: 800WDISNEY  
Address: DISNEY WORLD, ORLANDO, FL 12345-6789  
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

\*\*\*\*\*  
\*\*\* PHONE BOOK \*\*\*

-----  
Name: TWEETIE BIRD  
Phone: 1115551234  
Address: DISNEY WORLD, CALIFORNIA 99887-6655  
Comment: SYLVESTER'S FAVOURITE FOOD.

Name: SYLVESTER CAT  
Phone: 800WDISNEY  
Address: DISNEY WORLD, ORLANDO, FL 12345-6789  
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

\*\*\*\*\*  
\*\*\* PHONE ENTRY \*\*\*

-----  
Name: TWEETIE BIRD  
Phone: 1115551234  
Address: DISNEY WORLD, CALIFORNIA 99887-6655  
Comment: SYLVESTER'S FAVOURITE FOOD.

\*\*\*\*\*  
\*\*\* PHONE BOOK \*\*\*

-----  
Name: TWEETIE BIRD  
Phone: 1115551234  
Address: DISNEY WORLD, CALIFORNIA 99887-6655  
Comment: SYLVESTER'S FAVOURITE FOOD.

Name: SYLVESTER CAT  
Phone: 800WDISNEY  
Address: DISNEY WORLD, ORLANDO, FL 12345-6789  
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

\*\*\*\*\*  
\*\*\* PHONE ENTRY \*\*\*

-----  
Name: TWEETIE BIRD  
Phone: 1115551234  
Address: DISNEY WORLD, CALIFORNIA 99887-6655  
Comment: SYLVESTER'S FAVOURITE FOOD.

Figure 10-30 (Part 1 of 2). Output of the RUNJOB step for Scenario 4.

```

*** PHONE ENTRY ***

Name: SYLVESTER CAT
Phone: 800DISNEY
Address: DISNEY WORLD, ORLANDO, FL 12345-6789
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

*** PHONE ENTRY ***

Name: TWEETIE BIRD
Phone: 111551234
Address: DISNEY WORLD, CALIFORNIA 99887-6655
Comment: SYLVESTER'S FAVOURITE FOOD.

*** PHONE ENTRY ***

Name: TWEETIE BIRD
Phone: 111551234
Address: DISNEY WORLD, CALIFORNIA 99887-6655
Comment: SYLVESTER'S FAVOURITE FOOD.

*** PHONE ENTRY ***

Name: SYLVESTER CAT
Phone: 800DISNEY
Address: DISNEY WORLD, ORLANDO, FL 12345-6789
Comment: LARGE TUXEDO-TABBY CAT WHO LOVES TO CHASE SMALL YELLOW TWEETY BIRDS.

*** PHONE BOOK ***

Name: TWEETIE BIRD
Phone: 111551234
Address: DISNEY WORLD, CALIFORNIA 99887-6655
Comment: SYLVESTER'S FAVOURITE FOOD.

*** PHONE ENTRY ***

Name: TWEETIE BIRD
Phone: 111551234
Address: DISNEY WORLD, CALIFORNIA 99887-6655
Comment: SYLVESTER'S FAVOURITE FOOD.

*** PHONE ENTRY ***

Name: TWEETIE BIRD
Phone: 111551234
Address: DISNEY WORLD, CALIFORNIA 99887-6655
Comment: SYLVESTER'S FAVOURITE FOOD.
```

*Figure 10-30 (Part 2 of 2). Output of the RUNJOB step for Scenario 4.*

The previous sections in this chapter discussed examples of SOMobjects, including its language neutrality. The rest of this book will discuss more advanced topics of both SOMobjects and distributed SOMobjects, plus the SOMobjects frameworks.

---

## Chapter 11. SOMobjects Advanced Topics

This chapter discusses some of the more advanced topics of SOMobjects. Once you understand the basic concepts of SOMobjects and have tried some of the examples, you may be interested in the following topics:

- “Understanding Classes and Hierarchy”
- “The Four Kinds of SOMobjects Methods” on page 11-6
- “Initializing and Uninitializing Objects” on page 11-8
- “C/C++ Methods and Functions” on page 11-25
- “Customizing SOM Runtime Features” on page 11-28

---

### Understanding Classes and Hierarchy

The SOMobjects metaclasses have slightly different relationships than regular classes, especially with regards to hierarchy. This section includes the following topics:

- “Parent Class vs. Metaclass”
- “SOM-derived Metaclasses” on page 11-3

### Parent Class vs. Metaclass

There is a distinct difference between the notions of “parent” (or base) class and “metaclass.” Both notions are related to the fact that a class defines the methods and variables of its instances, which are therefore called *instance methods* and *instance variables*.

One class may be derived from another class. That is, a child class is a subclass of a parent class. In such cases, the child class inherits the instance methods and instance variables from the parent class. For example, the child class “Dog” inherits instance methods and instance variables from its parent class “Animal”, such as methods for breathing and eating, and a variable for storing its weight. Since “Dog” inherits these from “Animal”, any given dog instance could breathe, eat and store its weight.

A *metaclass* is a class whose instances are class objects, and whose instance methods and instance variables (as described above) are therefore the methods and variables of class objects. For this reason, a metaclass is said to define class methods—the methods that a class object performs. For example, the metaclass of “Animal” might be “AnimalMClass”, which defines the methods that can be invoked on class “Animal” (such as, to create Animal instances—objects that are not classes, like an individual pig or cat or elephant or dog).

Note: It is important to distinguish the methods of a class object (that is, the methods that can be invoked on the class object, which are defined by its metaclass) from the methods that the class defines for its instances.

To summarize: the parent of a class provides inherited methods that the class's instances can perform; the metaclass of a class provides class methods that the class itself can perform. The distinctions between parent class and metaclass are summarized in Figure 11-1 on page 11-2.

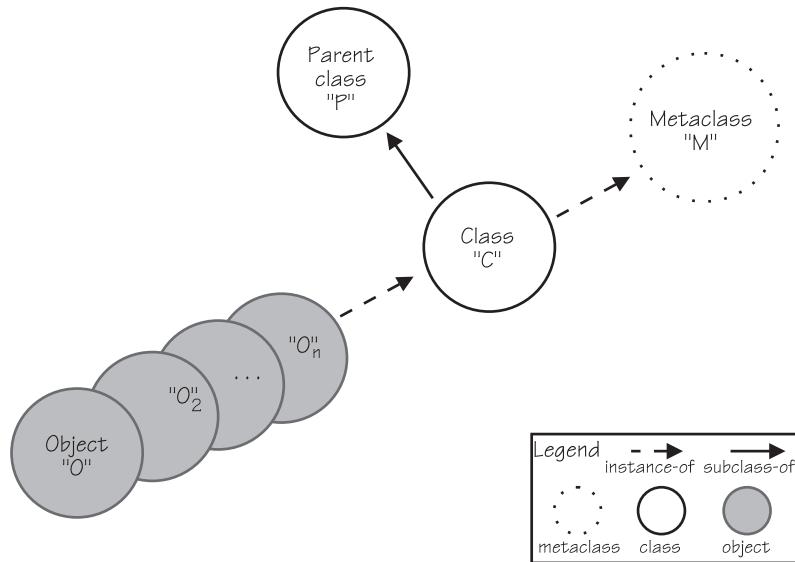


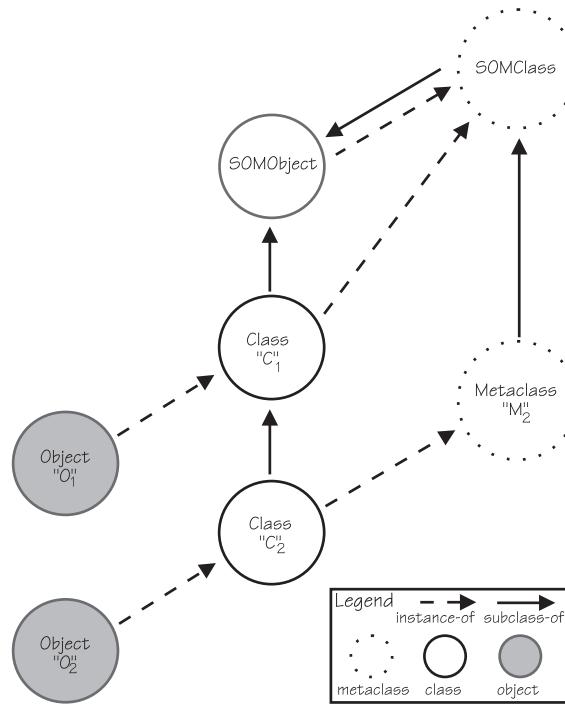
Figure 11-1. A class has both parent classes and a metaclass

In Figure 11-1, any class "C" has both a metaclass and one or more parent class(es):

- The parent class(es) of "C" provide the inherited instance methods that individual instances (objects " $O_n$ ") of class "C" can perform. Instance methods that an instance " $O_n$ " performs might include (a) initializing itself, (b) performing computations using its instance variables, (c) printing its instance variables, or (d) returning its size. New instance methods are defined by "C" itself, in addition to those inherited from C's parent classes.
- The metaclass "M" defines the class methods that class "C" can perform. For example, class methods defined by metaclass "M" include those that allow "C" to (a) inherit its parents' instance methods and instance variables, (b) tell its own name, (c) create new instances, and (d) tell how many instance methods it supports. These methods are inherited from SOMClass. Additional methods supported by "M" might allow "C" to count how many instances it creates.
- Each class "C" has one or more parent classes and exactly one metaclass. (The single exception is SOMObject, which has no parent class.) Parent class(es) must be explicitly identified in the IDL declaration of a class. (SOMObject is given as a parent if no subsequently-derived class applies.) If a metaclass is not explicitly listed, the SOMobjects runtime 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 11-2 on page 11-3, a sequence of classes is defined (or derived), stemming from SOMObject. The child class (or subclass) at the end of this line (" $C_2$ ") inherits instance methods from all of its ancestor classes (here, SOMObject and " $C_1$ "). An instance created by " $C_2$ " can perform any of these instance methods. In an analogous manner, a line of metaclasses is defined, stemming from SOMClass. Just as a new class is derived from an existing class (such as SOMObject), a new metaclass is derived from an

existing metaclass (such as SOMClass). In this example, both SOMObject and class “C<sub>1</sub>” are instances of the SOMClass metaclass, whereas class “C<sub>2</sub>” is an instance of metaclass “M<sub>2</sub>”, which inherits from SOMClass.



*Figure 11-2. Parent classes and metaclasses each have their own independent inheritance hierarchies*

## SOM-derived Metaclasses

As previously discussed, a class object can perform any of the class methods that its metaclass defines. New metaclasses are typically created to modify existing class methods or introduce new class method(s).

Three factors are essential for effective use of metaclasses in SOMobjects:

- First, every class in SOMobjects is an object that is implemented by a metaclass.
- Second, you can define and name new metaclasses, and can use these metaclasses when defining new SOMobjects classes.
- Finally, and most importantly, metaclasses cannot interfere with the fundamental guarantee required of every OOP system: specifically, any code that executes without method-resolution error on instances of a given class will also execute without method-resolution errors on instances of any subclass of this class.

SOMobjects can make this guarantee while also allowing you to explicitly define and use named metaclasses. This is possible because SOMobjects automatically determines an appropriate metaclass that supports this guarantee, automatically deriving new metaclasses by subclassing at run time when this is necessary.

## Understanding Metaclass Incompatibility

To understand metaclass incompatibility, consider the situation in Figure 11-3. Here, class “A” is an instance of metaclass “AMeta”. Assume that “AMeta” supports a method “bar” and that “A” supports a method “foo” that uses the expression “\_bar( \_somGetClass(somSelf) ).” That is, method “foo” invokes “bar” on the class of the object on which “foo” is invoked. For example, when method “foo” is invoked on an instance of class “A” (say, object “O<sub>1</sub>”), this in turn invokes “bar” on class “A” itself.

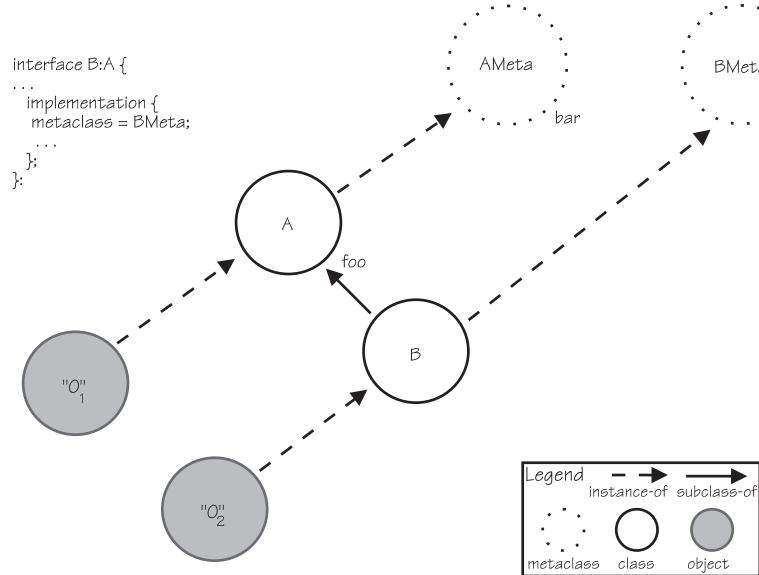


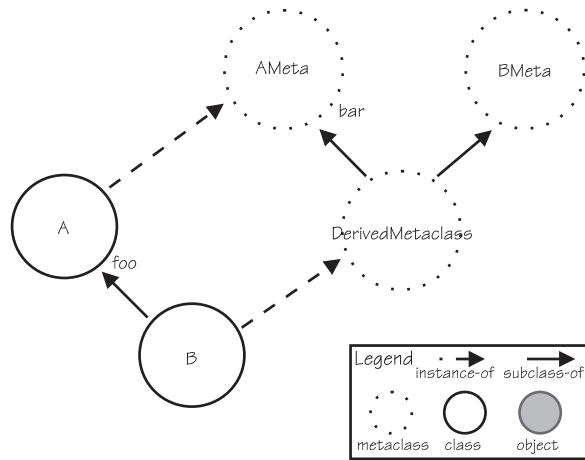
Figure 11-3. Example of metaclass incompatibility

Now consider what happens when “A” is subclassed by “B”, a class that has the explicit metaclass “BMeta” declared in its SOMobjects IDL source data set, as shown in Figure 11-3. If the class hierarchy were formed as in Figure 11-3, then an invocation of “foo” on “O<sub>2</sub>” would fail, because metaclass “BMeta” does not support the “bar” method introduced by “AMeta”.

There is only one way that “BMeta” can support this specific method—by inheriting it from “AMeta” (“BMeta” could introduce another method named “bar”, but this would be a different method from the one introduced by “AMeta”). Therefore, in this example, because “BMeta” is not a subclass of “AMeta”, “BMeta” cannot be allowed to be the metaclass of “B”. That is, “BMeta” is not compatible with the requirements placed on “B” by the fundamental principle of OOP referred to above. This situation is referred to as *metaclass incompatibility*.

## SOMobjects Derived Metaclasses

SOMobjects does not allow hierarchies with metaclass incompatibilities. Instead, SOMobjects automatically builds *derived metaclasses* when this is necessary. The resulting class hierarchy in this example is depicted in Figure 11-4 on page 11-5, where SOMobjects 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”.

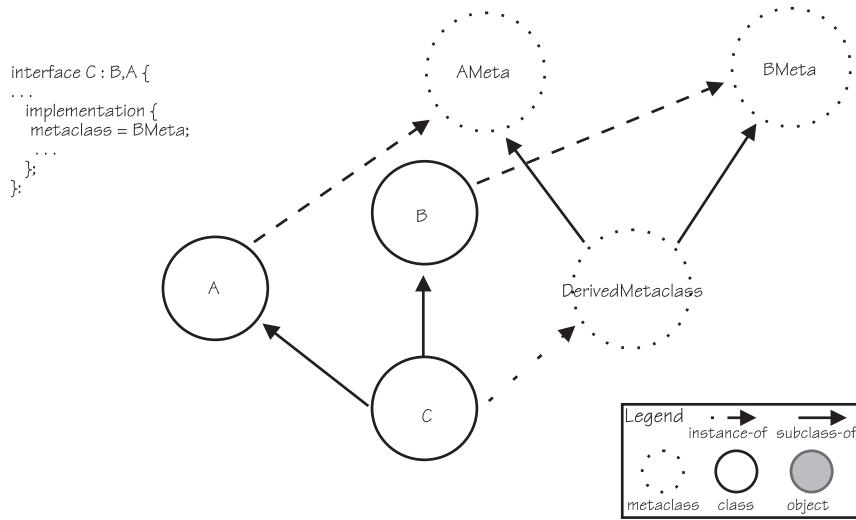


*Figure 11-4. Example of a derived metaclass*

There are three important aspects of SOMobjects's approach to derived metaclasses:

- First, the creation of SOM-derived metaclasses is integrated with programmer-specified metaclasses. If a programmer-specified metaclass already supports all the class methods and variables needed by a new class, then the programmer-specified metaclass will be used as is.
- Second, if SOMobjects must derive a different metaclass than the one explicitly indicated by the programmer (in order to support all the necessary class methods and variables), then the SOM-derived metaclass inherits from the explicitly indicated metaclass first. As a result, the method procedures defined by the specified metaclass take precedence over other possibilities (see the following section on inheritance and the discussion of resolution of ambiguity in the case of multiple inheritance).
- Finally, the class methods defined by the derived metaclass invoke the appropriate initialization methods of its parents to ensure that the class variables of its instances are correctly initialized.

As further explanation for the automatic derivation of metaclasses, consider the following multiple-inheritance example. In Figure 11-5 on page 11-6, class “C” does not have an explicit metaclass declaration in its SOMobjects 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 SOMobjects did not build the derived metaclass, you would have to do so yourself.)



*Figure 11-5. Multiple inheritance requires derived metaclasses*

In summary, SOMobjects allows and encourages the definition and explicit use of named metaclasses. With named metaclasses, you can not only affect the behavior of class instances by choosing the parents of classes, but you can also affect the behavior of the classes themselves by choosing their metaclasses. Because the behavior of classes in SOMobjects includes the implementation of inheritance itself, metaclasses in SOMobjects 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, SOMobjects relieves you 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 SOMobjects is predicated on binary compatibility with respect to changes in class implementations.

You might, at one point in time, know the metaclasses of all ancestor classes of a new subclass, and, as a result, be able to explicitly derive an appropriate metaclass for the new class. Nevertheless, SOMobjects must guarantee that this new class will still execute and perform correctly when any of its ancestor class's implementations are changed (which could even include specifying different metaclasses). Derived metaclasses allow SOMobjects to make this guarantee. You need never worry about the problem of metaclass incompatibility; SOMobjects does this for you. Instead, explicit metaclasses can simply be used to "add in" whatever behavior is desired for a new class. SOMobjects automatically handles anything else that is needed.

---

## The Four Kinds of SOMobjects Methods

SOMobjects supports four different kinds of methods: static methods, nonstatic methods, dynamic methods, and direct-call procedures. The following paragraphs explain these four method categories and the kinds of method resolution available for each.

## Static Methods

These are similar in concept to C++ virtual functions. Static methods are normally invoked using offset resolution via a method table, as described in “Method Resolution Advanced Topics” on page 11-30, 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 language bindings are used to implement a class, the 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 SOMobjects methods.

Static methods introduced by a class can be overridden (redefined) by any descendant classes of the class. When language bindings are used to implement a class, the IDL **override** modifier is specified to indicate that a class overrides a given inherited method. When a static method is resolved using offset resolution, it is not important which interface is accessing the method. The actual class of the object determines the method procedure that is selected.

**Note:** All SOM IDL modifiers are described in “Modifier Statements” on page 3-25.

## 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 language bindings are used to implement a class, the 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 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 the method **somAddDynamicMethod**. Because there is no way for SOMobjects to know about dynamic methods before run time, offset resolution is not available for dynamic methods. Only name-lookup or dispatch-function resolution can be used to invoke dynamic methods. Dynamic methods cannot be overridden.

## Direct-call Procedures

These are similar in concept to C++ static member functions. Direct-call procedures are not given positions in SOMobjects method tables, but are accessed directly from a class’s **ClassData** structure. Strictly speaking, none of the previous method-resolution approaches apply for invoking a direct-call procedure, although 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 language bindings are used to implement a class, the IDL **procedure** modifier is used to indicate that a given method is a direct-call procedure.

---

## Initializing and Uninitializing Objects

This section discusses the initialization and uninitialization of objects. Subsequent topics introduce the methods and capabilities that SOMobjects provides to facilitate this.

Object creation is the act that enables the execution of methods on an object. In SOMobjects, this means storing a pointer to a method table into a word of memory. This single act converts raw memory into an (uninitialized) 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 SOMobjects. Initialization is a capability supported by certain methods available on an object. An object's class determines the implementation of the methods available on the object, and thus determines its initialization behavior.

The instance variables encapsulated by a newly created object must be brought into a consistent state before the object can be used. This is the purpose of initialization methods. Because, in general, every ancestor of an object's class contributes instance data to an object, it is appropriate that each of these ancestors contribute to the initialization of the object.

SOMobjects thus recognizes initializers as a special kind of method. One advantage of this approach is that special metaclasses are not required for defining 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.

SOMobjects provides functions and methods that class designers can easily exploit in order to implement default or customized initialization of objects. These functions and methods are fully supported by emitters that produce the implementation template file. The following sections describe the declaration, implementation, and use of initializer (and uninitializer) methods.

## Initializer Methods

As noted above, in SOMobjects 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 SOMobjects Compiler emitters. The SOMobjects framework for initialization of objects is described in the following topics:

- Declaring new initializers in IDL
- Considerations regarding 'somInit' initialization
- Implementing initializers
- Using initializers when creating new objects

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.

The SOMobjects Compiler's emitters provide special support for methods that are declared as initializers in the .idl file. To supplement the **somDefaultInit** method, class designers can also declare additional initializers in their own classes.

Two IDL modifiers are provided for declaring initializer methods and controlling their execution, **init** and **directinitclasses**:

- The **init** modifier is required in order to designate that a given method is an initializer; that is, to indicate that the method both uses and supports the object-initialization protocol described here.
- The **directinitclasses** modifier can be used to control the order of execution of initializer method procedures provided by the different ancestors of the class of an object.

For full definitions of **init** and **directinitclasses**, see “Modifier Statements” on page 3-25.

Every SOMobjects class has a list that defines (in sequential order) the ancestor classes whose initializer method procedures the class should invoke. If a class's IDL does not specify an explicit **directinitclasses** modifier, the default for this list is simply the class's parents in left-to-right order.

Using the **directinitclasses** list and the actual run-time class hierarchy above itself, each class inherits from **SOMClass** the ability to create a data structure of type **somInitCtrl**. This structure is used to control the execution of initializers. Moreover, it represents a particular visit-ordering that reaches each class in the transitive closure of **directinitclasses** exactly once. To initialize a given object, this visit-ordering occurs as follows: While recursively visiting each ancestor class whose initializer method procedure should be run, SOMobjects first runs the initializer method procedures of all of that class's **directinitclasses** if they have not already been run by another class's initializers, with ancestor classes always taken in left-to-right order.

For example, Figure 11-6 on page 11-10 shows an inheritance hierarchy along with the ordering produced when an instance of the class numbered **7** is initialized, assuming that each class simply uses its parents as its **directinitclasses**. Note that the class numbered **3** is at the top of a diamond.

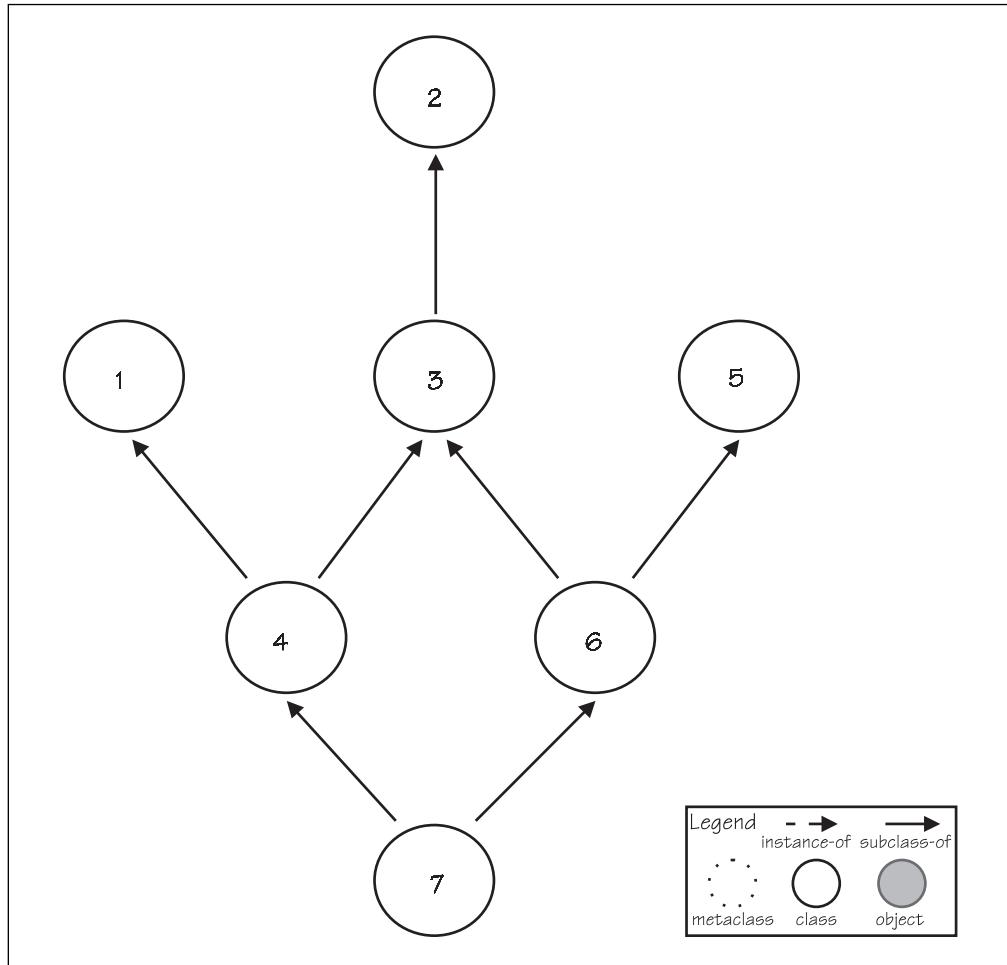


Figure 11-6. A default initializer ordering of a class's inheritance hierarchy.

In this example, the **somInitCtrl** data structure for class **7** is what tells node **6** in Figure 11-6 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 IDL

When defining SOMobjects classes, programmers can easily declare and implement new initializers. Classes can have as many initializers as desired, and subclasses can invoke whichever of these they want. When introducing new initializers, developers must adhere to the following rules:

- All initializer methods take a **somInitCtrl** data structure as an initial **inout** parameter (its type is defined in the header file `somapi.h`), and
- All initializers return **void**.

Accordingly, the **somDefaultInit** initializer introduced by **SOMObject** takes a **somInitCtrl** structure as its (only) argument, and returns **void**. Here is the IDL syntax for this method, as declared in somobj.idl:

```
void somDefaultInit (inout somInitCtrl ctrl);
```

When introducing a new initializer, it is also necessary to specify the **init** modifier in the **implementation** section. The **init** modifier is what tells emitters that the new method is actually an initializer, so the method can be properly supported from the language bindings. As described below, this support includes the generation of special initializer stub procedures in the implementation template file, as well as bindings containing ancestor-initialization macros and class methods that invoke the class implementor's new initializers.

It is a good idea to begin the names of initializer methods with the name of the class (or some other string that can be unique for the class). This is important because all initializers available on a class must be newly introduced by that class (that is, you cannot override initializers except for **somDefaultInit**). Using a class-unique name means that subclasses will not be unnecessarily constrained in their choice of initializer names.

Figure 11-7 shows 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;
 };
};
```

Figure 11-7. Two classes that introduce new initializers.

Here, interface "Example1" declares three new initializers. Notice the use of **inout somInitCtrl** as the first argument of each initializer, and also note that the **init** modifier is used in the **implementation** section. These two things are required to declare initializers. Any number of initializers can be declared by a class. "Example2" declares two initializers.

"Example1" and "Example2" both override the **somDefaultInit** initializer. This initializer method is introduced by **SOMObject** and is special for two reasons: First, **somDefaultInit** is the only initializer that can be overridden. And, second, SOMobjects arranges that this initializer will always be available on any class (as further explained below).

Object-initialization methods by default invoke the **somInit** method, which class implementers could override to customize initialization as appropriate. SOMobjects supports this approach, so that existing code (and class binaries) will execute correctly. However, the **somDefaultInit** method is now the preferred form of initialization because it offers greatly improved efficiency.

Even if no specialized initialization is needed for a class, you should still **override** the **somDefaultInit** method in the interest of efficiency. If you do not override **somDefaultInit**, then a generic (and therefore less efficient) **somDefaultInit** method procedure will be used for your class. This generic method procedure first invokes **somDefaultInit** on the appropriate ancestor classes. Then it checks to determine if the class overrides **somInit** and, if so, calls any customized **somInit** code provided by the class.

When you override **somDefaultInit**, the emitter's implementation template file will include a stub procedure similar to those used for other initializers, and you can fill it in as appropriate (or simply leave it as is). Default initialization for your class will then run much faster than with the generic method procedure. Examples of initializer stub procedures (and customizations) are given below.

In summary, the initializers available for any class of objects are **somDefaultInit** (which you should always **override**) plus any new initializers explicitly declared by the class designer. Thus, "Example1" objects may be initialized using any of four different initializers (the three that are explicitly declared, plus **somDefaultInit**). Likewise, there are three initializers for the "Example2" objects. Some examples of using initializers are provided below.

### Considerations re: 'somInit' Initialization

SOMobjects does not constrain initialization to be done in any particular way or require the use of any particular ordering of the method procedures of ancestor classes. SOMobjects does provide functions and methods that class designers can easily utilize in order to implement default initialization of objects. These functions and methods are provided by the **somInit** object-initialization method introduced by the **SOMObject** class and supported by the emitters. The emitters create an implementation template file with stub procedures for overridden methods that automatically chain parent-method calls upward through parent classes. Many of the class methods that perform object creation call **somInit** automatically.

Because it takes no arguments, **somInit** best serves the purpose of a default initializer. SOMobjects programmers also have the option of introducing additional "non-default" initialization methods that take arguments. In addition, by using metaclasses, you can introduce new class methods as class methods that first create an object (generally using **somNewNoInit**) and then invoke some non-default initializer on the new object.

For a number of reasons, **somInit** 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 class methods that take

arguments. Instead, because the **init** modifier identifies initializers, usage-binding emitters can now provide these class methods. This results in simpler designs and more efficient programs.

However, you cannot use both methods. In particular, if a class overrides both **somDefaultInit** and **somInit**, its **somInit** code will never be executed. It is recommended that you always override **somDefaultInit** for object initialization.

## Implementing Initializers

When new initializers are introduced by a class, as in the preceding examples, the implementation template file generated by the emitters automatically contains an appropriate stub procedure for each initializer method, for the class implementor's use. The body of an initializer stub procedure consists of two main sections:

- The first section performs calls to ancestors of the class to invoke their initializers.
- The second section is used by the programmer to perform any "local" initializations appropriate to the instance data of the class being defined.

In the first section, by default, the parents of the new class are the ancestors whose initializers are called. When something else is desired, the IDL **directinitclasses** modifier can be used to explicitly designate the ancestors whose initializer methods should be invoked by a new class's initializers.

**Important:** Under no circumstances can the number or the ordering of ancestor initializer calls in the first section of an initializer stub procedure be changed. The control masks used by initializers are based on these orderings. (If you want to change the number or ordering of ancestor initializer calls, you must use the **directinitclasses** modifier.) The ancestor initializer calls themselves can be modified as described below.

Each call to an ancestor initializer is made using a special macro (much like a parent call) that is defined for this purpose within the implementation bindings. These macros are defined for all possible ancestor initialization calls. Initially, an initializer stub procedure invokes the default ancestor initializers provided by **somDefaultInit**. However, a class implementor can replace any of these calls with a different initializer call, as long as it calls the same ancestor (see the example in "Selecting Non-default Ancestor Initializer Calls" on page 11-14). 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 in Figure 11-8 on page 11-14. 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
 */
}

```

*Figure 11-8. Implementing initializers.*

In this example, notice that the "Example2\_withName" initializer is an IDL callstyle method, so it receives an **Environment** argument. In contrast, **somDefaultInit** is introduced by the **SOMObject** class (so it has an OIDL callstyle initializer, without an environment).

**Important:** If a class is defined where multiple initializers have exactly the same signature, then the C++ usage bindings will not be able to differentiate among them. That is, if there are multiple initializers defined with environment and long arguments, for example, then C++ clients would not be able to make a call using only the class name and arguments, such as:

```
new Example2(env, 123);
```

Rather, C++ users would be forced to first invoke the **somNewNoInit** method on the class to create an uninitialized object, and then separately invoke the desired initializer method on the object. This call would pass a zero for the control argument, in addition to passing values for the other arguments. For further discussion of client usage, see "Using Initializers when Creating New Objects" on page 11-15.

### Selecting Non-default Ancestor Initializer Calls

Often, it will be appropriate (in the first section of an initializer stub procedure) to change the invocation of an ancestor's **somDefaultInit** initializer to some other initializer available on the same class. The rule for making this change is simple: Replace **somDefaultInit** with the name of the desired ancestor initializer, and add any new arguments that are required by the replacement initializer. Important: Under no circumstances can you change anything else in the first section.

This example shows how to change an ancestor-initializer call correctly. Since there is a known "Example1\_withName" initializer, the following default ancestor-initializer call (produced within the stub procedure for "Example2\_withName") can be changed from

```
Example2_Init_Example1_somDefaultInit(somSelf, ctrl);
```

to

```
Example2_Init_Example1_Example1_withName(somSelf, ev, ctrl, name);
```

Notice that the revised ancestor-initializer call includes arguments for an **Environment** and a name, as defined by the "Example1\_withname" initializer.

## Using Initializers when Creating New Objects

There are several ways that client programs can take advantage of the **somDefaultInit** object initialization. If desired, clients can use the SOMobjects API directly (rather than taking advantage of the usage bindings). Also, the general class method, **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 SOMobjects 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, **classNameNew**, 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 *classNameNew\_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 uninitialized the object before freeing memory.

## A Complete Example

Figure 11-9 illustrates the implementation and use of initializers and destructors from the C++ bindings. The first part shows the IDL for three classes with initializers. For variety, some of the classes use callstyle OIDL and others use callstyle IDL.

```
#include <somobj.idl>

interface A : SOMObject {
 readonly attribute long a;
 implementation {
 releaseorder: _get_a;
 functionprefix = A;
 somDefaultInit: override, init;
 somDestruct: override;
 somPrintSelf: override;
 };
};

interface B : SOMObject {
 readonly attribute long b;
 void BwithInitialValue(inout somInitCtrl ctrl,
 in long initialValue);
 implementation {
 callstyle = OIDL;
 releaseorder: _get_b, BwithInitialValue;
 functionprefix = B;
 BwithInitialValue: init;
 somDefaultInit: override, init;
 somDestruct: override;
 somPrintSelf: override;
 };
};

interface C : A, B {
 readonly attribute long c;
 void CwithInitialValue(inout somInitCtrl ctrl,
 in long initialValue);
 void CwithInitialString(inout somInitCtrl ctrl,
 in string initialString);
 implementation {
 releaseorder: _get_c, CwithInitialString,
 CwithInitialValue;
 functionprefix = C;
 CwithInitialString: init;
 CwithInitialValue: init;
 somDefaultInit: override, init;
 somDestruct: override;
 somPrintSelf: override;
 };
};
```

Figure 11-9. A C++ example illustrating the implementation and use of initializers and destructors.

### Implementation Code

Based on the foregoing class definitions, Figure 11-10 on page 11-18 illustrates several important aspects of initializers. The C++ code within this figure 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.
 */

#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",
 somSelf->somGetClassName(), somSelf, somSelf->_get_a((Environment*)0));

 return (SOMObject*)((void*)somSelf);
}
SOM_Scope void SOMLINK BBwithInitialValue(B *somSelf,
 somInitCtrl* ctrl,
 long initialValue)

```

Figure 11-10 (Part 1 of 2). C++ Implementation code and a client program for the initialization and destructor classes.

```

{
 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",
 somSelf->somGetClassName(), somSelf, somSelf->_get_b());
 return (SOMObject*)((void*)somSelf);
}

```

*Figure 11-10 (Part 2 of 2). C++ Implementation code and a client program for the initialization and destructor classes.*

The following initializer in Figure 11-11 on page 11-20 for an object from the class “C” 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;
}

```

*Figure 11-11 (Part 1 of 2). Illustration of how a default ancestor initializer call is replaced with a non-default initializer call.*

```

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",
 somSelf->somGetClassName(), somSelf,
 somSelf->_get_a((Environment*)0),
 somSelf->_get_b(),
 somSelf->_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;
}

```

*Figure 11-11 (Part 2 of 2). Illustration of how a default ancestor initializer call is replaced with a non-default initializer call.*

Figure 11-12 on page 11-22 is a C++ program that creates instances of “A”, “B”, and “C” using the initializers defined above.

```

main()
{
 SOM_TraceLevel = 1;

 A *a = new A;
 a->somPrintSelf();
 delete a;
 printf("\n");

 B *b = new B();
 b->somPrintSelf();
 delete b;
 printf("\n");

 b = new B(22);
 b->somPrintSelf();
 delete b;
 printf("\n");

 C *c = new C;
 c->somPrintSelf();
 delete c;
 printf("\n");

 c = new C((Environment*)0, 44);
 c->somPrintSelf();
 delete c;
 printf("\n");

 c = new C((Environment*)0, "66");
 c->somPrintSelf();
 delete c;
}

```

*Figure 11-12. Illustration of how a default ancestor initializer call is replaced with a non-default initializer call.*

Figure 11-13 on page 11-23 is the output from the preceding program:

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

```

*Figure 11-13 (Part 1 of 2). Illustrated output of a default ancestor initializer call being replaced with a non-default initializer call.*

```

{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

```

*Figure 11-13 (Part 2 of 2). Illustrated output of a default ancestor initializer call being replaced with a non-default initializer call.*

## 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). For a class object, however, **somDefaultInit** normally just sets the name of the class to "unknown," after which the **somInitMIClass** method must be used for the major portion of class initialization. Of course, metaclasses can override **somDefaultInit** to initialize introduced class variables that require no arguments for their initialization.

**Note:** Because **somNew** does not call **somInitMIClass**, class objects returned from invocations of **somNew** on a metaclass are not yet useful class objects.

The **somInitMIClass** method (introduced by **SOMClass**) is invoked on a new class object using arguments to indicate the class name and the parent classes from which inheritance is desired (among other arguments). This invocation is made by whatever routine is used to initialize the class. (For SOMobjects classes using the C or C++ implementation bindings, this is handled by the **somBuildClass** procedure, which is called by the implementation bindings automatically.) The **somInitMIClass** method is often overridden by a metaclass to influence initialization of new classes in some way. Typically, the overriding procedure begins by making parent method calls, and then performs additional actions thereafter.

However, in general, none of the methods introduced by **SOMClass** should be overridden. As a result, customizing the initialization of class objects (other than through overriding **somDefaultInit** for initialization of class variables) is not recommended.

---

## C/C++ Methods and Functions

This section describes methods, functions and macros that client programs can use regardless of the programming language (C or C++) 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 the *OS/390 SOMobjects Programmer's Reference, Volume 1* 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.

**somFindCIsInFile 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 the *OS/390 SOMobjects Programmer's Reference, Volume 1* 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 *OS/390 SOMobjects Programmer's Reference, Volume 1* for more information on a specific method or function.

---

## Customizing SOM Runtime Features

SOMobjects is designed to be policy free and highly adaptable. Most of the SOMobjects behavior can be customized by subclassing the built-in classes and overriding their methods, or by replacing selected functions in the SOMobjects runtime library with application code. This section contains more advanced topics describing how to customize the following aspects of SOMobjects behavior:

- “Customizing Class Loading and Unloading”
- “Method Resolution Advanced Topics” on page 11-30
- “Customizing Memory Management” on page 11-31
- “Customizing SOMobjects Output Processing” on page 11-32
- “Customizing Error Handling” on page 11-33

## Customizing Class Loading and Unloading

SOMobjects 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 DLL Loading

To dynamically load a SOMobjects 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 SOMobjects touches the operating system. Explicit invocation of these functions is never required. However, they are provided to allow class implementers to insert their own code between the operating system and SOMobjects, 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 SOMobjects class *className* and returning either the value zero (for success) or a nonzero system-specific error code. The output argument *modHandle* is used to return a token that can subsequently be used by the DLL-unloading routine (described below) to unload the DLL. The default DLL-loading routine returns the DLL's *module handle* in this argument. The remaining arguments are used as follows:

#### Argument

Usage

*fileName*

The file name of the DLL to be loaded, which can be either a simple name or a full path name.

*functionName*

The name of the routine to be called after the DLL is successfully loaded by the SOMClassMgrObject. This routine is responsible for creating the class objects for the class(es) contained in the DLL. Typically, this argument has

the value “**SOMInitModule**”, which is obtained from the function **SOMClassInitFuncName** described above. If no **SOMInitModule** entry exists in the DLL, the default DLL-loading routine looks in the DLL for a procedure with the name *classNameNewClass* instead. If neither entry point can be found, the default DLL-loading routine fails.

*majorVersion*

The major version number to be passed to the class initialization function in the DLL (specified by the *functionName* argument).

*minorVersion*

The minor version number to be passed to the class initialization function in the DLL (specified by the *functionName* argument).

An application program can replace the default DLL-loading routine by assigning the entry point address of the new DLL-loading function (such as *MyLoadModule*) to the global variable **SOMLoadModule**, as follows:

```
#include <som.h>
/* Define a replacement routine: */
long myLoadModule (string className, string fileName,
 string functionName, long majorVersion,
 long minorVersion, somToken *modHandle)
{
 (Customized code goes here)
}
...
SOMLoadModule = MyLoadModule;
```

## Customizing DLL Unloading

To unload a SOMobjects class, the **SOMClassMgrObject** calls the function pointed to by the global variable **SOMDeleteModule**. The **SOMDeleteModule** function has the following signature:

**long (\*SOMDeleteModule) (in somToken modHandle);**

This function is responsible for unloading the DLL designated by the *modHandle* parameter and returning either zero (for success) or a nonzero system-specific error code. The parameter *modHandle* contains the value returned by the DLL loading routine (described above) when the DLL was loaded.

An application program can replace the default DLL-unloading routine by assigning the entry point address of the new DLL-unloading function (such as, *MyDeleteModule*) to the global variable **SOMDeleteModule**, as follows:

```
#include <som.h>
/* Define a replacement routine: */
long myDeleteModule (somToken modHandle)
{
 (Customized code goes here)
}
...
SOMDeleteModule = MyDeleteModule;
```

## Method Resolution Advanced Topics

In “Method Resolution” on page 2-9, we described the more common method resolution called *offset resolution*. There are two other forms of method resolution, called *name-lookup* and *dispatch-function* resolution which are not as commonly used.

### Name-lookup Resolution

Name-lookup resolution is similar to the method resolution techniques employed by Objective-C\*. It is slower than offset resolution, roughly two to three times the cost of an ordinary procedure call. 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 runtime, or
- The method is added to the class interface at runtime, or
- The name of the class introducing the method isn't known until runtime.

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 runtime which of the two methods should be invoked (because, for example, it will not be known until runtime which class's instance the method will be applied to).

Name-lookup resolution is always performed by the class of an object, in response to a method call. (Offset resolution, by contrast, requires no method calls.) In name-lookup resolution, the class of the target object obtains a pointer to the desired method. In general, this will require a name-based search through various data structures maintained by ancestor classes.

Offset and name-lookup resolution achieve the same net effect (that is, they select the same method procedure); they just achieve it differently (via different mechanisms for locating the method's method token). Offset resolution is faster, because it does not require searching for the method token, but name-lookup resolution is more flexible.

When defining (in SOMobjects IDL) the interface to a class of objects, the class implementer can decide, for each method, whether the SOMobjects 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 SOMobjects use either technique, on a per-method-call basis. Chapter 6, “Developing Client Applications” on page 6-1 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 target object's class. Thus, a class implementer 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.

"Method Declarations Within an Interface Declaration" on page 3-20 describes how SOMobjects's default criterion for selecting method procedures is "replaced" using dispatch functions. For more information on dispatch-function resolution, see the description and examples for the somDispatch and somOverrideMTab methods in the *OS/390 SOMobjects Programmer's Reference, Volume 1*.

## Customizing Memory Management

The memory management functions used by the SOMobjects runtime 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 SOMobjects memory-management function variables and their ANSI standard library equivalents is given in Table 11-1 below.

| Table 11-1. Memory-Management Functions |                                  |             |                  |
|-----------------------------------------|----------------------------------|-------------|------------------|
| SOMobjects 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 |

An application program can replace SOMobjects's memory management functions with its own memory management functions to change the way SOMobjects allocates memory (for example, to perform all memory allocations as suballocations in a shared memory heap). This replacement is possible because SOMCalloc, SOMMalloc, SOMRealloc, and SOMFree are actually *global variables* that point to SOMobjects's default memory management functions, rather than being the names of the functions themselves. Thus, an application program can replace SOMobjects's default memory management functions by assigning the entry-point address of the user-defined memory management function to the appropriate global variable. For example, to replace the default free procedure with the user-defined function **MyFree** (which must have the same signature as the ANSI C **free** function), an application program would require the following code:

```
#include <som.h>
/* Define a replacement routine: */
void myFree (somToken memPtr)
{
 (Customized code goes here)
}
...
SOMFree = MyFree;
```

**Note:** In general, all of these routines should be replaced as a group. For instance, if an application supplies a customized version of SOMMalloc, it should also supply corresponding SOMCalloc, SOMFree, and SOMRealloc functions that conform to this same style of memory management.

### Clearing memory for objects

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

```
#include "origcls.h"

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

 /* Code to use myObject */

 _somFree (myObject);
}
```

### Clearing memory for the Environment

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

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

## Customizing SOMobjects Output Processing

The SOMobjects character-output function is invoked by all of the SOMobjects error-handling and debugging macros whenever a character must be generated as in error messages (see *OS/390 SOMobjects Messages, Codes, and Diagnosis* and **somLPrintf**, **somPrintf** and **somvprintf** functions in *OS/390 SOMobjects Programmer's Reference, Volume 1* for more information). 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. 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.

An application program would use code similar to the following to install the replacement routine:

```

#include <som.h>
/* Define a replacement routine: */
long myCharacterOutputRoutine (char c)
{
 (Customized code goes here)
}

/* After the next stmt all output */
/* will be sent to the new routine */
SOMOutCharRoutine = myCharacterOutputRoutine;

```

## 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 SOMobjects, pointed to by the global variable SOMError, has the following signature:

```
void (*SOMError) (long errorCode , string fileName, long lineNumber);
```

The default error-handling procedure inspects the *errorCode* argument and takes appropriate action, depending on the last decimal digit of *errorCode* (see *OS/390 SOMobjects Messages, Codes, and Diagnosis* 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 (long errorCode, string fileName, long lineNumber)
{
 (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 SOMobjects, SOMobjects 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 SOMobjects runtime 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 SOMobjects methods remain usable and can be re-executed following an error.

---

## Chapter 12. Distributed SOMobjects (DSOM) Advanced Topics

---

### Who Should Read This Chapter

This chapter is for the programmer who understands the distributed SOMobjects (DSOM) concepts discussed in Chapter 7, "Distributed SOMobjects" on page 7-1, and who wants to understand some of the more advanced concepts. This chapter describes the following topics:

- "TCP/IP Communications and the Well-known Port"
- "Designing Local/Remote Transparent Programs" on page 12-2
- "Inquiring about a Remote Object Interface or Implementation" on page 12-18
- "Working with Object References" on page 12-19
- "Finding Existing Objects" on page 12-21
- "Creating Remote Objects Using User-Defined Metaclasses" on page 12-21
- "Updating the Implementation Repository Dynamically Using the Programmatic Interface" on page 12-22
- "Advanced Memory-Management Options" on page 12-24
- "Passing Objects by Copying" on page 12-28
- "Passing Foreign Data Types" on page 12-29
- "Writing Clients that are also Servers" on page 12-33
- "Programming a Server" on page 12-33
- "Building Client-Only "Stub" DLLs" on page 12-51
- "Creating User-Supplied Proxies" on page 12-52
- "Customizing the Default Base Proxy Class" on page 12-55
- "Peer vs. Client-Server Processes" on page 12-56
- "Dynamic Invocation Interface" on page 12-59
- "Guidelines for Direct-to-SOM DSOM Programmers" on page 12-66

---

### TCP/IP Communications and the Well-known Port

Distributed SOMobjects (DSOM) uses TCP/IP *sockets* as the communication vehicle between the distributed components. The addressing information TCP/IP requires to deliver information across the distributed components is the *Internet Protocol (IP)* address and the *well-known port*.

The IP address of the machine where the server programs reside is declared in the implementation repository, a data set that ties specific classes to specific servers. During configuration of the client / server environment, the REGIMPL utility is used to build the implementation repository, declaring which servers (known as **Implementation aliases**) can support which classes. Part of the implementation alias declaration is protocol information that contains the **Host name** designation, which is found at the IP address of the server.

The **well-known port** is a 16-bit integer indicating which process on the host machine should be given the information. In other words, the IP address is like a ZIP code that identifies an entire community of addresses, and the port is synonymous with a PO Box number identifying the final delivery point. The same host can therefore support more than one server environment. Generally, an integer in the range 1024 to 65,534 is chosen on the server system when the SOM subsystem is

installed, then copied to the client environment before an application is executed. The port should be unique, that is, not already in use by other TCP/IP socket services (values below 1024 are typically reserved for services such as Telnet, FTP, and NFS; do not use those values). DSOM examples often show SOMDPORT=14502, but any unique value is sufficient as long as it is *well-known* by clients and their servers. If SOMDPORT is not specified, the default value is 9393. See Appendix A, “Setting up Configuration Files” on page A-1 to see where and how to specify SOMDPORT in the configuration file.

---

## 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 inquire 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 = _newFoo(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 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, _SOMDCClientProxy)) {
 /* 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;
 }
}
```

Another way to create objects with local/remote transparency is to use OMG's COSS Life Cycle service. The COSS Life Cycle service cannot be used from a client-only, pure client, process. This service also suggests the use of factories for object creation. According to it, a client first finds a factory using a factory finder. The client then asks the factory to create an instance through either a generic Create method or a factory-specific Create method. SOMobjects COSS Life Cycle

implementation supports the OMG Factory Finder interface and the generic Create interface. It supports extensions to these interfaces where location information can be specified.

#### **Guideline**

In summary, there are three ways to create objects: Class-based, Factory-based, and COSS Factory-based. Of these, only the last two are suitable for local/remote transparent programming. Use the COSS Factory-based object creation if you want to develop OMG-compliant code. If you want only to write transparent SOM code, then use Factory-based creation.

## **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 **SOMObject**, 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.

When using COSS Life Cycle interfaces, clients get back Life Cycle Objects that support a **delete** method mapped to **somFree**.

#### 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. For OMG's COSS Life Cycle interfaces, use **delete** instead of **somFree**.

## Memory Management of Parameters

In SOMobjects the default memory management policy is corba. All parameter 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 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 **somExceptionFree** 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 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. For example, do not use the code "if (!objref) not transparent". Instead, use the code "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. (For example, the local copy could be made by a DTS C++ compiler using a copy constructor.)

## 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. SOMObjects 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.

## Assignment in DTS C++

A situation similar to the following can occur in a DTS C++ program:

```
Dog Lassie;
Dog Rover = getRemoteDog(); /* Rover is a proxy */
Lassie = Rover; /* What should happen now? */
```

What are the semantics of the above assignment, where one is a local object and the other is a proxy? Is it a copy? If so, deep or shallow? What happens to the current state of Lassie?

The answers depend on how the assignment operator is implemented on Dog.

### Guideline

For writing local/remote transparent assignment operator methods:

- The copy should not break the polymorphism of the from object (for example, it shouldn't try to slice the from object and copy the top slice of bits). Obviously, this will fail when the from object is a proxy. To preserve polymorphism while making a copy, make only method calls and attribute access calls to get the from object's state.
- If it implements deep copy then the secondary copies resulting from this operation should also preserve polymorphism.

---

## 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: either SOM factories that wrapper class objects or COSS factories.
- 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

This 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 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 a 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 12-1. SOM Kernel Functions Classification*

| <b>Method Name</b>         | <b>Classification</b>          |
|----------------------------|--------------------------------|
| <b>somApply</b>            | Variable, Transparent, Special |
| <b>somClassResolve</b>     | Not Forwarded, Transparent     |
| <b>somIsObj</b>            | Not Forwarded, Transparent     |
| <b>is_nil</b>              | Not Forwarded, Transparent     |
| <b>somParentNumResolve</b> | Not Forwarded, Transparent     |
| <b>somParentResolve</b>    | Not Forwarded, Transparent     |
| <b>somResolve</b>          | Not Forwarded, Transparent     |
| <b>somResolveByName</b>    | Not Forwarded, Transparent     |

*Table 12-2. SOMObject Methods Classification*

| Method Name                           | Classification                  |
|---------------------------------------|---------------------------------|
| <code>create_request</code>           | Not Forwarded, Transparent      |
| <code>create_request_args</code>      | Not Forwarded, Transparent      |
| <code>duplicate</code>                | Not Forwarded, Transparent      |
| <code>get_implementation</code>       | Forwarded, Transparent          |
| <code>get_interface</code>            | Forwarded, Transparent          |
| <code>is_proxy</code>                 | Not Forwarded, Transparent      |
| <code>release</code>                  | Not Forwarded, Transparent      |
| <code>somCastObj</code>               | Forwarded, Transparent          |
| <code>somDefaultAssign</code>         | Forwarded, Transparent          |
| <code>somDefaultConstAssig</code>     | Forwarded, Transparent          |
| <code>somDefaultConstCopyInit</code>  | Forwarded, Transparent          |
| <code>somDefaultConstVAssig</code>    | Forwarded, Transparent          |
| <code>somDefaultConstVCopyInit</code> | Forwarded, Transparent          |
| <code>somDefaultInit</code>           | Forwarded, Transparent, Special |
| <code>somDefaultCopyInit</code>       | Forwarded, Transparent          |
| <code>somDefaultVAssign</code>        | Forwarded, Transparent          |
| <code>somDefaultVCopyInit</code>      | Forwarded, Transparent          |
| <code>somDestruct</code>              | Forwarded, Transparent, Special |
| <code>somDispatch</code>              | Variable, Transparent, Special  |
| <code>somClassDispatch</code>         | Not Forwarded, Transparent      |
| <code>somDumpSelf</code>              | Not Forwarded, Transparent      |
| <code>somDumpSelfInt</code>           | Not Forwarded, Transparent      |
| <code>somFree</code>                  | Variable, Transparent, Special  |
| <code>somGetClass</code>              | Forwarded, Transparent          |
| <code>somGetClassName</code>          | Forwarded, Non-Transparent      |
| <code>somGetSize</code>               | Not Forwarded, Transparent      |
| <code>somInit</code>                  | Forwarded, Transparent          |
| <code>somIsA</code>                   | Not Forwarded, Transparent      |
| <code>somIsInstanceOf</code>          | Not Forwarded, Transparent      |
| <code>somPrintSelf</code>             | Not Forwarded, Transparent      |
| <code>somResetObj</code>              | Forwarded, Transparent          |
| <code>somRespondsTo</code>            | Not Forwarded, Transparent      |
| <code>somUninit</code>                | Forwarded, Transparent          |

*Table 12-3. SOMClass Methods*

| Method                        | Classification             |
|-------------------------------|----------------------------|
| <b>somCheckVersion</b>        | Forwarded, Transparent     |
| <b>somDescendedFrom</b>       | Forwarded, Transparent     |
| <b>somGetInstanceSize</b>     | Forwarded, Transparent     |
| <b>somGetName</b>             | Forwarded, Non-Transparent |
| <b>somGetNumMethods</b>       | Forwarded, Transparent     |
| <b>somGetNumStaticMethods</b> | Forwarded, Transparent     |
| <b>somGetParents</b>          | Forwarded, Transparent     |
| <b>somGetVersionNumbers</b>   | Forwarded, Transparent     |
| <b>somNew</b>                 | Forwarded, Transparent     |
| <b>somNewNoInit</b>           | Forwarded, Transparent     |

*Table 12-4. SomClassMgr Methods*

| Method Name                        | Classification             |
|------------------------------------|----------------------------|
| <b>somClassFromId</b>              | Forwarded, Transparent     |
| <b>somFindClass</b>                | Forwarded, Transparent     |
| <b>somFindClssInFile</b>           | Forwarded, Transparent     |
| <b>somGetInitFunction</b>          | Forwarded, Non-Transparent |
| <b>_get_somInterfaceRepository</b> | Forwarded, Transparent     |
| <b>_set_somInterfaceRepository</b> | Forwarded, Transparent     |
| <b>_get_somRegisteredClasses</b>   | Forwarded, Transparent     |
| <b>somLocateClassFile</b>          | Forwarded, Non-Transparent |
| <b>somRegisterClass</b>            | Forwarded, Transparent     |
| <b>somSubstituteClass</b>          | Forwarded, Transparent     |
| <b>somUnloadClassFile</b>          | Forwarded, Transparent     |
| <b>somUnregisterClass</b>          | Forwarded, Transparent     |
| <b>somClassFromId</b>              | Forwarded, Transparent     |
| <b>somFindClass</b>                | Forwarded, Transparent     |
| <b>somFindClssInFile</b>           | Forwarded, Transparent     |

*Table 12-5. DSOM Functions*

| Method Name                   | Classification                  |
|-------------------------------|---------------------------------|
| <b>send_multiple_requests</b> | Forwarded, Transparent, Special |
| <b>somdCreate</b>             | Not-Forwarded, Transparent      |

*Table 12-6. ImplementationDef Methods*

| Method Name                        | Classification         |
|------------------------------------|------------------------|
| <code>get_config_file</code>       | Forwarded, Transparent |
| <code>get_impl_id</code>           | Forwarded, Transparent |
| <code>get_impl_alias</code>        | Forwarded, Transparent |
| <code>get_impl_def_class</code>    | Forwarded, Transparent |
| <code>get_impl_def_struct</code>   | Forwarded, Transparent |
| <code>get_impl_program</code>      | Forwarded, Transparent |
| <code>get_impl_flags</code>        | Forwarded, Transparent |
| <code>get_impl_server_class</code> | Forwarded, Transparent |
| <code>get_impl_refdata_file</code> | Forwarded, Transparent |
| <code>get_impl_refdata_bkup</code> | Forwarded, Transparent |
| <code>get_impl_hostname</code>     | Forwarded, Transparent |
| <code>get_impl_version</code>      | Forwarded, Transparent |
| <code>get_protocols</code>         | Forwarded, Transparent |
| <code>get_svr_objref</code>        | Forwarded, Transparent |

*Table 12-7. ImplRepository Methods*

| Method Name                         | Classification         |
|-------------------------------------|------------------------|
| <code>add_class_toImpldef</code>    | Forwarded, Transparent |
| <code>addImpldef</code>             | Forwarded, Transparent |
| <code>deleteImpldef</code>          | Forwarded, Transparent |
| <code>findAllAliases</code>         | Forwarded, Transparent |
| <code>findAllImpldefs</code>        | Forwarded, Transparent |
| <code>findClassesByImpldef</code>   | Forwarded, Transparent |
| <code>findImpldef</code>            | Forwarded, Transparent |
| <code>findImpldefByAlias</code>     | Forwarded, Transparent |
| <code>findImpldefByClass</code>     | Forwarded, Transparent |
| <code>removeClassFromAll</code>     | Forwarded, Transparent |
| <code>removeClassFromImpldef</code> | Forwarded, Transparent |
| <code>updateImpldef</code>          | Forwarded, Transparent |

*Table 12-8. ORB Methods Classification*

| Method Name                   | Classification             |
|-------------------------------|----------------------------|
| <code>object_to_string</code> | Not-Forwarded, Transparent |
| <code>string_to_object</code> | Not-Forwarded, Transparent |

*Table 12-9. Principal Methods Classification*

| Method Name           | Classification         |
|-----------------------|------------------------|
| <code>userName</code> | Forwarded, Transparent |
| <code>hostName</code> | Forwarded, Transparent |

*Table 12-10. Contained Methods Classification*

| Method Name           | Classification             |
|-----------------------|----------------------------|
| <code>describe</code> | Forwarded, Non-Transparent |
| <code>within</code>   | Forwarded, Non-Transparent |

*Table 12-11. Container Methods Classification*

| Method Name                    | Classification             |
|--------------------------------|----------------------------|
| <code>contents</code>          | Forwarded, Transparent     |
| <code>describe_contents</code> | Forwarded, Non-Transparent |

*Table 12-12. InterfaceDef Method Classification*

| Method Name                     | Classification             |
|---------------------------------|----------------------------|
| <code>describe_interface</code> | Forwarded, Non-Transparent |

*Table 12-13. Repository Methods Classification*

| Method Name                  | Classification         |
|------------------------------|------------------------|
| <code>lookup_id</code>       | Forwarded, Transparent |
| <code>lookup_modifier</code> | Forwarded, Transparent |
| <code>release_cache</code>   | Forwarded, Transparent |

## 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.

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 “Using the IR Framework” on page 15-12.

---

## 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 SOMDOObject, 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 SOMDOObject, 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 SOMDOObject, the SOMDOObject, 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.

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 SOM subsystem 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). “Programming a Server” on page 12-33 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.

---

## 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.

---

## Creating Remote Objects Using User-Defined Metaclasses

An application may wish to define its own constructor methods for a particular class, via a user-supplied metaclass. In this case, the **somdNewObject** method should not be used, since it simply calls the default constructor method, **somNew**, defined by **SOMClass**.

Instead, the application can obtain a proxy to the actual class object in the server process. It can do so via the **somdGetClassObj** method, invoked on the SOMDServer proxy returned by one of the **somdFindServerXxx** methods. The application-defined constructor method can then be invoked on the proxy for the remote class object.

**Note:** The same issues apply to destructor methods. If the application defines its own destructor methods, they can be called via the class object returned by **somdGetClassObj**, as opposed to calling **somdDestroyObject**.

The following example creates a new object in a remote server using an application-defined constructor method, "makeCar", which is assumed to have been defined in the metaclass of "Car", named "MetaCar".

```

#include <somd.h>
#include <Car.h>
main()
{
 Environment ev;
 SOMDServer server;
 Car car;
 MetaCar carClass;

 SOM_InitEnvironment(&ev);
 SOMD_Init(&ev);

 /* find a Car server */
 server = _somdFindAnyServerByClass(SOMD_ObjectMgr, &ev, "Car");

 /* get the class object for Car */
 carClass = (MetaCar) _somdGetClassObj(server, &ev, "Car");

 /* create the car object */
 car = _makeCar(carClass, &ev, "Red", "Toyota", "2-door");

 ...
}

```

## Updating the Implementation Repository Dynamically Using the Programmatic Interface

The implementation repository can be accessed and updated dynamically using the programmatic interface provided by the **ImplRepository** class (defined in "implrep.idl"). The global variable **SOMD\_ImplRepObject** is initialized by **SOMD\_Init Function** to point to the **ImplRepository** object. The following methods are defined on it:

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

Adds an implementation definition to the implementation repository. (Note: The value of the "impl\_id" attribute is ignored. A unique **Implementation ID** will be generated for the newly added **ImplementationDef**.)

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

Deletes an implementation definition from the implementation repository, given the ID of the implementation definition.

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

Updates the implementation definition (defined by the "impl\_id" of the supplied **ImplementationDef**) in the implementation repository

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

Returns a server implementation definition given its ID.

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

Returns a server implementation definition, given its user-friendly alias.

```
sequence<ImplementationDef> findImplDefByClass (in string classname)
```

Returns a sequence of **ImplementationDefs** for those servers that have an association with the specified class. Typically, a server is associated with the classes it knows how to implement, by registering its known classes via the **add\_class\_to\_ImplDef** method.

```
ORBStatus: findAllImplDefs (out sequence<ImplementationDef> outImplDefs)
```

Retrieves all **ImplementationDef** objects in the implementation repository.

The following methods maintain an association between server implementations and the names of the classes they implement. These methods effectively maintain a mapping of <className, Implid>.

```
void addClassToImplDef (in ImplId implid, in string classname)
```

Associates a class, identified by name, with a server, identified by its **Implementation ID**. This type of association is used to look up server implementations via the **findImplDefByClass** method.

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

Removes the association of a particular class with a server.

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

Removes the association of a particular class from all server implementations in the implementation repository.

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

Returns a sequence of class names associated with a server.

With the **ImplRepository** programmatic interface, it is possible for an application to define additional server implementations at run time. This is done by adding an instance of the **ImplementationDef** class to the Implementation Repository and modifying the necessary fields in the **ImplementationDef** object. For more information on how to modify the attributes of the **ImplementationDef** class, see the section on the **ImplementationDef** class in *OS/390 SOMobjects Programmer's Reference, Volume 1*.

---

## 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 3-24.

The SOM Compiler does not perform error checking for memory-management modifiers. To ensure correct behavior in your DSOM application, you should ensure 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 **SOMAlloc**. 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.

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;
_set_myattrib : object_owns_parameters = "myattrib";
```

## 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";
```

## 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 3-24.

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 by-value, 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 **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 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 = \n
"C,struct,dynamic(A_Foreign_Marshaler,foobar)";
/* or */
a_foreign : impctx = \n
"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 12-30, marshals, demarshals or frees foreign data types, similar to the

**SOMDForeignMarshaler::marshal** method. This function has the signature:

```
void _System 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.

---

## 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 “Peer vs. Client-Server Processes” on page 12-56.

---

## Programming a Server

Server programs execute and manage object implementations. That is, they are responsible for:

- Notifying the SOM subsystem that they are ready to begin processing requests,
- Accepting client requests,
- Loading class library DLLs when required,
- Creating/locating/destroying local objects,
- Demarshalling client requests into method invocations on their local objects,
- Marshalling method invocation results into responses to clients, and
- Sending responses back to clients.

As mentioned previously in Chapter 7, “Distributed SOMobjects” on page 7-1, 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. A second server program, somossrv, is also provided for use with the SOMobjects object services (see the chapter on the “Object Services Server” in *OS/390 SOMobjects Object Services* for information about using somossrv). Each of these server programs can be found in the product dataset *sommvs.SGOSLOAD*. The server program name is specified in the PROC for a given server. Workload Manager will start that PROC when it receives a request to start a given server. The “generic” server program is called **somdsvr**. WLM maintains “profile” information for each server in a table. That information points to a proc which in turn points to the server program name (somdsvr in this case). A second server program, **somossrv**, is also provided, to be used with the SOMobjects object services.

The “generic” server program is called **somdsvr** and can be found in  
*sommvs.SGOSLOAD*

Some applications may require additional flexibility or functionality than what is provided by the generic server program. In that case, application-specific server programs can be developed. This section discusses the steps involved in writing such a server program.

To create a server program, a server writer needs to know what services the DSOM run-time environment will provide and how to use those services to perform the duties (listed above) of a server. The DSOM run-time environment provides several key objects that can be used to perform server tasks. These objects and the services they provide will be discussed in this section. Examples showing how to use the run-time objects to write a server are also shown.

Programming the server consists of the following topics:

- “Server Run-Time Objects” on page 12-35
- “Server Activation” on page 12-37
- “Example Server Program” on page 12-38
- “Initializing a Server Program” on page 12-38
- “Processing Requests” on page 12-40
- “Exiting a Server Program” on page 12-41
- “Managing Objects in the Server” on page 12-42
- “Identifying the Source of a Request” on page 12-50
- “Compiling and Linking Servers” on page 12-51

## Server Run-Time Objects

There are three DSOM run-time objects that are important in a server:

- The server's *implementation definition* (**ImplementationDef** or a class derived from **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, the class of its server object, 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 by simply changing the attributes of the server's **ImplementationDef** object in the Implementation Repository.

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. Because the server's **ImplementationDef** object represents the server's identity to DSOM, and because this object is stored in a variable global to the 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. This can be accomplished through use of the REGIMPL utility. For more information, refer to *OS/390 SOMobjects Configuration and Administration Guide*.

## SOM Object Adapter (SOMOA)

The **SOM Object Adapter (SOMOA)** is the main interface between the server application and the DSOM run time. See Figure 12-1 to see the relationship of the SOMOA with the servers and the DSOM runtime.

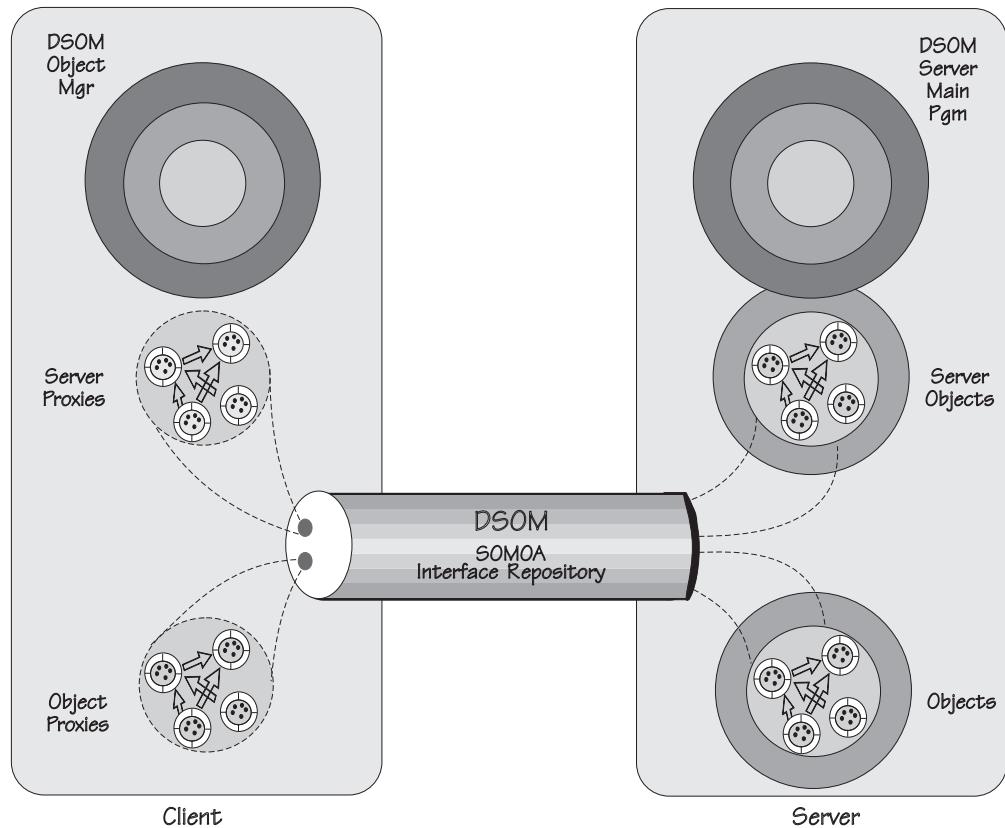


Figure 12-1. Relationship of the SOMOA with servers and the DSOM runtime.

The **SOMOA** is responsible for most of the server duties between the server application and the DSOM runtime. In particular, the **SOMOA** object handles all communications and interpretation of inbound requests and outbound results. When clients send requests to a server, the requests are received and processed by the **SOMOA**.

The **SOMOA** works together with the server object to create and resolve DSOM references to local objects, and dispatch methods on objects.

There is one **SOMOA** object per server process. (The **SOMOA** class is implemented as a *single instance class*.) 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 chapter shows how an application might override the basic **SOMDServer** methods in order to tailor the server object functionality to be a particular application. Also see the chapter on the "Object Services Server" in the *OS/390 SOMobjects 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 **SOMDImplDefObject** 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 SOM subsystem in conjunction with the Workload Manager.
- Manually via command line invocation, or under application control

When a server is activated automatically by the SOM subsystem, 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” 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 the SOM subsystem may obtain its **ImplementationDef** from the Implementation Repository in any manner that is convenient: by ID, by alias, and so forth.

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 “Updating the Implementation Repository Dynamically Using the Programmatic Interface” on page 12-22.

## 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. 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 SOM SOM subsystem 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

Initializing a server program consists of the following topics:

- Initializing the DSOM run-time environment
- Initializing the server's ImplementationDef
- Initializing the SOM object adapter
- When initialization fails

## 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** 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 Function**. 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 the SOM subsystem), then the **ImplementationDef** can be retrieved from the implementation repository. Otherwise, the server program can register its implementation with the implementation repository dynamically (as shown in “Updating the Implementation Repository Dynamically Using the Programmatic Interface” on page 12-22).

The server can retrieve its **ImplementationDef** from the implementation repository by invoking the **find\_ImplDef** method on **SOMD\_ImplRepObject**. It supplies, as a key, the implementation ID of the desired **ImplementationDef**.

The following code shows how a server program might initialize the DSOM run-time environment and retrieve its **ImplementationDef** from the implementation repository.

```
#include <somd.h> /* needed by all servers */
main(int argc, char **argv)
{
 Environment ev;
 SOM_InitEnvironment(&ev);
 /* Initialize the DSOM run-time environment */
 SOMD_Init(&ev);
 /* Retrieve its ImplementationDef from the Implementation
 Repository by passing its implementation ID as a key */
 SOMD_ImplDefObject =
 _find_ImplDef(SOMD_ImplRepObject, &ev, argv[1]);
 ...
}
```

## Initializing the SOM Object Adapter

The next step the server must take before it is ready to accept and process requests from clients is to create a **SOMOA** object and initialize the global variable **SOMD\_SOMOAObject** to point to it. This is accomplished by the assignment:

```
SOMD_SOMOAObject = SOMOANew();
```

**Note:** The **SOMOA** object is not created automatically by **SOMD\_Init Function** 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** object:

```
_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 SOM subsystem. Client applications are assisted by the SOM subsystem in conjunction with WLM to get the server binding information.

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**.

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.

**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 SOM subsystem (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 SOM subsystem.

**Note:** A server program should *not* call **activate\_impl\_failed** once it has called **impl\_is\_ready**.

## Processing Requests

The **SOMOA** is the object in the DSOM run-time environment that receives client requests and transforms them into method calls on local server objects. In order for **SOMOA** to listen for a request, the server program must invoke one of two methods on **SOMD\_SOMOAObject**. If the server program wishes to turn control over to **SOMD\_SOMOAObject** completely (that is, effectively have **SOMD\_SOMOAObject** go into an infinite request-processing loop), then it invokes the **execute\_request\_loop** method on **SOMD\_SOMOAObject** as follows:

```
rc = _execute_request_loop(SOMD_SOMOAObject, &ev, SOMD_WAIT);
```

**Note:** This is the way the DSOM-provided "generic" server program interacts with **SOMD\_SOMOAObject**.

The **execute\_request\_loop** method takes an input parameter of type **Flags**. The value of this parameter should be either SOMD\_WAIT or SOMD\_NO\_WAIT. If SOMD\_WAIT is passed as an argument, **execute\_request\_loop** will return only when an error occurs. If SOMD\_NO\_WAIT is passed, it will return when there are no more outstanding messages to be processed. SOMD\_NO\_WAIT is usually used when the server is being used with the event manager. See "Peer vs. Client-Server Processes" on page 12-56 for details.

If the server wishes to incorporate additional processing between request executions, it can invoke the **execute\_next\_request** method to receive and execute requests one at a time:

```
for(;;) {
 rc = _execute_next_request(SOMD_SOMOAObject, &ev, SOMD_NO_WAIT);
 /* perform app-specific code between messages here, e.g., */
 if (!rc) numMessagesProcessed++;
}
```

Just like **execute\_request\_loop**, **execute\_next\_request** has a **Flags** argument that can take one of two values: SOMD\_WAIT or SOMD\_NO\_WAIT. If **execute\_next\_request** is invoked with the SOMD\_NO\_WAIT flag and no message is available, the method returns immediately with a return code of SOMDERROR\_NoMessages. If a request is present, it will execute it. Thus, it is possible to "poll" for incoming requests using the SOMD\_NO\_WAIT flag.

## Exiting a Server Program

When a server program exits, it should notify the DSOM run time that it is no longer accepting requests. This should be done whether the program exits normally, or as the result of an error.

To notify DSOM when the server program is exiting, the **deactivate\_impl** method defined on **SOMOA** should be called. For example,

```
/* tell DSOM (via SOMOA) that server is now terminating */
_deactivate_impl(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
```

**Note:** For robustness, it would be worthwhile to add appropriate "exit handlers" or "signal handlers" to your application servers that call the **deactivate\_impl** method upon abnormal program termination.

Finally, at the end of a server program, the **SOMD\_Uninit** procedure must be called to free DSOM run-time objects, 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 **SOMOAobject** 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 SOMOOBJECTS 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 **SOMOAobject** 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, and LifeCycle). The **somOS::Server** provides persistent object references and other services. See the chapter on Object Services Server in *OS/390 SOMObjects Object Services* for more information.

### 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 (SOMDOobject).
- 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 (SOMDOobject)
  - 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**. An application-specific implementation of the method **SOMDServer::somdRefFromSOMObj** would use one or more of these methods to create a new object reference (SOMDOobject).

### 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 (SOMDOobject) 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 */
SOMDOobject create_SOM_ref(in SOMObject somobj,
 in ImplementationDef impl);
SOMObject get_SOM_object(in SOMDOobject somref);
/* from SOMDOobject'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 SOMDOObject 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);
```

### 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**.

Because the new Persistent Object Service (POSSOM) uses the **somOS::Server** to provide persistent objects with persistent references, POSSOM users do not need to implement their own persistent servers. Therefore, the text in this section that describes how to build your own persistent server is obsolete when using POSSOM. You can still build your own persistent server as described in this section, but we recommend that you use POSSOM and let it perform these functions for you.

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 SOMObject SOMLINK
 somdRefFromSOMObj(SOMPServer somSelf,
 Environment *ev,
 SOMObject obj)
{
 SOMObject 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 SOMDOObject (the "Ref" in **somdRefFromSOMObj**) from a SOMObject. This implementation invokes **SOMOA::create** to create a SOMDOObject. The prerequisites for asking SOMOA to create a SOMDOObject are:

- Some **ReferenceData** to be associated with the SOMDOObject. 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 SOMDOObject.

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,
 SOMObject 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**:

- Ensure 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 = SOMERROR_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 7-20 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 *OS/390 SOMobjects Programmer's Reference, Volume 1* for a complete description.)

```
void somdDispatchMethod(in SOMObject somobj,
 out somToken retVal,
 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, \n
 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. SOMobjects uses this information to 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 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 rejects any requests from a client that does not present a valid authentication token that it first acquires from the security server. For OS/390 that security server is integrated with the RACF authentication service and a valid authentication token is returned to the client, for use with the target application server, only when the client is authenticated to the RACF registry accessed by the security server. See *OS/390 SOMobjects Configuration and Administration Guide* for more information about the security mechanisms available in SOMobjects.

For more information on CORBA and DSOM, see "How DSOM complies with the Common Object Request Broker Architecture (CORBA)" on page 7-35.

## Compiling and Linking Servers

All server programs must include the header file "somd.h" (or for C++, "somd.xh") in addition to any "<className>.h" (or "<className>.xh") header files they require from application classes. All DSOM server programs must link to the SOMobjects library:

Once the source (in this case, C and header files) is in place, the C/C++ for OS/390 compiler is used to compile all source files to produce object files, which also reside in PDSs. The **GOSSMPCC** sample JCL file shows how the compiler is invoked for an application program. Note that the **DLL EXPORTALL** compiler options are specified, since this SOMobjects code is packaged as DLLs. Optionally, EXPORT parameters would be specified as PRAGMA statements within the C code.

The object files are pre-linked and linked with the DSOM object library *sommvs.SGOSPLKD* and the LE runtime library to produce a module. The sample JCL **GOSSMPLK** shows the pre-link and linker invocation. The result is a module, still in a PDS.

DSOM programs and class libraries are compiled and linked like any other SOM program or library. The header file "somd.h" (or for C++, "somd.xh") should be included in any source program that uses DSOM services. DSOM run-time calls can be resolved by linking with the SOMobjects library.

For more information, see the section on compiling, prelinking, and linking in "Compiling, Prelinking, and Linking" on page 6-33.

---

## Building Client-Only "Stub" DLLs

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 automati-

ically 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 IDL class interface specification, using the h emitter, the ih emitter, and the c emitter. (Alternatively, the xh, xih, and xc emitters can be used.)
- Compile these files together to yield a client-side "stub" DLL, in the same way that regular class DLLs are compiled.

Note that 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 the DSOM run time. The DSOM run time constructs a proxy class by creating an instance of a class that inherits the interface and implementation of **SOMDClientProxy**, and the interface (but not the implementation) of the target class.

**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 *OS/390 SOMobjects Programmer's Reference, Volume 1* or see Chapter 17, "Metaclass Framework" on page 17-1.

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 7-24.

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 invoca-

tions (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 **SOMDClientProxy** and the target object (in this case "Foo"). Thus, the IDL file for the user-supplied proxy class "Foo\_Proxy" (note the two underscores) is as follows:

```
// fooproxy.idl
#include <somdcprx.idl>
#include <foo.idl>
interface Foo_Proxy : SOMDClientProxy, Foo
{
implementation
{
 dllname="fooproxy.dll";
 _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.  
**Important:** The user-supplied proxy class *must* be named "`<targetClassName>_Proxy`" (with two underscores in the name) and **SOMDClientProxy** *must* be the first class in the list of parent classes; for example,

```
interface Foo_Proxy : SOMDClientProxy, Foo
```

In the implementation section of the **.idl** data set, 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** data set. 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 datasets in the STEPLIB

---

## Customizing the Default Base Proxy Class

Continuing the example from the previous topic, imagine that an application derives 100 subclasses from the "Foo" class. If the application wishes to cache the "Foo::attribute\_long" attribute in the proxies for all remote Foo-based objects, the application could supply 100 user-supplied proxy classes, developed in the manner described above. However, this would become a very tedious and repetitive task!

Alternatively, it is possible to provide a customized base proxy class for use in the dynamic generation of DSOM proxy classes. This allows an application to provide a customized base proxy class, from which other dynamic DSOM proxy classes can be derived. This is particularly useful in situations where an application would like to enhance many or all dynamically generated proxy classes with a common feature.

As described in the previous topic, proxy classes are derived from the **SOMDClientProxy** class by default. It is the **SOMDClientProxy** class that overrides **somDispatch** in order to forward method calls to remote objects.

The **SOMDClientProxy** class can be customized by deriving a subclass in the usual way (being careful not to replace **somDispatch** or other methods that are fundamental to implementing the proxy's behavior). To extend the above example further, the application might define a base proxy class called "MyClientProxy" that defines a long attribute called "attribute\_long," which will be inherited by Foo-based proxy classes.

The SOM IDL modifier **baseproxyclass** can be used to specify which base proxy class DSOM should use during dynamic proxy-class generation. To continue the example, if the class "MyClientProxy" were used to construct the proxy class for a class "XYZ," then the **baseproxyclass** modifier would be specified as follows:

```

// xyz.idl

#include <somdtype.idl>
#include <foo.idl>

interface XYZ : Foo
{
 ...
 implementation
 {
 ...
 baseproxyclass = MyClientProxy;
 };
};

```

It should be noted that:

- Base proxy classes must be derived from **SOMDClientProxy**.
- If a class "XYZ" specifies a custom base-proxy class, as in the above example, subclasses of "XYZ" do *not* inherit the value of the **baseproxyclass** modifier. If needed, the **baseproxyclass** modifier must be specified explicitly in each class.
- 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.

## Peer vs. Client-Server Processes

The client-server model of distributed computing is appropriate when it is convenient (or necessary) to centralize the implementation and management of a set of shared objects in one or more servers. However, some applications require more flexibility in the distribution of objects among processes. Specifically, it is often useful to allow processes to manage and export some of their objects, as well as access remote objects owned by other processes. In these cases, the application processes do not adhere to a strict client-server relationship. Instead, they cooperate as "peers", behaving both as clients and as servers.

Peer applications must be written to respond to incoming asynchronous requests, in addition to performing their normal processing.

The topics that make up peer vs. client-server processes are:

- Event-Driven DSOM Programs Using Event Manager (EMan)
- Sample Server Using EMan

## Event-Driven DSOM Programs Using Event Manager (EMan)

EMan is not a replacement for threads, but it supports processing of asynchronous requests. EMan allows a program to handle events from multiple input sources — but the handlers run on a single thread, under control of EMan's main loop.

As described in the following sections, this is a very restrictive environment and is easily subject to deadlock for anything other than oneway messages.

DSOM provides a runtime function, `SOMD_RegisterCallback`, which is used by DSOM to associate user-supplied event handlers with DSOM's communications sockets with EMan.

DSOM server programs which use EMan must be very careful not to get into deadlock situations. This is quite easy to do with DSOM, since method calls are synchronous. If two cooperating processes simultaneously make calls on each other, a deadlock could result. Likewise, if a method call on remote object B from A requires a method call back to A, a deadlock cycle will exist. (Of course, the number of processes and objects which create the cyclic dependency could be greater than two.) To illustrate:

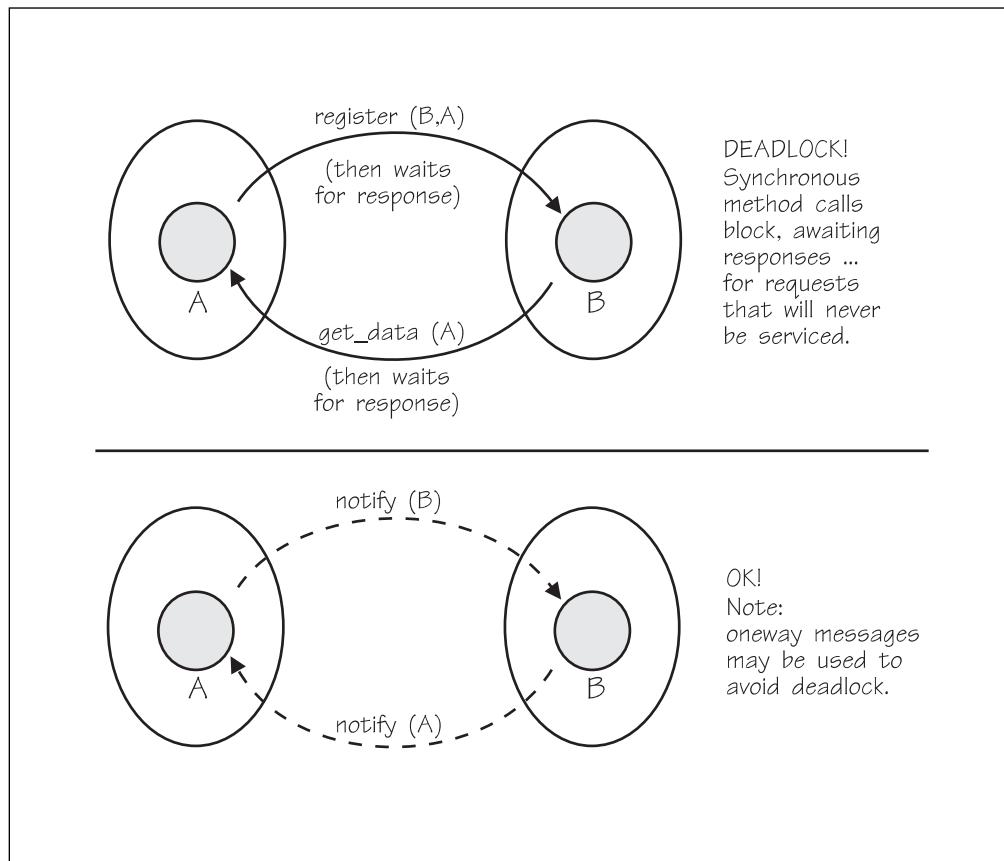


Figure 12-2. Potential Deadlocks Exist Using EMan and DSOM

The application developer must be careful to avoid situations where cooperating processes are likely to make calls upon each other, creating a cyclic dependency. Some applications may find it appropriate to use *oneway* messages to avoid deadlock cycles, since oneway messages do not cause a process to block. It may also be possible for an application to defer the actual processing of a method that may "call back" an originating process, by scheduling work using EMan client events.

## Sample Server Using EMan

The following server code has been distilled from one of the DSOM sample applications provided with SOMobjects. It is an example of a server which has an interval timer that signals another server (via DSOM) whenever its timer "pops". Thus, it is both a client (of the server it signals) and a server (because it can receive timer notifications from other servers).

The IDL for the server object class to be used by this server program is as follows. Note that the "noteTimeout" method is *oneway*, in order to avoid deadlock.

```
interface PeerServer : SOMDServer
{ oneway void noteTimeout(in string serverName);
 // Notification that a timer event occurred in server serverName
};
```

The example server program is outlined as follows. It is assumed that "eman.h" has been included by the program.

- Perform DSOM initialization up to, but not including, asking **SOMOA** to start handling requests.

```
MyEMan = SOMEEManNew();
SOM_InitEnvironment(&ev);
SOM_InitEnvironment(&peerEv);
SOMD_Init(&ev);

somPrintf("What is the alias for this server? ");
gets(thisServer);

SOMD_ImplDefObject = _find_impldef_by_alias(SOMD_ImplRepObject,
 &ev, thisServer);
SOMD_SOMOAObject = SOMOANew();
_impt_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
```

- Register a "DSOM event" with EMan, having EMan callback to a procedure that asks the **SOMOA** to process any pending DSOM requests.

```
void SOMD_RegisterCallback(SOMEEMan emanObj, EMRegProc *func);
void DSOMEVENTCALLBACK (SOMEEvent event, void *eventData)
{
 Environment ev;
 SOM_InitEnvironment(&ev);
 _execute_request_loop(SOMD_SOMOAObject, &ev, SOMD_NO_WAIT);
}

SOMD_RegisterCallback (MyEMan, DSOMEVENTCALLBACK);
```

- Ask user to provide "target server's alias", where the target server is the one that this server will signal when its timer "pops". Then get a proxy for that server.

```
somPrintf("What is the alias for the target server? ");
gets(inbuf);
RemotePeer = _somdFindServerByName(SOMD_ObjectMgr, &ev, inbuf);
```

- Ask user to provide the timer's interval (in milliseconds)

```
somPrintf("What is the timer interval, in milliseconds?");
gets(inbuf);
Interval = atoi(inbuf);
```

- Register a timer event with EMan, having EMan call back a procedure that will invoke the notification method on the target server.

```

void TimerEventCallBack (SOMEEvent event, void *eventData)
{ Environment ev;
 SOM_InitEnvironment(&ev);
 /* call the peer, with a oneway message */
 _noteTimeout(RemotePeer, &ev, thisServer);
}

data = SOMEEMRegisterDataNew();
_someClearRegData(data, &ev);
_someSetRegDataEventMask(data, &ev, EMTimerEvent, NULL);
_someSetRegDataTimerInterval(data, &ev, Interval);
somPrintf("Type <Enter> key when ready to go: ");
gets(inbuf);
regId = _someRegisterProc(MyEMan, &ev, data, TimerEventCallBack,
NULL);

```

**Important:** Do *not* use **someRegister** or **someRegisterEv** to register "callback methods" that would be executed on proxy objects. Instead, write a callback routine that invokes the desired method on the proxy, like the one shown above, and register that routine using the method **someRegisterProc**.

Note: EMan currently uses the methods **someRegister** and **someRegisterEv** to obtain the address of a method-procedure to call when a specified event occurs. If EMan directly calls the method-procedure versus **somDispatch**, the method call will not be forwarded to the remote object as desired.

- Start the EMan event processing loop.

```

_someProcessEvents(MyEMan, &ev);

```

Before the sample is run, two server implementations should be registered with REGIMPL. The implementations are identical except for their aliases. One may be called "peerServer1" and the other "peerServer2". The "PeerServer" class should be specified as their server-object class.

Whenever peerServer1's timer pops, the Event Manager causes a method, "noteTimeout", to be sent to the server object in peerServer2. PeerServer2's server object executes this method by displaying a message on its window. Whenever peerServer2's timer pops, a similar sequence occurs with peerServer1. The two servers will run continuously until interrupted.

## Dynamic Invocation Interface

DSOM supports the CORBA dynamic invocation interface (DII), which clients can use to dynamically build and invoke requests on objects. This section describes how to use the DSOM DII. Currently, DSOM supports dynamic request invocation only on objects outside the address space of the request initiator, via proxies. The **somDispatch** method (non-CORBA) can be used to invoke methods dynamically on either local or remote objects, however.

To invoke a request on an object using the DII, the client must explicitly construct and initiate the request. A request is comprised of an object reference, an operation, a list of arguments for the operation, and a return value from the operation. A key to proper construction of the request is the correct usage of the **NamedValue** structure and the **NVList** object. The return value for an operation is supplied to the request in the form of a **NamedValue** structure. In addition, it is usually most

convenient to supply the arguments for a request in the form of an **NVList** object, which is an ordered set of **NamedValues**.

This section covers the following topics:

- “The NamedValue Structure”
- “The NVList Class” on page 12-61
- “Creating Argument Lists” on page 12-62
- “Building a Request” on page 12-62
- “Initiating a Request” on page 12-63
- “Invoking a Request with Example Code” on page 12-64

## The NamedValue Structure

The **NamedValue** structure is defined in C as:

```
typedef unsigned long Flags;

struct NamedValue {
 Identifier name; // argument name
 any argument; // argument
 long len; // length/count of arg value
 Flags arg_modes; // argument mode flags
};
```

where:

*name* is an **Identifier** string as defined in the CORBA specification, and

*arg* is an **any** structure with the following declaration:

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

*\_type* is a **TypeCode**, which has an opaque representation with operations defined on it to allow access to its constituent parts. Essentially the **Typecode** is composed of a field specifying the CORBA type represented and possibly additional fields needed to fully describe the type. See “Using TypeCode pseudo-objects” on page 15-18 for more information about **TypeCodes**.

*\_value* is a pointer to the value of the any structure. **Important:** The contents of “*\_value*” should always be a pointer to the value, regardless of whether the value is a primitive, a structure, or is itself a pointer (as in the case of object references, strings and arrays). For object references, strings and arrays, *\_value* should contain a *pointer to the pointer* that references the value. For example:

```
string testString;
any testAny;

testAny._value = &testString;
```

*len* is the number of bytes that the argument value occupies.

The *arg\_modes* field is a bitmask (unsigned long) and may contain the following flag values:

|           |                                                 |
|-----------|-------------------------------------------------|
| ARG_IN    | the associated value is an input-only argument  |
| ARG_OUT   | the associated value is an output-only argument |
| ARG_INOUT | the associated argument is an in/out argument   |

These flag values identify the parameter passing mode when the **NamedValue** represents a method parameter. Additional flag values have specific meanings for **Request** and NVList methods and are listed with their associated methods.

## The NVList Class

An **NVList** contains an ordered set of **NamedValues**. 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 *OS/390 SOMobjects Programmer's Reference, Volume 1* for a detailed description of the methods defined on the **NVList** object.

## Creating Argument Lists

A very important use of the **NVList** is to pass the argument list for an operation when creating a request. CORBA 1.1 specifies two methods, defined in the **ORB** class, to build an argument list: **create\_list** and **create\_operation\_list**. The IDL prototypes for these methods are as follows:

```
ORBStatus create_list(
 in long count, /* # of items */
 out NVList new_list);

ORBStatus create_operation_list(
 in OperationDef oper,
 out NVList new_list);
```

The **create\_list** method returns an **NVList** with the specified number of elements. Each of the elements is empty. It is the client's responsibility to fill the elements in the list with the correct information using the **set\_item** method. Elements in the **NVList** must contain the arguments in the same order as they were defined for the operation. Elements are numbered from 0 to count-1.

The **create\_operation\_list** method returns an **NVList** initialized with the argument descriptions for a given operation (specified by the **OperationDef**). The arguments are returned in the same order as they were defined for the operation. The client only needs to fill in the *item\_value* and *value\_len* in the elements of the **NVList**.

In addition to these CORBA-defined methods, DSOM provides a third version, defined in the **SOMDObject** class. The IDL prototype for this method is as follows:

```
ORBStatus create_request_args(
 in Identifier operation,
 out NVList arg_list,
 out NamedValue result);
```

Like **create\_operation\_list**, the **create\_request\_args** method creates the appropriate **NVList** for the specified operation. In addition, **create\_request\_args** initializes the **NamedValue** that will hold the result with the expected return type. The **create\_request\_args** method is defined as a companion to the **create\_request** method, and has the advantage that the **InterfaceDef** for the operation does not have to be retrieved from the Interface Repository.

**Note:** The **create\_request\_args** method is *not* defined in CORBA 1.1. Hence, the **create\_operation\_list** method, defined on the **ORB** class, should be used instead when writing portable CORBA-compliant programs.

## Building a Request

There are two ways to build a **Request** object. Both begin by calling the **create\_request** method defined by the **SOMDObject** class. The IDL prototype for **create\_request** is as follows:

```

ORBStatus create_request(
 in Context ctx,
 in Identifier operation,
 in NVList arg_list,
 inout NamedValue result,
 out Request request,
 in Flags req_flags);

```

The *arg\_list* can be constructed using the procedures described above and is passed to the **Request** object in the **create\_request** call. Alternatively, *arg\_list* can be specified as NULL and repetitive calls to **add\_arg** can be used to specify the argument list. The **add\_arg** method, defined by the **Request** class, has the following IDL prototype:

```

ORBStatus add_arg(
 in Identifier name,
 in TypeCode arg_type,
 in void* value,
 in long len,
 in Flags arg_flags);

```

The "arg\_modes" field of the *result* **NamedValue** parameter to **create\_request** is ignored.

## Initiating a Request

There are two ways to initiate a request, using either the **invoke** or **send** method defined by the **Request** class. The IDL 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.

## Invoking a Request with Example Code

Below is an incomplete example showing how to use the DII to invoke a request having the following method procedure prototype:

```

string _testMethod(testObject obj,
 Environment *ev,
 long input_value,
);
main()
{
 ORBStatus rc;
 Environment ev;
 SOMDOObject obj;
 NVList arglist;
 NamedValue result;
 Context ctx;
 Request reqObj;
 OperationDef opdef;
 Description desc;
 OperationDescription opdesc;
 static long input_value = 999;

 SOM_InitEnvironment(&ev);
 SOMD_Init(&ev);

 /* create the argument list */
 /* get the operation description from the Interface Repository */
 opdef = _lookup_id(SOM_InterfaceRepository, *ev,
 "testObject::testMethod");
 desc = _describe(opdef, &ev);
 opdesc = (OperationDescription *) desc.value._value;

 /* fill in the TypeCode field for the result */
 result.argument._type = opdesc->result;

 /* Initialize the argument list */
 rc = _create_operation_list(SOMD_ORBObject, &ev, opdef,
 &arglist);

 /* get default context */
 rc = _get_default_context(SOMD_ORBObject, &ev, &ctx);

 /* put value and length into the NVList */
 _get_item(arglist, &ev, 0, &name, &tc, &dummy, &dummylen,
 &flags);

 _set_item(arglist, &ev, 0, name, tc, &input_value,
 sizeof(input_value), flags);
 ...
 /* create the request - assume the object reference came from
 somewhere - from a file or returned by a previous request */
 rc = _create_request(obj, &ev, ctx, "testMethod",
 arglist, &result, &reqObj, (Flags)0);

 /* invoke request */
 rc = invoke(reqObj, &ev, (Flags)0);

 /* print result */
 printf("result: %s\n",*(string*)(result.argument._value));
 return(0);
}

```

---

## Guidelines for Direct-to-SOM DSOM Programmers

When you implement Direct-to-SOM (DTS) SOM classes to be distributed via DSOM, realize that your class will be subclassed (to create a proxy class), and that remote users of your class will actually be using an instance of the subclass (a proxy). This implies that your class's implementation should respect polymorphic behavior. Respecting polymorphic behavior, in turn, means ensuring that polymorphic methods operate on the right data, by using accessor methods rather than direct access to data members.

When you implement a DTS DSOM client program, assume that the objects you are using may be instances of subclasses of the class you think you are using (for example, a proxy object for a remote object). This means that you should always access objects via pointers, and that you should use accessor methods rather than accessing data members directly.

In addition, you should also adhere to the following techniques:

- Use the **SOMAttribute** pragma (used by the DTS compiler) for all data to be accessed via DSOM.
- Use the **SOMNoDataDirect(on)** pragma (used by the DTS compiler).
- Provide a default constructor for DTS SOM classes to be distributed via DSOM.
- Declare method parameter types and return types explicitly. For example, if a method returns an array of **shorts**, use a return type that is an explicit array, rather than **short \***. (DSOM interprets **short \*** as a pointer to a single **short**, and hence would only send a single **short** in the request message.)

---

## Chapter 13. Collection Classes

The Collection Classes constitute a large group of classes and methods provided for the programmer's convenience. Collection Classes, sometimes called Foundation Classes, are a set of classes whose purpose is to contain other objects. These classes and their related methods implement most of the common data structures encountered in programming, thus relieving the programmer of those coding tasks. You can use collection classes in client code or as the basis for deriving new classes.

Reference documentation for these classes and their related methods is in *OS/390 SOMobjects Programmer's Reference, Volume 3*.

This chapter consists of the following topics:

- “Categories of Collection Classes”
- “IsSame versus IsEqual Comparisons” on page 13-2
- “Class Inheritance versus Element Inheritance” on page 13-2
- “Object-Initializer Methods” on page 13-2
- “Naming Conventions” on page 13-3
- “Abstract Classes” on page 13-3
- “Main Collection Classes” on page 13-4
- “Iterator Classes” on page 13-8
- “Mixin Classes” on page 13-9
- “Supporting Classes” on page 13-10

---

### Categories of Collection Classes

The collection classes are organized into the following categories:

#### **Abstract Classes**

Defines the conceptual operations implemented by methods in other classes and subclasses.

#### **Main Collection Classes**

Represents each of the implemented data structures for collecting elements into a group.

#### **Iterator Classes**

Defines an iterator class corresponding to each of the main collection classes. This class enables clients to iterate through each of the objects in the collection.

#### **Mixin Classes**

Defines characteristics that apply to more than one kind of collection class, such as ordering or linking. A collection class may require certain mixin characteristics in objects it collects. To facilitate this, a mixin class can be mixed in with an existing user class to derive a new collectible class.

## Supporting Classes

Provides additional capabilities used internally by collection classes. This class is of interest primarily to those deriving new collection classes.

---

## IsSame versus IsEqual Comparisons

The distinction between the **IsSame** and **IsEqual** operation is an important concept when making comparisons. The various collection classes use one or the other of these approaches to compare contained objects. The operations are defined as follows:

- **IsSame** is true when two objects are the same object; both parts of a comparison are testing the same instantiation.
- **IsEqual** is true when two objects are equivalent objects; two different instantiations contain the same values, or at least values that can be considered the same. Stated differently, the two instantiations are isomorphic.

---

## Class Inheritance versus Element Inheritance

There are two distinct aspects of inheritance that pertain to each class:

- The inheritance of the collection class itself that is derived from its parent or base class.
- The inheritance of the elements or objects that can be inserted into a collection class as a value.

Do not assume that these two inheritances are the same. There are times when a collection class has one parent, but the objects to be inserted into the collection class may have a different parent. Further, a collection class may mandate that elements or values meet certain inheritance requirements before the elements can be stored in that collection container.

---

## Object-Initializer Methods

Most of the collection classes provide optional initializers. These are methods with which a newly created instance can be initialized to some state other than its default. All initializer methods use the following format:

**somf<className>Init<optional postfix>**

The initializers can be used to reset certain default properties of the collection classes. For example, although the **somf\_THashTable** class uses an **IsSame** approach for comparing objects, the initializer method could be used to cause an instance of **somf\_THashTable** to compare using **IsEqual** instead.

Some initializer methods cannot be overridden by inheriting classes. That is, the initializer methods can be used, but they cannot be redefined. Refer to the *OS/390 SOMobjects Programmer's Reference, Volume 3* method descriptions to determine if the method can be overridden or not.

---

## Naming Conventions

The class names for Mixin classes all begin with the prefix **somf\_M**. All other collection classes have names beginning with the prefix **somf\_T**.

The method names of methods defined by the collection classes all begin with the prefix **somf**, without an underscore after the prefix.

---

## Abstract Classes

The concept of an abstract base class is a C++ variation on the abstract class. An abstract base class is a class that not only describes the general concept, it also can not be instantiated. Another aspect of the abstract base class is the notion of pure virtual function. Any child of the parent abstract base class must override each pure virtual function (method) in order to use the function.

While the idea of a pure virtual function is primarily a C++ concept, it is a valid concept in SOM. This is especially true when defining basic behavior in a parent class that applies for all children of that class. This concept allows class implementers the flexibility to use either of two approaches:

- Declare an interface in the parent class to a method that all children must override and redefine. If the method is not overridden, the parent class will print a corresponding message and processing will halt.
- Declare and define an interface in the parent to a method that the children can either accept as their base definition or can override and redefine.

The **somf\_TIterator** class provides an example of an abstract class that declares a method interface in the parent that all children must override and redefine. Specifically, the **somf\_TIterator** class declares the **somfFirst Method** and the **somfNext Method**, that all children derived from **somf\_TIterator** must override.

The Abstract Classes include the following classes:

### **somf\_TCollection Class**

Represents a group of objects.

### **somf\_TIterator Class**

Declares the behavior common to all iterator classes. An iterator for a particular collection class will iterate over each element contained in an object of that class.

### **somf\_TSequence Class**

Declares the behavior common to all collections whose elements are ordered.

### **somf\_TSequencelerator Class**

Declares the behavior of all iterators for children of the **somf\_TSequence** class. This class is a child of the **somf\_TIterator** class.

## Main Collection Classes

The set of main collection classes contains data structure classes for the following kinds of data structures, as described in the subsequent topics:

- “Hash Table Class: somf\_THashTable”
- “Dictionary Class: somf\_TDictionary” on page 13-5
- “Set Class: somf\_TSet” on page 13-5
- “Deque, Queue and Stack Class: somf\_TDeque” on page 13-6
- “Linked List Class: somf\_TPrimitiveLinkedList” on page 13-6
- “Sorted Sequence Class: somf\_TSortedSequence” on page 13-6
- “Priority Queue Class: somf\_TPriorityQueue” on page 13-7
- “Choosing the Best Class” on page 13-7

### Hash Table Class: somf\_THashTable

A hash table is a table consisting of (key, value) pairs. The *key* provides the means for mapping into the table, and the *value* is the data element to be stored in the hash table. A hash table data structure is implemented by the **somf\_THashTable Class**.

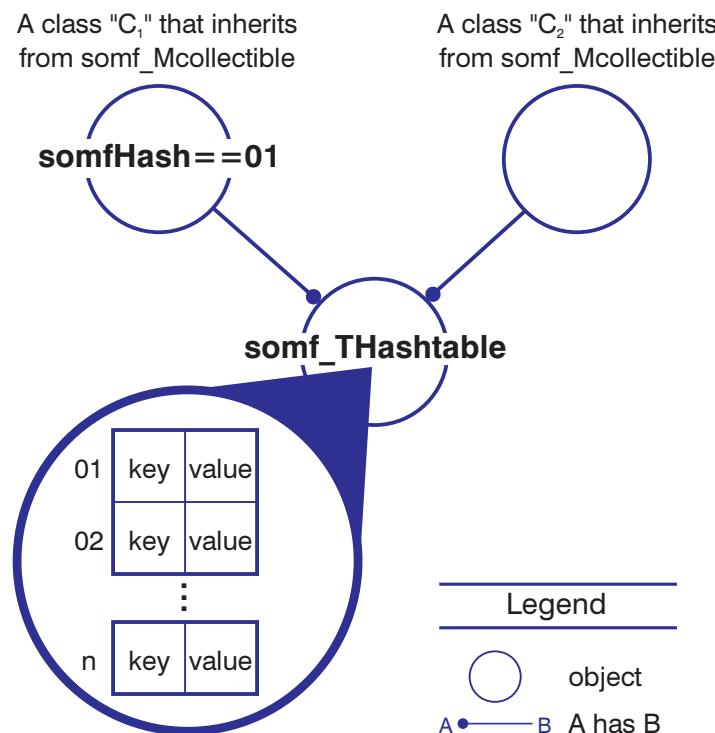


Figure 13-1. Format of the somf\_THashTable class

The key is not used directly to map into the table, because the key could be any sort of class containing a wide variety of data. Rather, the key must have a corresponding **somfHash** method (inherited from the **somf\_MCollectible Class**) which provides a hashing algorithm to compute the hash value, or probe, used to map directly into the table.

If the key were a character string, the hashing algorithm could be either the number of characters in the string or the sum of the ASCII code for each letter. The programmer can choose which algorithm to use, but that algorithm must be consistent.

Since **somfHash** will probably determine its hashing probe based on the state of a particular instance of the class, a specific state must consistently mean that **somfHash** returns the same hashing probe. If the state of the object changes, **somfHash** could return a different hashing probe, but that algorithm must then remain consistent until the state changes again. In short, once a method is used, it should consistently map to the same location thereafter, provided nothing else changes.

Both the keys and the values inserted into the **somf\_THashTable** must inherit from the **somf\_MCollectible** class. When the key inherits from the **somf\_MCollectible** class, it overrides **somfHash** to provide the hashing algorithm for the table.

Objects of the **somf\_THashTable** class use the **IsSame** operation to compare objects (see **IsSame versus IsEqual Comparisons**). This means that a single instance of the key can map into the hash table only once. Keys that are equal can be used without a problem, but the same key can only appear in the hash table once.

**Note:** In the event that multiple values are needed for the same key, you might consider storing a deque, a set or a list as the value. Then, the value in the pair would be a pointer to another collection.

## Dictionary Class: **somf\_TDictionary**

A dictionary is an unordered data structure with (key, value) pairs. It can be thought of as the cousin of the hash table, because they are so similar. The dictionary's primary difference from the hash table is the dictionary uses an **IsEqual** operation to compare objects (see **IsSame versus IsEqual Comparisons**). This means equal keys can only appear in the dictionary once.

The dictionary data structure is implemented by the **somf\_TDictionary Class**. Both the keys and values that are inserted into the **somf\_TDictionary** inherit from the **somf\_MCollectible Class**. Just as for the hash table, when a key inherits from the **somf\_MCollectible** class, it overrides the **somfHash** method to provide the hashing algorithm for the dictionary

In the event that multiple values are needed for keys that are equal, you might consider storing a deque, a set, or a list as the value. Then, the value in the pair would be a pointer to another collection.

## Set Class: **somf\_TSet**

A set is an unordered collection of objects where the objects can only appear once. A set does not contain (key, value) pairs; it only contains objects. A set is different from a deque or list, because sets involve a unique group of methods: intersections, unions, exclusive or, and differences. These are common concepts in set theory.

Set structures are implemented by the **somf\_TSet Class**, and the objects you can insert into them inherit from the **somf\_MCollectible Class**.

## Deque, Queue and Stack Class: **somf\_TDeque**

This class encompasses three kinds of lists: queue, stack and deque. All three of these data structures are implemented in the **somf\_TDeque Class**, with different methods processing the logically different structures. However, the **somf\_TDeque** class is more than all three data structures combined, because elements can be inserted and removed from any point in the **somf\_TDeque**. In addition, the **somf\_TDeque** is probably the most flexible of the data structures, because an element can appear in it more than once, and the only ordering in the data structure is determined by how elements are inserted into it.

All elements inserted into the **somf\_TDeque** inherit from the **somf\_MCollectible Class**.

## Linked List Class: **somf\_TPrimitiveLinkedList**

A linked list is a collection where each element in the list is linked to the elements in front of and behind it. Insertion into the list is relative to the elements already in the list.

A linked list is implemented by the **somf\_TPrimitiveLinkedList Class**. This is probably one of the simplest of data structures, but consequently it is one of the most restrictive:

- The **somf\_TPrimitiveLinkedList** class is the only main collection class that does not inherit from **somf\_MCollectible Class**. This cuts down on processing overhead, but it also means that an instance of **somf\_TPrimitiveLinkedList** cannot be inserted into any other main collection class.
- Elements in the **somf\_TPrimitiveLinkedList** class inherit from **somf\_MLinkable Class**, which means the linkability of each object is inherent in the object itself. Hence, each object can be inserted into the **somf\_TPrimitiveLinkedList** only once, because it has only one set of unalterable forward/backward links.

Although the **somf\_TPrimitiveLinkedList** class carries no baggage from the other classes and is as compact as possible, it is not very flexible. If more flexibility is needed for a linked list, you should probably consider the **somf\_TDeque Class**.

## Sorted Sequence Class: **somf\_TSortedSequence**

A sorted sequence is a collection where the order of its elements is determined by how those elements relate to each other. A sorted sequence data structure is implemented by the **somf\_TSortedSequence Class**.

Before elements can be inserted, the **somf\_TSortedSequence** first must determine the relationship of the elements to each other. Therefore, elements eligible for insertion into a **somf\_TSortedSequence** data structure must inherit from the **somf\_MOrderableCollectible Class** and must override methods **somfIsEqual**, **somfIsLessThan** and **somfIsGreater Than**, so that **somf\_TSortedSequence** will be able to position them properly.

## Priority Queue Class: **somf\_TPriorityQueue**

A *priority queue* is a special case of the sorted sequence. Its ordering is based on the elements relationship to each other, but this queue is geared toward holding larger volumes of data.

Robert Sedgewick in Algorithms in C++ (Addison-Wesley, 1992) describes a priority queue as:

"In many applications, records with keys must be processed in order, but not necessarily in full sorted order and not necessarily all at once. Often a set of records must be collected, then the largest processed, then perhaps more records collected, then the next largest processed, and so forth. An appropriate data structure in such an environment is one that supports the operations of inserting a new element and deleting the largest element. Such a data structure, which can be contrasted with queues (delete the oldest) and stacks (delete the newest) is called a priority queue."

The priority queue is implemented by the **somf\_TPriorityQueue Class**. The unique purpose of the **somf\_TPriorityQueue** is to provide a high-performance sorted sequence, to accommodate a large volume of data. A drawback is that the **somf\_TPriorityQueue** class defines slightly fewer methods than does the **somf\_TSortedSequence Class**.

Before elements can be inserted into it, the **somf\_TPriorityQueue** must first determine the relationship of those elements to each other. Thus, to be eligible for insertion into the **somf\_TPriorityQueue**, elements must inherit from the **somf\_MOrderableCollectible Class** and must override the methods **somfIsEqual**, **somfIsLessThan** and **somfIsGreater Than**, so that the **somf\_TPriorityQueue** will be able to position them correctly.

## Choosing the Best Class

If you are unsure which main collection class you should use, the following selection chart may prove helpful.

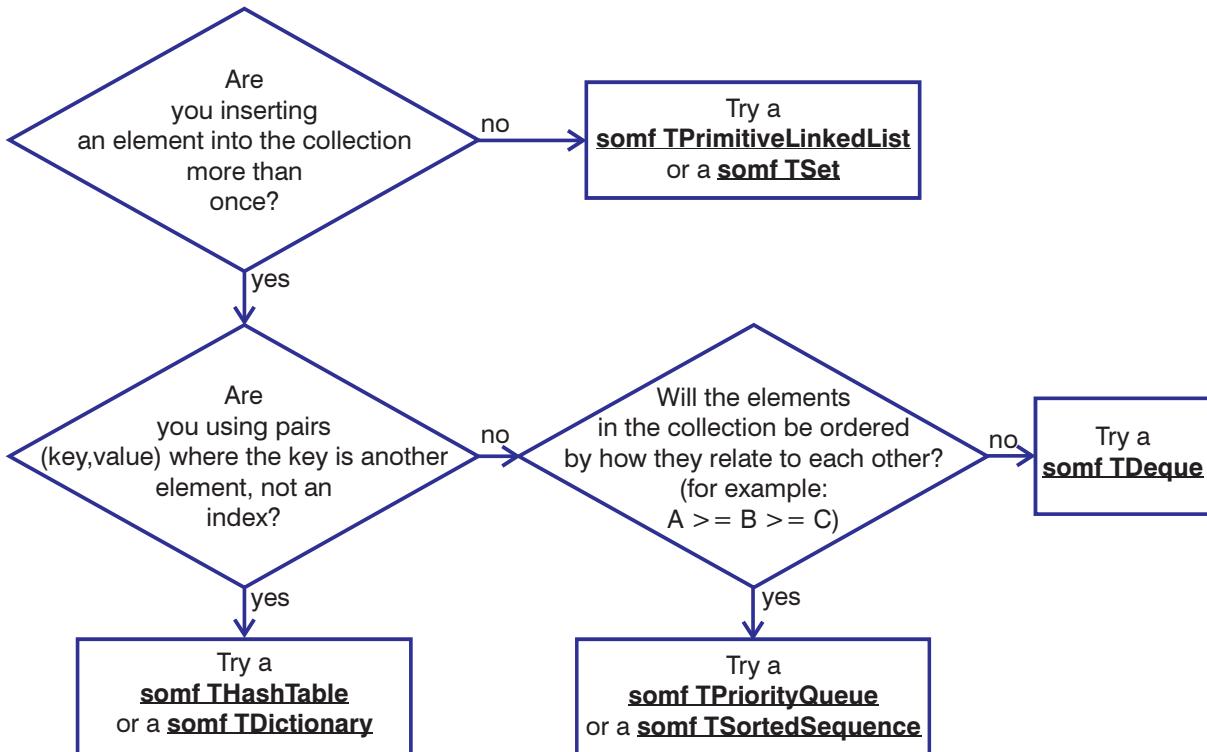


Figure 13-2. Selection chart for the Main Collection Classes

## Iterator Classes

Each of the main collection classes has a corresponding iterator class defined for it. An iterator for a particular object will iterate over each contained object in the collection. To illustrate, the following example uses an iterator of class **somf\_TSetIterator** to iterate over each element in an object of the **somf\_TSet** class.

```

#include <tset.h>
#include <tsetitr.h>

...

somf_TSet *set;
somf_TSetIterator *itr;
somf_MCollectible *obj;
Environment *ev;

ev = somGetGlobalEnvironment();
set = somf_TSetNew();

/* A bunch of stuff happens to set */

itr = somf_TSet_somfCreateIterator(set, ev);
obj = _somfFirst(itr, ev);
while (obj != SOMF_NIL)
{
 obj = _somfNext(itr, ev);
}
...

```

The **somfFirst** method gets the first element in the collection, and the **somfNext** method thereafter gets each next element. Using an iterator allows you to sequentially look at each element in the collection and do some appropriate processing on it.

The iterator was initialized using the method **somfCreateIterator**. All classes that inherit from **somf\_TCollection** must provide a **somfCreateIterator** method. This shows one of the two ways to initialize an iterator; the other way is to use the constructor-initializer associated with the iterator. For example, it could have been declared using:

```
somf_TSetIterator itr;
itr = somf_TSetIteratorNew();
somf_TSetIterator_somfTSetIteratorInit(itr, ev, set);
```

Some people wonder why the iterator logic was not included in the main collection classes. One reason was so the user can create multiple iterators for a single instance of a collection class. If the methods were part of the main collection classes, each instance would be limited to the one iterator that came with it.

If a collection changes while the iterator is in use, the iterator becomes invalid and will issue a notice that it cannot continue to the next element. If a client program calls the collection's **somfAdd** method after starting to iterate through the collection, the iterator will not allow processing to continue. The iterator will have to be reset. The easiest way is to call the iterator's **somfFirst** method and start over.

If a collection is ordered, the iterator returns its elements in the correct order. If the collection is unordered or partially ordered, the iterator returns its elements in some random order.

---

## Mixin Classes

A Mixin class is a class designed "to be mixed in together with other classes to produce new subclasses" (Grady Booch, *Object Oriented Design with Applications*, Cummings Publishing). Mixin classes do not necessarily describe stand-alone characteristics: just characteristics that may be common to more than one kind of class. For example, classes describing a "car" or a "dress" might both inherit from a Mixin class describing "red."

Another characteristic of mixin classes is that they inherit only from other mixin classes (not including **SOMObject**). A mixin class can not inherit from some other base class without a **somf\_M** prefix.

For any object to be eligible for insertion into one of the main collection classes, that object must inherit from a Mixin class. This is necessary because the mixin class declares certain behavior that the main collection class requires in the object in order to process it. For example, the **somf\_MCollectible** mixin class declares the **somfIsEqual** method that is needed to compare objects in almost every collection.

To use a collection class for storing their own objects, programmers must first define a class whose instances will contain the required mixin characteristics. By

using multiple inheritance, a class can inherit from zero or more mixin classes, as well as from its logical parent. For example, if you want to store objects in a sorted sequence structure of class **somf\_TSortedSequence**, those objects must be instances of a class defined to inherit characteristics from the **somf\_MOrderableCollectible** mixin class, such as:

```
interface MySortSeqData : MyData, somf_MOrderableCollectible
```

The main collection classes uses several mixin classes:

#### **somf\_MCollectible Class**

Defines the generic methods needed by objects inserted into any of the collections classes. It provides the profile for the methods **somfIsEqual**, **somfIsSame** and **somfHash**.

#### **somf\_MLinkable Class**

Defines the general characteristics of objects that contain links.

#### **somf\_MOrderableCollectible Class**

Defines the general characteristics of objects that are ordered.

Table 13-1 maps each main collection class to the mixin class from which an object must inherit so that object can be eligible for insertion into the corresponding main collection class:

*Table 13-1. Mapping of main collection classes to mixin classes.*

| Main Collection Class     | Mixin Class from which an inserted object must inherit |
|---------------------------|--------------------------------------------------------|
| somf_TDeque               | somf_MCollectible                                      |
| somf_TDictionary          | somf_MCollectible                                      |
| somf_THashTable           | somf_MCollectible                                      |
| somf_TPrimitiveLinkedList | somf_MLinkable                                         |
| somf_TPriorityQueue       | somf_MOrderableCollectible                             |
| somf_TSet                 | somf_MCollectible                                      |
| somf_TSortedSequence      | somf_MOrderableCollectible                             |

---

## Supporting Classes

Many of the main collection classes use supporting classes. The **somf\_TSortedSequence** class, for example, uses the supporting class **somf\_TSortedSequenceNode** to define the behavior of a single node in a sorted sequence collection.

Included are the following supporting classes:

#### **somf\_TAssoc Class**

Is used to hold a pair of objects.

**somf\_TAssoc** may only be of interest if you are working with"

**somf\_THashTable** or **somf\_TDictionary**, since these two classes store (key, value) pairs.

#### **somf\_TDequeLinkable Class**

Inherits from **somf\_MLinkable** and provides a generic version of **somf\_MLinkable** containing a long value. The **somf\_TDequeLinkable** class is used by **somf\_TDeque**.

**somf\_TDequeLinkable** will probably not be of interest unless you plan to derive a new collection class.

#### **somf\_TSortedSequenceNode Class**

Represents a node in a tree containing elements of the **somf\_MOrderableCollectible** class. It contains a key and a link to a left and a right child.

**somf\_TSortedSequenceNode** will probably not be of interest unless you plan to derive a new collection class.

#### **somf\_TCollectibleLong Class**

Provides a generic **somf\_MCollectible** class containing a long value.

**somf\_TCollectibleLong** will be of interest if you need a generic **somf\_MCollectible** containing a long. **somf\_TCollectibleLong** is not used by any of the other Collection Classes.



---

## **Part 3. SOMobjects Frameworks**



---

## Chapter 14. Emitter Framework

Chapter 4, “SOMobjects Compiler” on page 4-1 discussed the various kinds of output that the SOMobjects Compiler generated. For example, the implementation template (c, xc), bindings for implementers (ih, xih), and bindings for client programs (h,xh).

If you think you might be creating outputs different from the outputs created in Chapter 4, “SOMobjects Compiler” on page 4-1, you will want to read this chapter.

SOMobjects provides collections of classes called frameworks. The Emitter Framework supports the class library builder in creating outputs of various kinds. These outputs can be used by the class library builder and the client programmer.

This chapter addresses the following topics:

- What is the Emitter Framework?
- The uses and benefits of the Emitter Framework
- Understanding the Emitter Framework
- Using the Emitter Framework

---

### What is the Emitter Framework?

The Emitter Framework is a set of classes that provide the information for creating your own emitters.

An emitter is a set of classes that is responsible for generating output forms. The Emitter Framework is a set of support classes and a general emitter class that can be subclassed to produce a specific emitter. These emitters constitute the “back-end” to the SOM Compiler.

The purpose of the SOM Compiler is to translate an IDL interface definition into one or more other useful forms. For example, an interface definition can be translated into a programming language binding file, an implementation template file, a text documenting the IDL interface, or a description that can drive a class browser (a tool which allows you to browse the class libraries).

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 output forms or “back-ends.”.

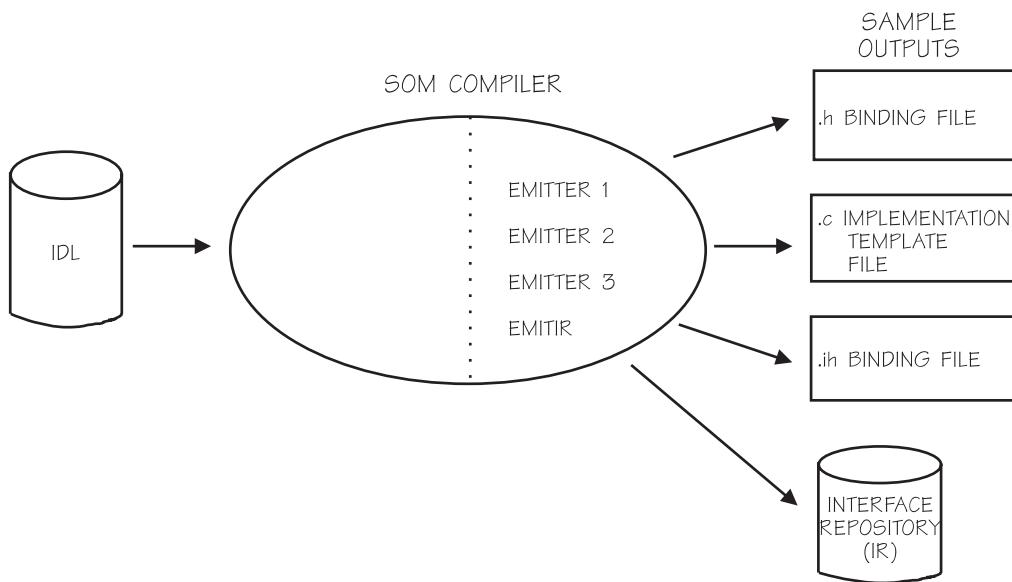
For example, an interface definition can be translated into a programming language binding file, an implementation template file, a documentation file, or a description that can drive a class browser.

SOMobjects allows any programming language to use SOMobjects classes (classes developed using SOM) if that language can:

- Call external procedures
- Store a pointer to a procedure and subsequently invoke that procedure, and

- Map IDL types onto the programming language's native types
- Use Language Environment for MVS & VM Version 1 Release 4 (or later) or MVS C/C++ Language Support feature on MVS/ESA SP Version 5 Release 2 at Language Environment for MVS/VM 1.4 level.

Figure 14-1 shows how the SOM Compiler is structured to utilize a number of “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.



*Figure 14-1. Structure of the SOM Compiler*

To make it easier to develop new emitters for use with the SOM Compiler, SOMobjects 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. SOMobjects also provides the NEWEMIT facility for automatically generating new emitters. This automatically generated emitter is then easily customized as needed for a particular output format.

## The Uses and Benefits of the Emitter Framework

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 IDL.
- 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 (as a template).
- Breaks up the control logic for output-file construction into small, easily maintained (and frequently reusable) units.

## Understanding the Emitter Framework

Understanding the Emitter Framework involves knowing the three types of Emitter Framework Classes. These are:

- The structure of the emitter framework
- Emitter framework classes

## The Structure of the Emitter Framework

The Emitter Framework is structured as shown in Figure 14-2.

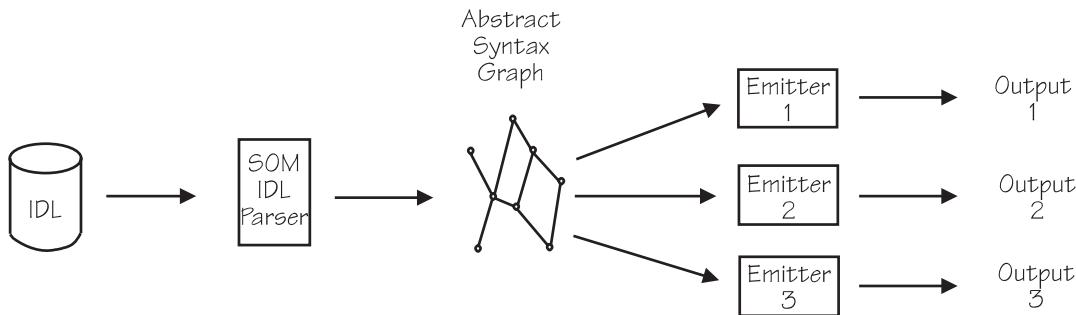


Figure 14-2. The Structure of the SOM Emitter Framework

The following topics comprise the structure of the emitter framework:

- The abstract syntax graph and the object graph builder
- The entry classes
- The emitter class
- The template class and template definitions

### The Abstract Syntax Graph and the Object Graph Builder

The input to the Emitter Framework is an abstract syntax graph of data structures. As shown in Figure 14-1 on page 14-2, 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 (a method declaration, a parameter, an attribute declaration, and so forth).

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, however.

1. 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 a .idl file are accessible to emitters written using the Emitter Framework.
2. 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* (SOMTClassEntryC, etc.), 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—that is, one piece or one “entry” from the complete IDL interface definition. An entry object serves two important functions:

1. Holding information about the corresponding syntactic element of an IDL specification.
2. Defining symbols that can be used as placeholders in an emitter’s *output template*.

“Emitter Framework Classes” on page 14-5 describes the entry classes and the use of symbols in more detail.

## The Emitter Class

The emitter class, SOMTEmitC, 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 (principally, the somtGenerateSections method) 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. For more information on the emitter class, refer to “The emitter class (SOMTEmitC)” on page 14-7.

## The template class and template definitions

The template class, SOMTTemplateOutputC, 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. For more information on the template class and template definitions, refer to "The template class and template definitions" on page 14-4.

The following section discusses the components of the Emitter Framework and describes the recommended procedure for producing a new emitter.

## Emitter Framework Classes

The Emitter Framework consists almost entirely of classes.

- The *emitter class* (SOMTEmitC) 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) manages the output of specific sections to the *target* (output) file. It provides a template facility to make the specification of the output file simple.
- Most of the classes are the *entry classes* (SOMTEEntryC and its subclasses), each of which represents some syntactic unit of an IDL definition.

The only Emitter Framework classes that users will explicitly instantiate (create instances of) are their own subclasses of the emitter class (SOMTEmitC). The remaining classes are instantiated automatically by the Emitter Framework.

Figure 14-3 shows the classes that comprise the Emitter Framework.

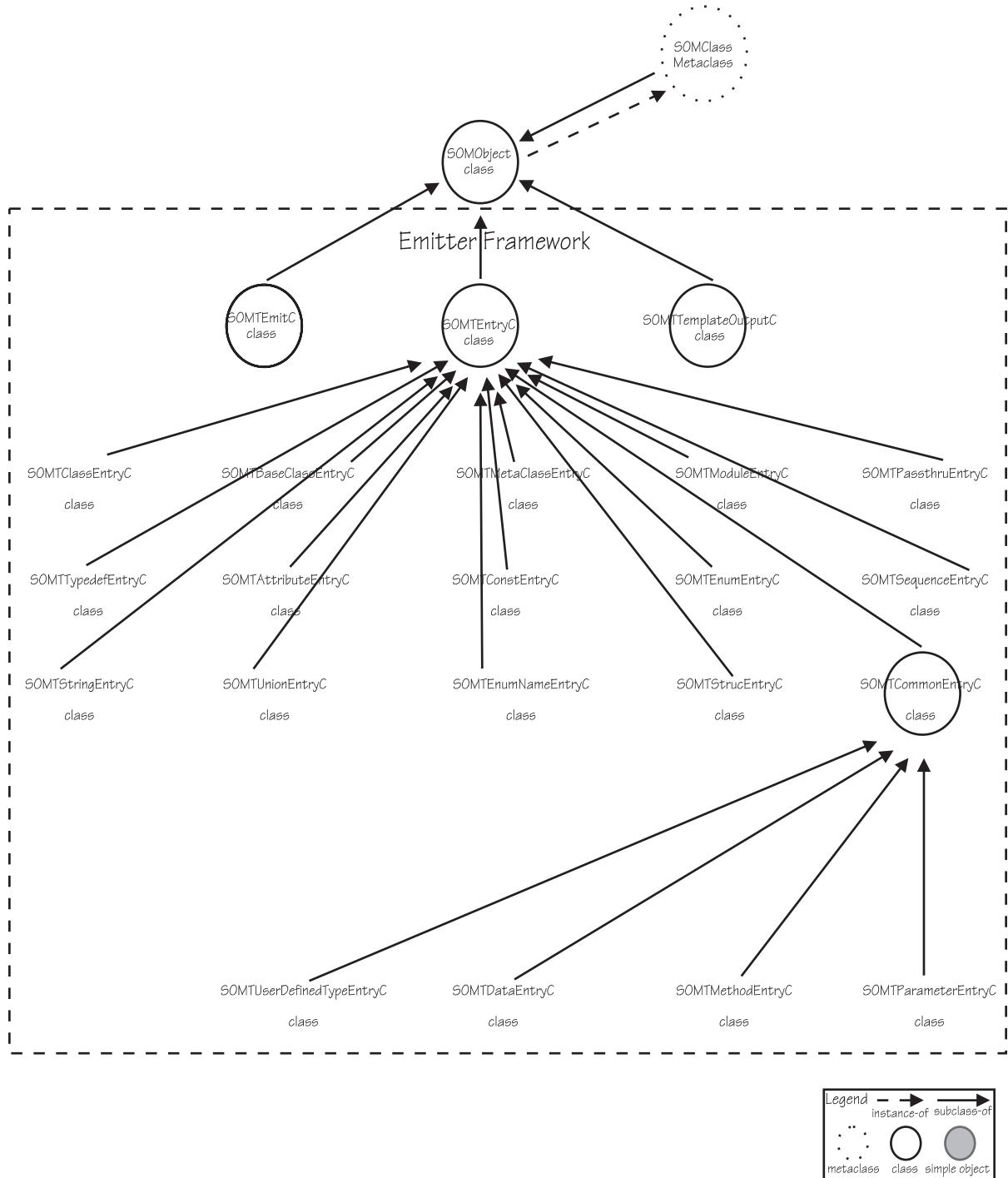


Figure 14-3. Emitter Framework classes

The remainder of this section discusses each of these classes. For more information on all of the attributes and methods supported by each class in the Emitter Framework, see *OS/390 SOMobjects Programmer's Reference, Volume 1*.

## The emitter class (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 (primarily the somtGenerateSections method), and perhaps add a few new methods. The next section describes this process in more detail and consists of the following topics:

- SOMEMitC attributes
- SOMEMitC methods that produce standard sections of an output file
- SOMEMitC names for the standard prolog and epilog sections
- SOMEMitC scanning methods
- SOMEMitC filter methods

**SOMEmitC Attributes:** 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 about which information will be emitted) is represented by an object of class SOMTClassEntryC. This object is the root of the object graph built by the object graph builder when the emitter is invoked on a class definition.
- The *target module* (the module about which information will be emitted) is represented by an object of class SOMTModuleEntryC. This object is the root of the object graph built by the object graph builder when the emitter is invoked on a module definition.
- The *template object* of the emitter (an instance of SOMTTemplateOutputC) maintains the symbol table and controls the format and content of the sections that the emitter produces. (The emitter itself controls which sections are actually 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 passthru in the input .idl file are directed to that emitter.

The SOMTEmitC class provides methods for:

- Opening the output template file (somtOpenSymbolsFile),
- 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), as well as setting standard section-name symbols (somtSetPredefinedSymbols),
- Generating the output file from the output template (somtGenerateSections). The somtGenerateSections method is the primary method that a new emitter will override from SOMTEmitC. This method controls which sections will be emitted and in what order.

**SOMEmitC Methods that Produce Standard Sections of an Output File:** The SOMTEmitC class also provides methods for emitting different standard sections of an output file. The standard sections are as follows, with the default section name given in parentheses:

|                      |                                                                                                                                                                                        |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Prolog</b>        | Describes text to be emitted before any other sections (prologS).                                                                                                                      |
| <b>Base Includes</b> | Determines how base (parent) class #include statements are emitted (baseIncludesS).                                                                                                    |
| <b>Meta Include</b>  | Determines how a metaclass #include statement is emitted (metaIncludeS).                                                                                                               |
| <b>Class</b>         | Determines what information about the class as a whole is emitted (classS).                                                                                                            |
| <b>Base</b>          | Determines what information about a base (parent) classes of a class is emitted (baseS).                                                                                               |
| <b>Meta</b>          | Determines what information about the class's metaclass is emitted (metaS).                                                                                                            |
| <b>Constant</b>      | Determines what information about user-defined constants is emitted (constantS).                                                                                                       |
| <b>Typedef</b>       | Determines what information about user-defined types is emitted (typedefS).                                                                                                            |
| <b>Struct</b>        | Determines what information about user-defined structs is emitted (structS).                                                                                                           |
| <b>Union</b>         | Determines what information about user-defined unions is emitted (unionS).                                                                                                             |
| <b>Enum</b>          | Determines what information about user-defined enumerations is emitted (enumS).                                                                                                        |
| <b>Attribute</b>     | Determines what information about the class's attributes is emitted (attributeS).                                                                                                      |
| <b>Methods</b>       | Determines what information about the methods of a class is emitted (methodsS). More specialized method sections can be specified using inheritedMethodsS or <b>overrideMethodsS</b> . |
| <b>Release</b>       | Determines how information about the release order statement of a class definition is emitted (releaseS).                                                                              |
| <b>Passthru</b>      | Determines what information about passthru statements is emitted (passthruS).                                                                                                          |
| <b>Data</b>          | Determines what information about internal instance variables of a class is emitted (dataS).                                                                                           |
| <b>Interface</b>     | Determines what information about the interfaces in a module is emitted (interfaceS).                                                                                                  |
| <b>Module</b>        | Determines what information about a module is emitted (moduleS).                                                                                                                       |
| <b>Epilog</b>        | 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).

**SOMEmitC Names for the Standard Prolog and Epilog Sections:** 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
moduleEpilogS.
```

The SOMTEmitC class provides methods for emitting each of the sections described above. For example, the somtEmitProlog method emits the prologS section, the somtEmitClass method emits the classS section, and so on.

**SOMEmitC Scanning Methods:** 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 scanning methods are provided by SOMTEmitC

```
somtScanBases, somtScanBasesF, somtScanConstants, somtScanTypedefs,
somtScanStructs, somtScanUnions, somtScanEnums, somtScanAttributes,
somtScanMethods, somtScanData, somtScanDataF, somtScanPassthru,
somtScanInterfaces, and somtScanModules.
```

(The somtScanBasesF, somtScanDataF, and somtScanMethods methods accept a *filter* argument, for selective scanning.) The topic entitled “The section-name symbols” at the end of this chapter 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 14-29).

**SOMEmitC Filter Methods:** 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, the somtNew method determines whether the specified method is introduced by the emitter’s target class. These filter methods can be used as arguments to the somtScanMethods method to control which methods are processed in a repeating section.

Filter methods provided by SOMTEmitC include somtNew, somtImplemented, somtOverridden, somtNotOverridden, somtInherited, somtAll, somtNewProc, somtNewNoProc, somtPrivOrPub, and somtVA. The Reference portion of this book describes each of these methods in more detail.

## The template output class (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. This section will describe in greater detail:

- Simple symbol substitution
- Complex symbol substitution
- SOMTTemplateOutputC methods

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 (defines) 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 in a good separation between decision logic and format specification. Also, because the format specification is isolated, its readability and maintainability are greatly enhanced.

**Simple Symbol Substitution:** 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:** (symbol names) 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.

**Complex Symbol Substitution:** 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 (`somtDashesE`), with “`//`” at the start of each line (`somtCPPE`), in simple C style with each line wrapped in “`/*`” and “`*/`” (`somtCSimpleE`), or in block C style with a leading “`/*`”, then a “`**`” on each line and a final “`*/`” (`somtCBlockE`).
- 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.

Emitting the “`classS`” and “`metaS`” sections from the above template, using the following IDL specification as input:

```

#include <somobj.idl>
#include <mpayroll.idl>

interface Payroll: SOMObject /* This is the interface for Payroll. */
{
 implementation {
 metaclass = M_Payroll;
 functionprefix = "payroll_";
 dataset name = payroll;
 . . .
 };
};

```

would produce the following output:

```

class: Payroll, functionprefix = payroll_, dataset name = payroll;
// This is the interface for Payroll.
metaclass: M_Payroll

```

(The formatting of comments varies, depending on the attributes of the emitter's template.)

**SOMTTemplateOutputC Methods:** The SOMTTemplateOutputC class provides methods for:

- Setting and getting the value of symbols in a template object's symbol table (somtGetSymbol, somtSetSymbol, somtSetSymbolCopyName, somtSetSymbolCopyValue, somtSetSymbolCopyBoth, somtCheckSymbol, and somtExpandSymbol),
- Emitting a particular section of the output template (somtOutputSection),
- Emitting a comment (somtOutputComment),
- Reading the output template file (somtReadSectionDefinitions), and others.

The topic “Defining new symbols” in section 5, “Writing an Emitter Advanced Topics,” describes how to use the symbol-setting methods to define new symbols.

## The entry classes (SOMTEEntryC, 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 in Figure 14-4 on page 14-13.

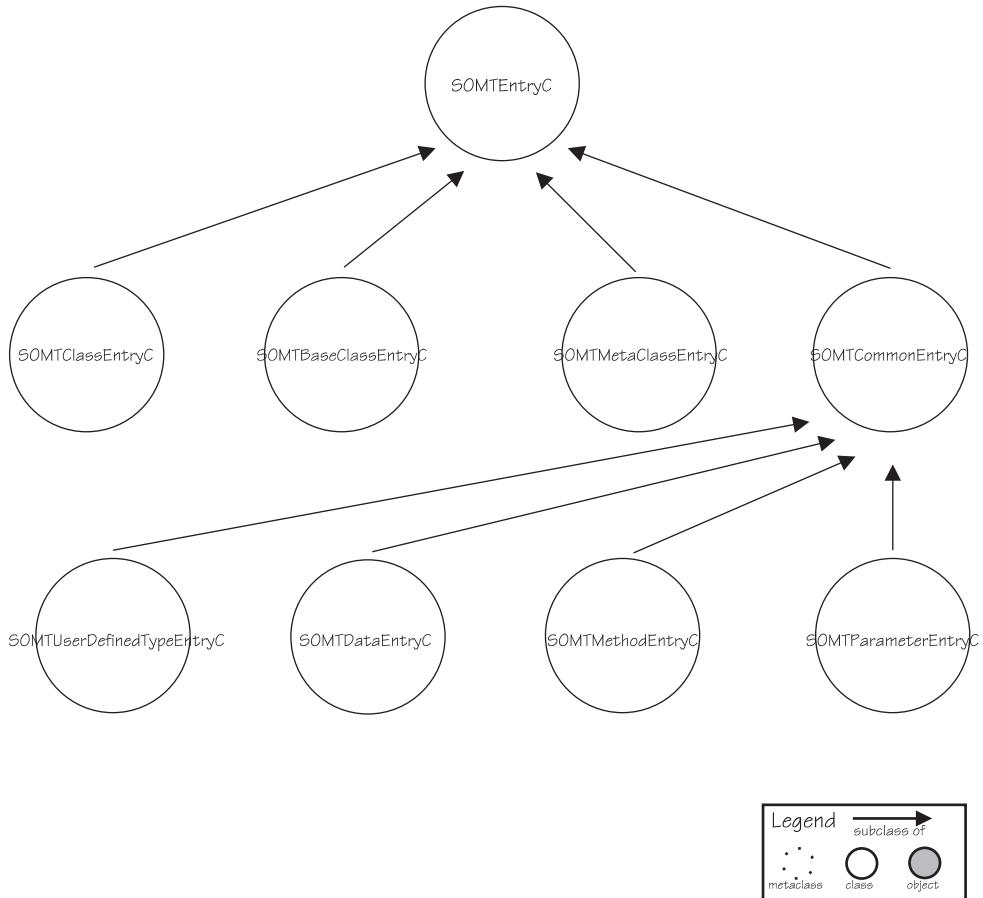


Figure 14-4. The Entry Class Hierarchy

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.

For more information on a particular entry class or method, refer to *OS/390 SOMobjects Programmer's Reference, Volume 1*.

## SOMTEntryC

The SOMTEntryC class provides attributes for accessing the name of an entry (sомtEntryName, somtIDLScopedName, and somtCScopedName), its entry type (that is, whether it represents a class, method, attribute, typedef, etc.) (sомtElementType and somtElementTypeName), its comment (sомtEntryComment), the line number in the .idl file where the entry is defined (sомtSourceLineNumber), its type code (sомtTypeCode), and whether the entry represents a reference to an entry rather than its definition (sомtIsReference).

**SOMTEntryC Methods for Accessing SOM IDL Modifiers** The SOMTEntryC class also provides methods for accessing the SOM IDL modifiers specified in the “implementation” section of an “interface” statement. Included are the methods:

- sомtGetModifierValue,
- sомtGetFirstModifier,
- sомtGetNextModifier,
- sомtFormatModifier, and
- sомtGetModifierList

When invoked on an instance of 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.

**SOMTEEntryC Method to Create Symbols and Define Values for Use in the Output Template:** The SOMTEEntryC class also provides the somtSetSymbolsOnEntry method, which can be used to create symbols and define their corresponding values for use in the output template. For example, SOMTClassEntryC'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. [Symbols are described in more detail under the later topic "The template output class (SOMTTTemplateOutputC)."]

## SOMTCommonEntryC

Entry objects that an emitter uses are actually instances of one of the subclasses of SOMTCommonEntryC, rather than of SOMTCommonEntryC itself. These subclasses are the classes SOMTMethodEntryC, SOMTDataEntryC, SOMTUserDefinedTypeEntryC, and SOMTPParameterEntryC.

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 somtTypeObj whose value is a pointer to a SOMTEEntryC object representing the type, the attribute somtType that gives a string representation of the type, the attribute somtArrayDimString 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: somtGetFirstArrayDimension, somtGetNextArrayDimension, somtIsArray, and somtIsPointer.

## SOMTClassEntryC

A SOMTClassEntryC object anchors the entire interface definition for a class. That is, all the parts of a class's interface definition are reachable from the class entry (SOMTClassEntryC object) that represents it. When an emitter is run on a class's interface definition (rather than on a module), the emitter has a distinct class entry called the *target class entry* which represents that class.

**SOMTClassEntryC Attributes:** The SOMTClassEntryC class provides attributes corresponding to the following characteristics of an IDL interface specification:

- Its source file name (somtSourceFileName),
- Its metaclass (somtMetaClassEntry),
- The class this class is a metaclass for, if any (somtMetaClassFor),
- Whether the entry represents a forward declaration of the class, rather than its definition (somtForwardRef),
- The module that contains the class, if any (somtClassModule),
- The number of methods the class introduces (somtNewMethodCount) or overrides (somtOverrideMethodCount),
- The number of static methods the class introduces (somtStaticMethodCount),

- The number of procedure methods the class introduces (somtProcMethodCount),
- The number of variable argument methods the class introduces (somtVAMethodCount),
- The number of parent (base) classes (somtBaseCount).

**SOMTClassEntryC Methods:** The class also provides methods for accessing each of a class's:

- parent (base) classes (somtGetFirstBaseClass and somtGetNextBaseClass),
- release order names (somtGetFirstReleaseName, somtGetNextReleaseName, and somtGetReleaseList),
- data items (somtGetFirstData and somtGetNextData),
- passthru (somtGetFirstPassthru and somtGetNextPassthru),
- methods (somtGetFirstMethod, somtGetNextMethod, somtGetFirstInheritedMethod, and somtGetNextInheritedMethod),
- constants (somtGetFirstConstant and somtGetNextConstant),
- attributes (somtGetFirstAttribute and somtGetNextAttribute),
- typedefs (somtGetFirstTypedef and somtGetNextTypedef),
- structs (somtGetFirstStruct and somtGetNextStruct),
- unions (somtGetFirstUnion and somtGetNextUnion),
- enumerations (somtGetFirstEnum and somtGetNextEnum), and
- sequences (somtGetFirstSequence and somtGetNextSequence).

SOMTClassEntryC also provides methods for accessing all type/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 (somtFilterNew) or overridden (somtFilterOverridden).

### **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. Rather, it 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 (somtGetFirstInterface and somtGetNextInterface),
- nested modules (somtGetFirstModule and somtGetNextModule),
- constants (somtGetFirstModuleConstant and somtGetNextModuleConstant),
- typedefs (somtGetFirstModuleTypedef and somtGetNextModuleTypedef),
- structs (somtGetFirstModuleStruct and somtGetNextModuleStruct),
- unions (somtGetFirstModuleUnion and somtGetNextModuleUnion),
- enumerations (somtGetFirstModuleEnum and somtGetNextModuleEnum), and
- sequences (somtGetFirstModuleSequence and somtGetNextModuleSequence).

SOMTModuleEntryC also provides methods for accessing the all of the above definitions in the order in which they were defined. These methods are somtGetFirstModuleDef and somtGetNextModuleDef.

## **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 (somtPassthruTarget), the target language (somtPassthruLanguage), and the passthru's contents (somtPassthruBody), as well as a method (somtIsBeforePassthru) for determining whether the passthru is a “before” or “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 (somtTypedefType) of the typedef and methods for accessing each of the declarator names of the typedef (somtGetFirstDeclarator and somtGetNextDeclarator). Because a single typedef may have several declarators (that introduce several user-defined types), 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 (SOMTDataEntryC 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, somtIsSelfRef, 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 (somtAttribType) and whether the attribute is readonly (somtIs Readonly). It also provides methods for accessing the attribute declarators (somtGetFirstAttributeDeclarator and somtGetNextAttributeDeclarator) and their get/set methods (somtGetFirstGetMethod, somtGetNextGetMethod, somtGetFirstSetMethod, and somtGetNextSetMethod).

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 parameter that specifies a variable argument list, using ellipses notation
- 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), and
- The context string literals of the method (somtContextArray).

The SOMTMethodEntryC class also provides methods for getting the method's parameters (somtGetFirstParameter, somtGetNextParameter, somtGetNthParameter, somtGetIDLParamList, somtGetShortCParamList, somtGetFullCParamList, somtGetShortParamNameList, somtGetFullParamNameList).

## **SOMTPParameterEntryC**

Method entries contain a reference to a parameter entry (SOMTPParameterEntryC 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 (somtParameterDirection) that indicates whether it is an in, out, or inout parameter, and attributes that give the parameter's declaration within a prototype (somtIDLParameterDeclaration and somtCParameterDeclaration).

## **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 (somtConstType and somtConstTypeObj) and the value (somtConstIsNegative, somtConstStringVal, somtConstNumVal, somtConstVal, and somtConstNumNegVal) 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 (somtGetFirstEnumName and somtGetNextEnumName).

## **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 (somtSeqLength) and type (somtSeqType).

## **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 (somtStringLength)

## **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 (somtSwitchType) and methods for accessing each of its cases (somtGetFirstCaseEntry and somtGetNextCaseEntry).

## **SOMTEnumNameEntryC**

Every enumeration entry (of type SOMTEnumEntryC) 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 (somtEnumVal) and a pointer to the enumeration that defines the enumerator name (somtEnumPtr).

## **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 (somtStructClass) and whether the struct actually represents an exception (somtIsException), and methods for accessing each of the struct members (somtGetFirstMember and somtGetNextMember).

## **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 (somtOriginalTypedef) and the base type of the user-defined type (somtBaseTypeObj). The somtBaseTypeObj attribute gives the primitive IDL type (float, short, char, etc.) that underlies a user-defined type, skipping over any intermediate user-defined types.

---

## Using the Emitter Framework

Using the Emitter Framework involves the following:

- Writing an emitter- the basics
- Running the NEWEMIT utility
- An example of NEWEMIT in more detail
- Writing an emitter- advanced topics

### Writing an Emitter—the Basics

As previously defined, emitter is a general term used to describe a back-end output component of the SOMObjects Compiler. Each emitter takes, as input, information about an interface generated by the SOMObjects Compiler (as it processes an IDL specification) and produces output organized in a different format.

With the NEWEMIT utility and the Emitter Framework, you can write your own special-purpose emitters. For example, you can write an emitter to produce binding files for different programming languages or documentation files.

### Running the NEWEMIT Utility

To create a new emitter, proceed through the following steps which can be seen in Figure 14-5 on page 14-20.

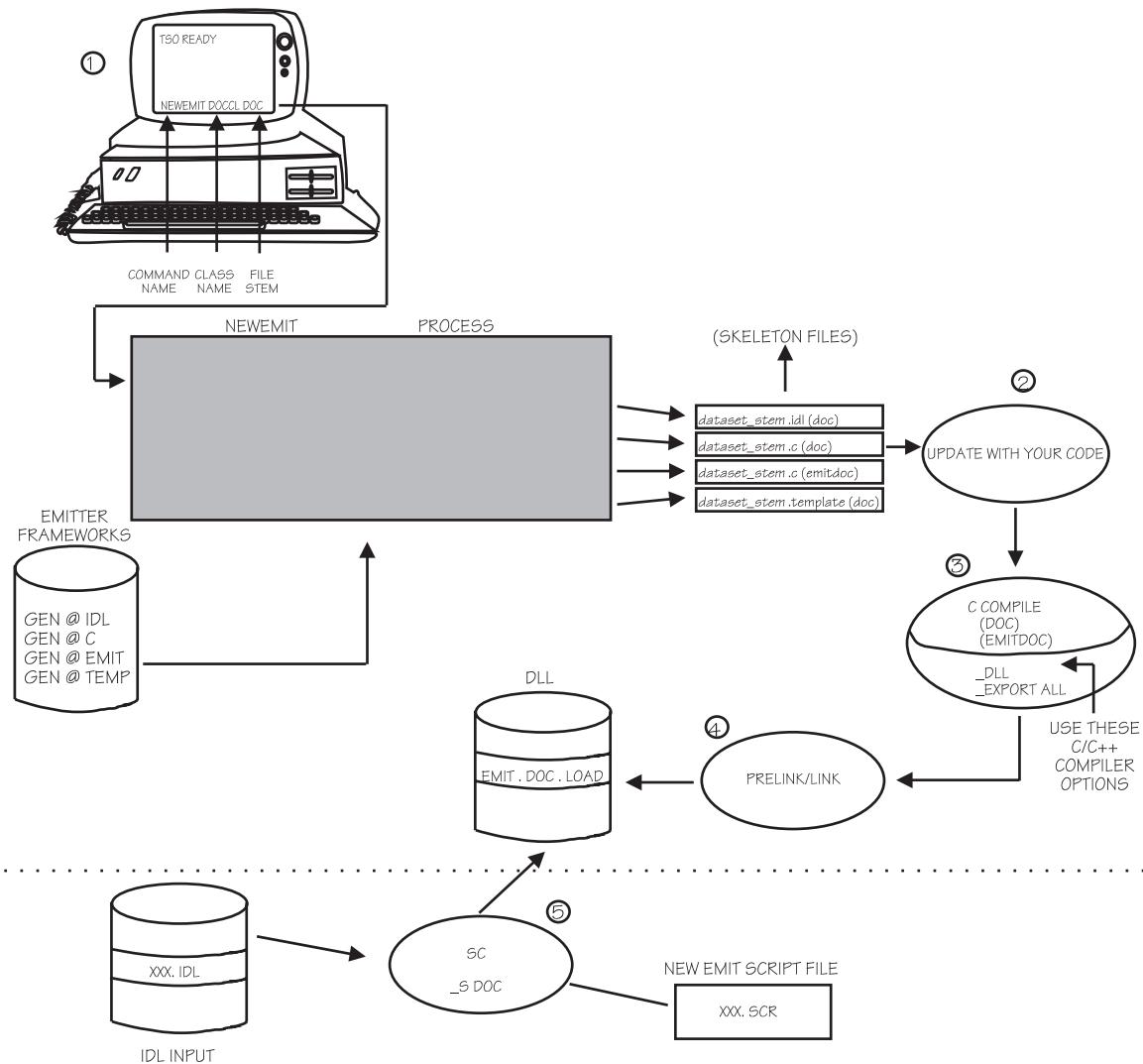


Figure 14-5. The NEWEMIT process.

**Step 1:** Run the NEWEMIT utility from TSO or batch using the following syntax:

`NEWEMIT[-C | -C++ ] className filesystem`

where

*className* is the class name to be used on the interface statement in the IDL stub created by NEWEMIT, and

*filesystem* is the user-specified filesystem that is used as the name for the various members created by NEWEMIT.

The optional -C or -C++ (lowercase is also valid) specifies the language in which the emitter will be written; the default is C.

**Notes:**

1. In order for the NEWEMIT utility to be used, you must first define the [somc] environment variable in your SOM profile. For example, if your [somc] variable defined in your SOM profile were

```
NEWEMITEFW=dataset_stem.test
```

then the following four datasets will be produced by the NEWEMIT utility:

- *dataset\_stem.idl(filestem)*. An IDL definition file.
- *dataset\_stem.c|cxx(filestem)*. A C or C++ implementation file.
- *dataset\_stem.c|cxx(EMITfilestem)*. A C or C++ emitter driver.
- *dataset\_stem.template(filestem)*. A sample template file.

2. The last qualifier defined in your **NEWEMITEFW** environment variable (in this example, the qualifier named "test") is replaced by the above filestem names.

Because there is a MVS limit of 8 bytes to a DSN qualifier (and NEWEMIT uses 4 bytes to add the prefix "EMIT" to the emitter driver member name), you will need to use a dataset name of 4 bytes or less when passing NEWEMIT its second parameter. This IDL definition specifies that the somtGenerateSectionsmethod will be overridden by the new subclass.

**Step 2:** Update the emitter driver program file (add method code) and the implementation file.

Update the sample output template file to create the framework of your output file.

In order to compile your source code, create the following files using the SOMobjects Compiler options (you must also use the *-maddstar* option when using the SOMobjects compiler):

- The C *ih* or C++ *xih* bindings for the class implementers.
- The C *h* or C++ *xh* bindings for the client program.

**Step 3:** Compile the implementation file and emitter driver program file (using the C or C++ compiler) to create the text decks.

**Note:** If you compile the implementation and emitter driver with the C compiler, be sure to specify the DLL and EXPORTALL options.

**Step 4:** Prelink and link the text files together. (Be sure to include an ENTRY CEESTART in your prelink and add the IMPORT data GOSSOMK, GOSSOME, and GOSSOMC in your prelink.)

Prelink and link the files to create the emitter DLL.

**Step 5:** To use a new emitter, run the SOMobjects Compiler against an IDL dataset and invoke the SOMobjects Compiler using the *-s* flag, as in the following example:

```
sc -snew_emitter_name 'dataset_stem.idl(idl_member)'
```

This SOM compiler invocation will use the new emitter specified on the *-s* flag to create a specific set of output files from the input IDL dataset. For example, if you created an emitter named EMITDOC that formats IDL for a word or text processor, then the command

```
sc -sdoc 'dataset_stem.idl(idl_member)'
```

would produce a file that could be processed through a word or text processing program.

## An Example of NEWEMIT in More Detail

This example will discuss the steps of the NEWEMIT process as shown in Figure 14-5 on page 14-20 in greater detail.

### Step 1

The *dataset\_stem.c* (or .C) file 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 member name of *doc*, we would execute the following command:

```
NEWEMIT DocEmitter DOC
```

The NEWEMIT command creates the following datasets:

```
dataset_stem.idl(doc) -- idl for the new emitter
dataset_stem.c(doc) -- The new emitter's implementation file
dataset_stem.c(emitdoc) -- The new emitter's driver program
dataset_stem.efw(doc) -- The new emitter's template.
```

The remaining steps involve customizing the emitter's implementation and driver files.

### Step 2

**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. Given the following IDL specification:

```
#include <somobj.idl>
interface employee : person {
 // Methods
 void calc_pay();
 void hire();
 void fire();
 void promote();
};
```

we would want the output file to look like this:

```
The following methods:
 calc_pay, of type void
 hire, of type void
 fire, of type void
 promote, of type void
are implemented by class Employee.
```

**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 (methods, attributes, and so forth). 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” (such as, classS, methodsS, baseS, and so on). 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 “Employee” 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:

*dataset\_stem.TEMPLATE(member)*

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 `dataset_stem TEMPLATE(DOC)` would be edited to contain the generalized output template shown above.

Note: Each of the entry classes (discussed earlier) defines a set of standard symbols based on the kind of entry they represent. These symbols are discussed in the later topic “Standard Symbols.” Its subtopic “The section-name symbols” 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 the topic “Defining new symbols.”

**Customizing Emitter Control Flow:** The NEWEMIT program creates a subclass of SOMTEmitC (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 the "methods" section for each method of the class.
 * If a "methodsProlog" section is defined, it will precede the
 * first method; and if a "methodsEpilog" section is defined, it
 * will follow the last method.
 */
 _somtScanMethods(somSelf,
 "somtImplemented",
 "somtEmitMethodsProlog",
 "somtEmitMethod",
 "somtEmitMethodsEpilog", 0);
 _somtEmitClass(somSelf); /* emit the "class" section */
 }

 return (TRUE);
}

```

Notice the use of the somtScanMethods method to iterate through the class's methods and emit the "methodsS" section for each method. SOMTEmitC also defines "scanning" methods for data items, base classes, passthru, attributes, constants, typedefs, struct, enums, union, interfaces, and nested modules.

### Steps 3 and 4

**Compiling, prelinking and linking 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.

You need to include an ENTRY CEESTART in your prelink.

You also need to include the following IMPORT data in your prelink step:

- GOSSOMK
- GOSSOME
- GOSSOMC

### Step 5

**Using a New Emitter:** 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. (Note that the value of the -s option is the file stem specified earlier to NEWEMIT.) Invoking the following sc command will produce an employee.doc file just like the one shown at the beginning of this section:

```
sc -sdoc 'dataset_stem.idl(employee)'
```

## Writing an Emitter—Advanced Topics

This section covers the following topics:

- “Defining New Symbols”
- “Customizing Section-emitting Methods” on page 14-29
- “Changing Section Names” on page 14-29
- “Shadowing” on page 14-30
- “Handling Modules” on page 14-31
- “Error Handling” on page 14-31

### Defining New Symbols

The Emitter Framework defines a number of symbols that can be used in output templates. (These are described in the section “Standard Symbols.”) Programmers can also 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. Note: 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,

```
_samtGetSymbol(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 Symbol Definition Methods:** New symbols are defined using one of the following methods, invoked on a template object:

samtSetSymbol,  
samtSetSymbolCopyValue,  
samtSetSymbolCopyName, and  
samtSetSymbolCopyBoth.

Arguments for each method are the name of the symbol and its value (both as strings). The differences among the four symbol-setting methods are whether they make a copy of the name/value to store in the symbol table, or whether the passed strings are stored. If no copy is made, then 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.

**Example of somtSetSymbolCopyBoth:** For example, to set the value of the “NewSym” symbol to value “Payroll”, use the following C method call:

```
_samtSetSymbolCopyBoth(t, "newSym", "Payroll");
```

where *t* is the template object for an emitter. (The somtTemplate attribute of an emitter refers to its template object.)

**Example of somtExpandSymbol:** 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:

```
:metho/PrefixS
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_samtTemplate(somSelf);
char buf[MAX_BUF_SIZE];
...
_samtSetSymbolCopyBoth(t, "methodPrefix",
 _samtExpandSymbol(t, "methodPrefixS", buf));
```

**Defining New Symbols with User-Defined Subclasses of an Entry Class:** 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 (SOMTCClassEntryC, SOMTMethodEntryC, and so forth). In this way, the new symbols will be defined at the same time the *standard* symbols are defined (when the somtSetSymbolsOnEntry method is invoked on an entry object). 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 SOMTCClassEntryC 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 symbol(s).

**Example of a User-Defined Subclass:** 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)
{
 SOMTemplateOutputC t = __get_somtTemplate(emitter);
 long status;
 status = parent_SOMTClassEntryC_somtSetSymbolsOnEntry
 (somSelf, emitter, prefix);
 _samtSetSymbolCopyBoth(t, somtNewSymbol(prefix, "ExtPrefix"),
 _samtGetModifierValue (somSelf, "externalprefix");
 return(status);
}
```

Notice that 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.

**Using the *prefix* parameter of the *somtSetSymbolsOnEntry* Method:** 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 (when it invokes somtSetSymbolsOnEntry on an entry) 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 (as in the above example).

It is important that overriding implementations of somtSetSymbolsOnEntry in subclasses of SOMTClassEntryC in particular use somtNewSymbol and the *prefix* parameter to define new symbols (rather than defining new symbols with a fixed name, such as “classExtPrefix”) because that method will be invoked not only to define symbols for the target class (for which “prefix” will be “class”), but also to define symbols for its base class(es) 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 the later topic entitled “Shadowing.”

## Customizing Section-emitting Methods

The somtGenerateSections method of SOMTEmitC invokes section-emitting methods, such as somtEmitClass and somtEmitMethod. 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 (a subclass of SOMTEmitC) can also 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 (instead of passing somtEmitMethod), so that for each of the target class’s methods, “somtEmitMethod2” is invoked (instead of somtEmitMethod). User-defined sections can also be emitted by changing the value of one of the predefined section-name symbols, as described under the next topic, “Changing section names.”

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 (an instance of SOMTMethodEntryC). 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 somtGetModifier method for obtaining 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. (SN stands for “section name.”) The default value of the “prologSN” symbol is “prologS.” Thus, somtEmitProlog emits the “prologS” section of the output template by default.

Table 14-1 on page 14-39 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 the section “myPrologS” instead of the section “prologS”, invoke the somtSetSymbol method as follows from within the somtGenerateSections method of your emitter, prior to invoking somtEmitProlog:

```
_samtSetSymbolCopyValue(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 (by default) 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 (SOMTClassEntryC, SOMTMethodEntryC, etc.) 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 (rather than creating instances of the original entry class).

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 (stored in *emitdataset name.c*) 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 *emitdataset name.c*).

## Handling Modules

When an emitter is run on a .idl file that contains a module, rather than a single class, the *cls* argument to the emit function in the emitter's driver program will be a structure such that (*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 emitter(s), passing a structure representing the module. It then runs the same emitter(s) again, passing a structure representing the first interface in the module. It then runs the same emitter(s) 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. This includes somtEmitMetaInclude, somtEmitMeta, somtScanBases, somtScanMethods, and so forth.

## 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
- somitinternal.

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.

**Example of Producing an Error Message Using the somterror Function:**

Below is an example of producing an error message using the somterror function (*entry* is an instance of one of the Entry classes, and *cls* is the emitter's target class):

```
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 (the one opened using the somtopenEmitFile function) 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.

**Protecting Critical Portions of Emitter Code:** The Emitter Framework provides two functions, somtunsetEmitSignals and somtresetEmitSignals, to protect 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();
```

See *OS/390 SOMobjects Programmer's Reference, Volume 1* for more information on the error-handling functions provided by the Emitter Framework.

**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 (or when defining an additional, related symbol), as this list indicates which entry class to subclass.

Each symbol is described in more detail in the second part of this section.

**1. 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 also 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
attribute BaseType
attributeComment
attributeLineNumber
attributeMods
```

Valid in the moduleS section:

```
moduleName
moduleIDLScopedName
moduleCScopedName
moduleComment
moduleLineNumber
moduleMods
```

**2. 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. (The somtSetSymbolsOnEntry method can be invoked on an entry object directly. It is also invoked automatically on the target class entry object and on its metaclass entry object by the somtFileSymbols method. It is also invoked automatically on each entry processed by one of the section-emitter or scanning methods of SOMTEmitC.)

#### **For SOMTEentryC**

|                            |                                                         |
|----------------------------|---------------------------------------------------------|
| <i>prefixName</i>          | The unscoped name of the entry.                         |
| <i>prefixIDLScopedName</i> | The scoped name of the entry, using “::” as delimiters. |
| <i>prefixCScopedName</i>   | The scoped name of the entry, using “_” as delimiters.  |

|                         |                                                                        |
|-------------------------|------------------------------------------------------------------------|
| <i>prefixComment</i>    | The comment that follows the entry in the IDL specification.           |
| <i>prefixLineNumber</i> | The line number where the IDL specification of the entry <i>ends</i> . |
| <i>prefixMods</i>       | The SOM IDL modifiers of the entry.                                    |

where *prefix* is replaced by the corresponding IDL syntactic unit being defined, either “module,” “attribute,” “constant,” “typedef,” “struct,” “enum,” “union,” “class,” “base,” “meta,” “method,” “data,” “passthru,” or a user-specified prefix.

#### **For SOMTCommonEntryC**

|                              |                                                           |
|------------------------------|-----------------------------------------------------------|
| <i>prefixType</i>            | The type of the entry. For methods, the return type.      |
| <i>prefixArrayDimensions</i> | The array dimensions of the entry, if it is an array.     |
| <i>prefixPointers</i>        | 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**

|                             |                                                                                             |
|-----------------------------|---------------------------------------------------------------------------------------------|
| <i>attributeDeclarators</i> | The list of attribute declarators.                                                          |
| <i>attributeBaseType</i>    | The base type of the attribute(s), not including pointer stars or array dimensions, if any. |

#### **For SOMTEnumEntryC**

|                  |                                                  |
|------------------|--------------------------------------------------|
| <i>enumNames</i> | The list of enumerator names of the enumeration. |
|------------------|--------------------------------------------------|

#### **For SOMTCClassEntryC**

|                            |                                                                                                                                                                                                                                     |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>classMajorVersion</i>   | The class's major version number.                                                                                                                                                                                                   |
| <i>classMinorVersion</i>   | The class's minor version number.                                                                                                                                                                                                   |
| <i>classSourceFile</i>     | The name of the IDL source file.                                                                                                                                                                                                    |
| <i>classSourceFileStem</i> | The file stem of the binding files to be produced from the input IDL file. If the input IDL has a “dataset name” modifier, then its value defines the symbol. Otherwise, the symbol will be the dataset name of the input IDL file. |
| <i>classReleaseOrder</i>   | The release order list for the class.                                                                                                                                                                                               |
| <i>classInclude</i>        | The expression to be used in include statements to access the appropriate file for this class (such as <somobj.idl>).                                                                                                               |

#### **For SOMTConstantEntryC**

|                                 |                                                                                                                                                                                                             |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>constantType</i>             | The type of the constant.                                                                                                                                                                                   |
| <i>constantValueEvaluated</i>   | The evaluated value of the constant. For constants of type string or char, this value includes the quotes.                                                                                                  |
| <i>constantValueUnevaluated</i> | The unevaluated value of the constant. Constants within the value expression are, however, replaced with their values. For constants of type string or char, this value does <i>not</i> include the quotes. |

#### **For SOMTMethodEntryC**

|                          |                                                                                                                                                                        |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| methodCReturnType        | The C/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 <i>explicit</i> parameters).                                                    |
| methodCParamList         | The formal parameter list (including types) for the method's procedure, in C/C++ form (including all parameters).                                                      |
| methodCParamListVA       | The formal parameter list (including types) for the method's procedure, in C/C++ form (including all parameters), with any <i>va_list</i> parameter replaced by “...”. |
| methodShortParamNameList | A list consisting of the names of the method's explicit parameters (excluding <i>somSelf</i> , <i>ev</i> , and <i>ctx</i> ).                                           |
| methodFullParamNameList  | A list consisting of the names of all of the method's procedure's formal parameters (including implicit method parameters and <i>somSelf</i> ).                        |
| methodRaises             | A list of the exceptions the method may raise.                                                                                                                         |
| methodContext            | A list of the context string literals for the method.                                                                                                                  |

#### **For SOMTPParameterEntryC**

|                         |                                                                                                                                                                          |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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/C++ form. This may differ from the IDL declaration, particularly when the parameter is an out or inout parameter. |

#### **For SOMTPassthruEntryC**

|                  |                                                                 |
|------------------|-----------------------------------------------------------------|
| passthruLanguage | The target language of the passthru, in upper case (e.g., “C”). |
| passthruTarget   | The file type for this passthru; for example, “h”, “ih”         |
| passthruBody     | The full contents of the passthru entry, including newlines.    |

#### **For SOMTSequenceEntryC**

|                |                                                                                 |
|----------------|---------------------------------------------------------------------------------|
| sequenceLength | The maximum length of the sequence, as declared in IDL, or zero if unspecified. |
|----------------|---------------------------------------------------------------------------------|

#### **For SOMTStringEntryC**

|              |                                                                               |
|--------------|-------------------------------------------------------------------------------|
| stringLength | The maximum length of the string, as declared in IDL, or zero if unspecified. |
|--------------|-------------------------------------------------------------------------------|

**For *SOMTTypedefEntryC***

|                    |                                                                                                         |
|--------------------|---------------------------------------------------------------------------------------------------------|
| 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 “section-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, the somtEmitProlog method emits the section whose name is specified by the “prologSN” symbol. (“SN means section name.”) 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 the earlier section “Defining new symbols,” before invoking somtEmitProlog.

Table 14-1 on page 14-39 lists all symbol names, their initial value, and the method that uses them.

*Table 14-1. Section-name symbol names, initial values, and methods that use them.*

| <b>Symbol Name</b>   | <b>Initial Value<br/>(Section Name)</b> | <b>Used by Method</b>      |
|----------------------|-----------------------------------------|----------------------------|
| attributeSN          | attributeS                              | somtEmitAttribute          |
| attributeEpilogSN    | attributeEpilogS                        | somtEmitAttributeEpilog    |
| attributePrologSN    | attributePrologS                        | somtEmitAttributeProlog    |
| atteSN               | 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               |
| metaIncludeSN        | 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        |



## Chapter 15. The Interface Repository Framework

In addition to the Emitter Framework, SOMobjects also provides another collection of classes called the Interface Repository (IR) Framework.

This chapter addresses the following topics:

- What is the Interface Repository (IR) Framework?
- The uses and benefits of the IR Framework
- Understanding the IR Framework
- Using the IR Framework

### What is the Interface Repository (IR) 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 (see Figure 15-1).

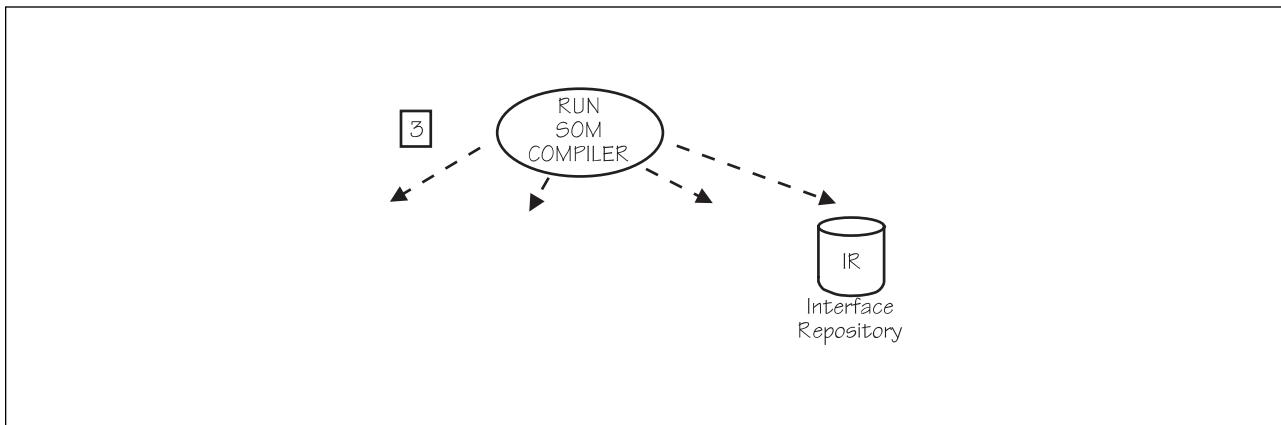


Figure 15-1. IR part of the SOMobjects "Big Picture".

The Interface Repository is a runtime database that contains a machine readable version of the IDL defined interfaces. 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. That is, within the SOM-unique implementation section of an IDL source file or through the use of the #pragma modifier statement, user-defined modifiers can be associated with any element of an IDL specification. (See Chapter 3, “SOMobjects Interface Definition Language (IDL)” on page 3-1.) When the SOM Compiler creates the Interface Repository from an IDL specification, these potentially arbitrary modifiers are stored in the IR and can then be accessed via the methods provided by the Interface Repository Framework. See Figure 15-2 for how the IR Framework is created and accessed.

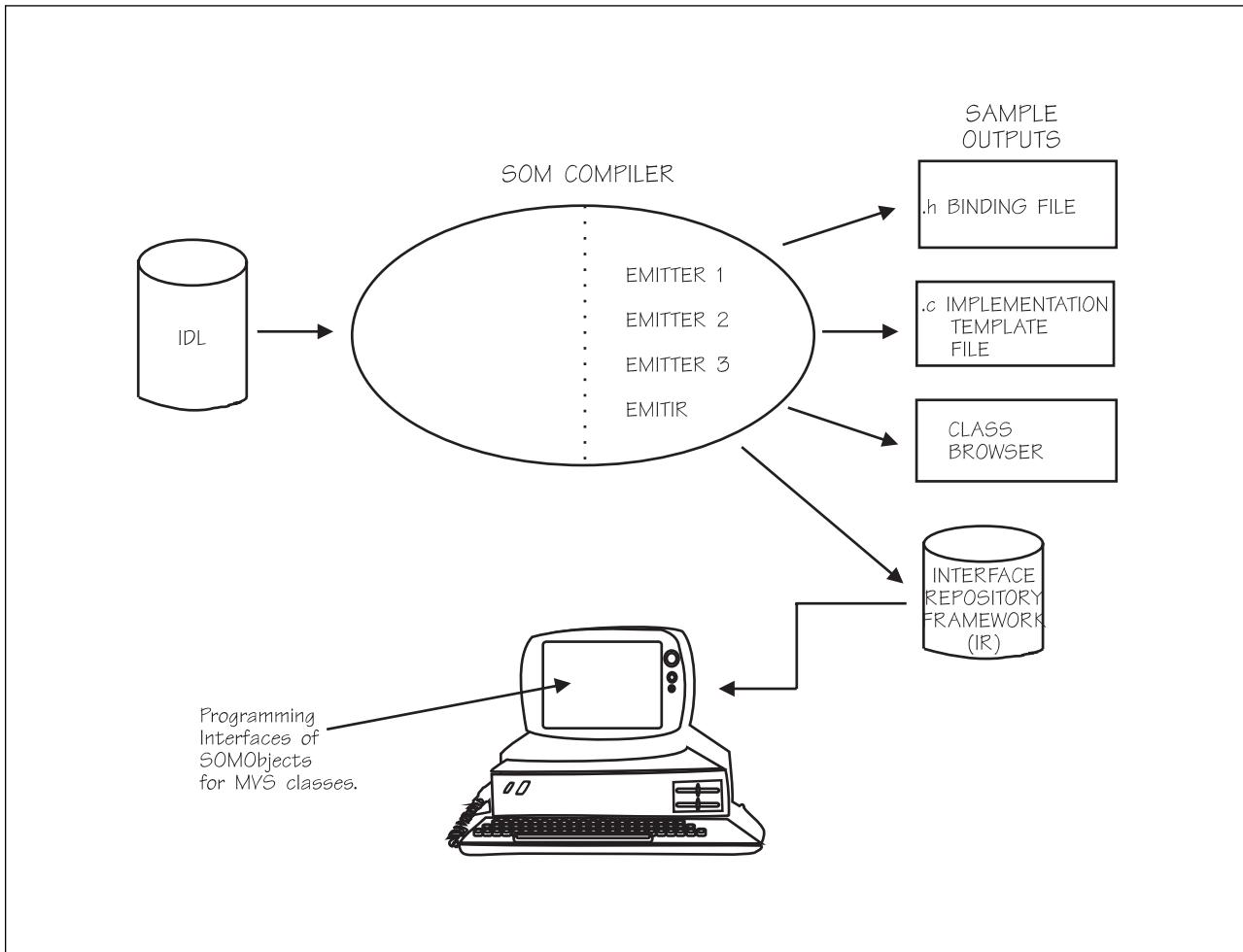


Figure 15-2. Relationship of the IR Framework to the SOMobjects Compiler and the end user.

---

## The Uses and Benefits of the IR Framework

The uses and benefits of the IR Framework are:

- It's needed for cross language support for non-header languages.
- It's required for systems that use dynamic class lookup.

### Needed for Cross Language Support for Non-header Languages

The IR Framework is valuable for cross-language support, especially for non-header languages, such as COBOL. Some languages do not have header (.h) support and still need to access the information that the IR Framework provides.

### Required for Systems that Use Dynamic Class Lookup

The IR is required at run time if the system uses dynamic class lookup. Dynamic class lookup is accomplished by calls to the somFindClass method in SOMClassMgr. The SOM system will look in memory for the requested class name and if not found will go to the IR to find the "dllname" modifier for the class. Any customizable SOM application will require dynamic class lookup.

SACL also adds an "gotShortName" modifier that is required for persistent objects. The SACL also has other modifiers that are used to set by the commit orders for persistent objects when using a DB2 database. SACL also uses the IR to dynamically construct the list of securable methods in their security dialogs.

So the IR provides a mechanism for storing information that can dynamically be retrieved. An application can define new modifiers that are required to use the application (like SACL's "gotShortName").

---

## Understanding the IR Framework

To understand the IR Framework, you need to know about the following:

- Managing IR files
  - The SOM IR file *sommvs.SGOSIR*
  - Managing IRs via the SOMIR environment variable
  - Placing private information in the IR
- Programming with IR objects
  - The IR eleven classes of objects
  - Methods introduced by IR classes
  - A word about memory management

## 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

*sommvs.SGOSIR*

SOMObjects includes an Interface Repository file (*sommvs.SGOSIR*) that contains objects describing all of the types, classes, and methods provided by the various frameworks of SOMObjects. Since all new classes will ultimately be derived from these predefined SOM classes, some of this information also needs to be included in a programmer's own interface repository files.

For example, suppose a new class, called "MyClass", is derived from SOMObject. When the SOM Compiler builds an Interface Repository for "MyClass", that IR will also include all of the information associated with the SOMObject class. This happens because the SOMObject class definition is inherited by each new class; thus, all of the SOMObject methods and typedefs are implicitly contained in the new class as well.

Eventually, the process of deriving new classes from existing ones would lead to a great deal of duplication of information in separate interface repository files. This would be inefficient, wasteful of space, and extremely difficult to manage. For example, to make an evolutionary change to some class interface, a programmer would need to know about and subsequently update all of the interface repository files where information about that interface occurred.

One way to avoid this dilemma would be to keep all interface definitions in a single interface repository (such as *sommvs.SGOSIR*). This is not recommended because a single interface repository would soon grow to be unwieldy in size and become a source of frequent access contention. Everyone involved in developing class definitions would need update access to this one file, and simultaneous uses might result in longer compile times.

## Managing IRs via the SOMIR Environment Variable

SOMobjects offers a more flexible approach to managing interface repositories. The SOMIR environment variable can reference an ordered list of separate IR files, which process from left to right. Taken as a whole, however, this gives the appearance of a single, logical interface repository. A programmer accessing the contents of “the interface repository” through the SOM Interface Repository framework would not be aware of the division of information across separate files. It would seem as though all of the objects resided in a single interface repository file.

A typical way to utilize this capability is as follows:

- The first (leftmost) Interface Repository in the SOMIR list would be *sommvs.SGOSIR*. This file contains the basic interfaces and types needed in all SOM classes.
- The second file in the list might contain interface definitions that are used globally across a particular enterprise.
- A third interface repository file would contain definitions that are unique to a particular department, and so on.
- The final interface repository in the list should be set aside to hold the interfaces needed for the project currently under development.

Developers working on different projects would each set their SOMIR environment variables to hold slightly different lists. For the most part, the leftmost portions of these lists would be the same, but the rightmost interface repositories would differ. When any given developer is ready to share his/her interface definitions with other people outside of the immediate work group, that person's interface repository can be promoted to inclusion in the master list.

With this arrangement of IR files, the more stable repositories are found at the left end of the list. For example, a developer should never need to make any significant changes to *sommvs.SGOSIR*, because these interfaces are defined by IBM and would only change with a new release of SOMobjects.

The Interface Repository 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 (that is, further to the right) interface repository 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 files have been installed. Only the “myown.ir” interface repository file will be updated with the interfaces found in *MYOWN.IDL* with member names “myclass1”, “myclass2”, and “myclass3”.

The MVS environment file is used to set up the SOM IR data set names. The following is an example of the environment file:

```
[somir]
SOMIR_DUMPALL=1
SOMIR='//SOM2.BASE.IR';//SOM2.MYOWN.IR';
[somc]
SMINCLUDE='//SOM2.BASE';//SOM2.MYOWN';
```

The following SOM Compiler commands can be entered from the TSO Ready to access the MVS environment file:

```
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.

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 Interface Repository Objects

The SOM Interface Repository Framework provides an object-oriented programming interface to the IDL information processed by the SOM Compiler. Unlike many frameworks that require you to inherit their behavior in order to use it, the Interface Repository Framework is useful in its own right as a set of predefined objects that you can access to obtain information. Of course, if you need to subclass a class to modify its behavior, you can certainly do so; but typically this is not necessary.

### The IR Eleven Classes of Objects

Figure 15-3 shows the set of classes that comprise the IR framework.

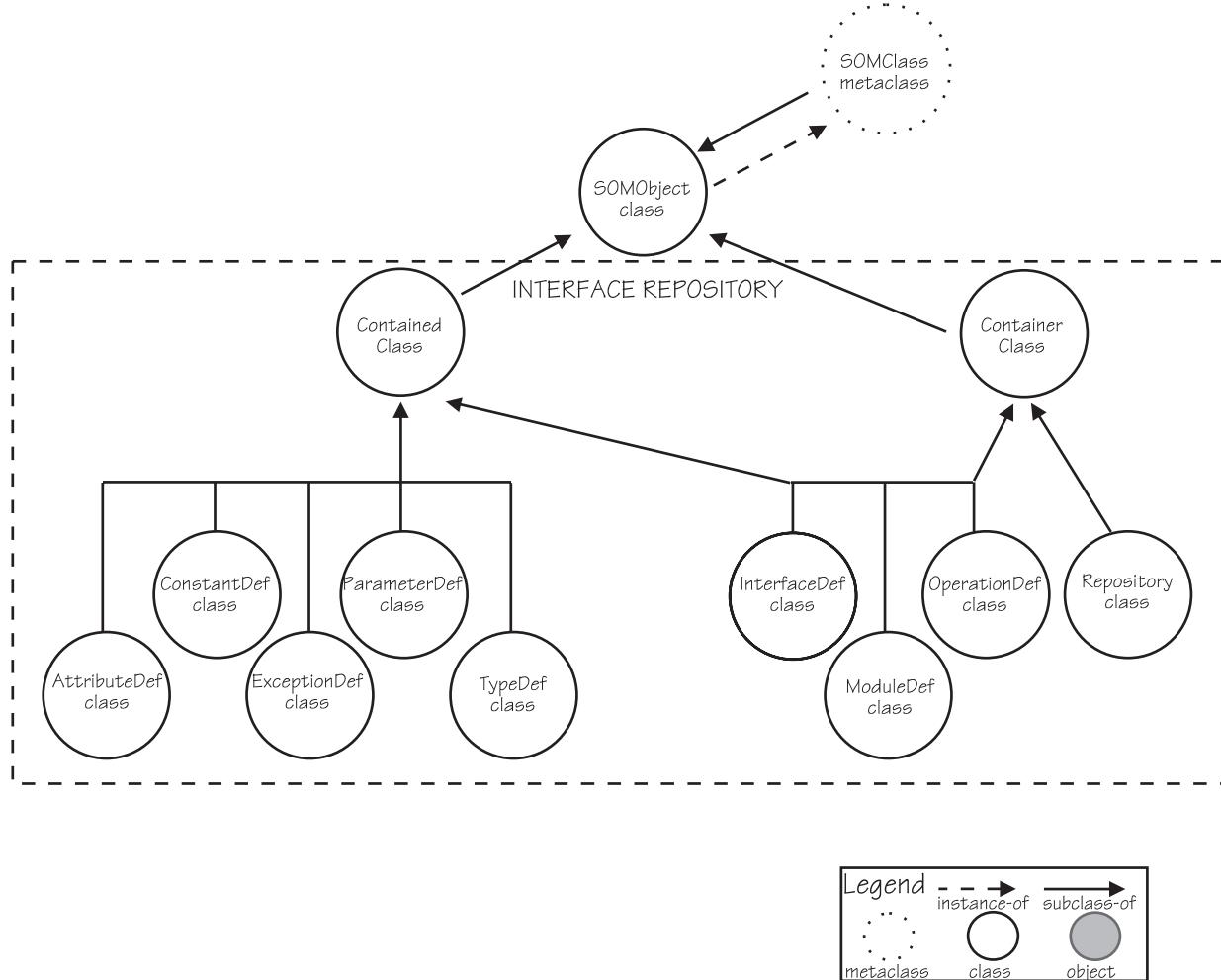


Figure 15-3. Interface Repository Framework classes

The SOM Interface Repository contains the fully-analyzed (compiled) contents of all information in an IDL source file. This information takes the form of objects that can be accessed from a running program. There are eleven 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) defined 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.

## **Methods introduced by IR classes**

The Interface Repository classes introduce nine new methods, which are briefly described below. Many of the classes simply override methods to customize them for the corresponding IDL element; this is particularly true for classes representing IDL elements that are only contained within another syntactic element.

**Contained class methods** (all IR objects are instances of this class and exhibit this behavior):

**describe**

Returns a structure of type **Description** containing all information defined in the IDL specification of the syntactic element corresponding to the target **Contained** object. For example, for a target **InterfaceDef** object, the **describe** method returns information about the IDL interface declaration. The **Description** structure contains a “name” field with an identifier that categorizes the description (such as, “InterfaceDescription”) and a “value” field holding an “any” structure that points to another structure containing the IDL information for that particular element (in this example, the interface’s IDL specifications).

**within**

Returns a sequence designating the object(s) of the IR within which the target **Contained** object is contained. For example, for a target **TypeDef** object, it might be contained within any other IR object(s) except an **OperationDef** object.

**Container class methods** (some IR objects contain other objects and exhibit this behavior):

**contents**

Returns a sequence of pointers to the object(s) of the IR that the target **Container** object contains. (For example, for a target **InterfaceDef** object, the **contents** method returns a pointer to each IR object that corresponds to a part of the IDL interface declaration.) The method provides options for excluding inherited objects or for limiting the search to only a specified kind of object (such as **AttributeDefs**).

**describe\_contents**

Combines the **describe** and **contents** methods; returns a sequence of **ContainerDescription** structures, one for each object contained by the target **Container** object. Each structure has a pointer to the related object, as well as “name” and “value” fields resulting from the **describe** method.

**lookup\_name**

Returns a sequence of pointers to objects of a given name contained within a specified **Container** object, or within (sub)objects contained in the specified **Container** object.

- **ModuleDef** class methods:

Override **describe** and **within**.

- **InterfaceDef** class methods:

**describe\_interface**

Returns a description of all methods and attributes of a given interface definition object that are held in the Interface Repository.

Also overrides **describe** and **within**.

- AttributeDef class method:

Overrides **describe**.

- OperationDef class method:

Overrides **describe**.

- ParameterDef class method:

Overrides **describe**.

- TypeDef class method:

Overrides **describe**.

- ConstantDef class method:

Overrides **describe**.

- ExceptionDef class method:

Overrides **describe**.

- Repository class methods:

#### **lookup\_id**

Returns the **Contained** object that has a specified **RepositoryId**.

#### **lookup\_modifier**

Returns the string value held by a SOM or user-defined **modifier**, given the name and type of the modifier, and the name of the object that contains the modifier.

#### **release\_cache**

Releases, from the internal object cache, the storage used by all currently unreferenced Interface Repository objects.

## 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.

**Attribute Access:** 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.

**Object References:** For all methods that give out object references (there are five: **within**, **contents**, **lookup\_name**, **lookup\_id**, and **describe\_contents**), when finished with the object, you are expected to release the object reference by

invoking the somFree method. (This is illustrated in the sample program that accesses all Interface Repository objects.) Do not release the object reference until you have either copied or finished using all of the information obtained from the object.

**Describe Methods:** The **describe** methods (**describe**, **describe\_contents**, and **describe\_interface**) return structures and sequences that contain information. The actual structures returned by these methods are passed by value (and therefore 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.

**Intermediate Objects:** 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.

---

## Using the IR Framework

To use the IR Framework, you need to know about the following:

- Using the SOMobjects Compiler to build an IR
- Accessing objects in the IR
  - “Using TypeCode pseudo-objects” on page 15-18
  - “Example of a TypeCode as a Pseudo-Object” on page 15-19
  - “Providing ‘Alignment’ Information” on page 15-21
  - “Using the ‘tk\_foreign’ TypeCode” on page 15-22
  - “TypeCode Constants” on page 15-23
  - “Using the IDL Basic Type ‘Any’” on page 15-23
  - “Building an Index for the Interface Repository” on page 15-24

## Using the SOM Compiler to Build an Interface Repository

SOMobjects includes an Interface Repository emitter that is invoked whenever the SOM Compiler is run using an sc command with the -u option (which “updates” the interface repository). The IR emitter can be used to create or update an Interface Repository file. The IR emitter expects that an environment variable, SOMIR, was first set to designate a file name for the Interface Repository. For example, to compile an IDL source file named “newcls.idl” and create an Interface Repository named “newcls.ir”, use a command sequence similar to the following:

```
sc -usir newcls
```

If the SOMIR environment variable is not set, the Interface Repository emitter creates a file named “<userid>.sommvsSGOSIR” in the current directory, where <userid> is the userid of the user invoking the SC.

The sc command runs the Interface Repository emitter plus any other emitters indicated by the environment variable SMEMIT (described in Chapter 4, “SOMobjects Compiler” on page 4-1) To run the Interface Repository emitter by itself, issue the sc command 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.

## Accessing Objects in the Interface Repository

As mentioned previously, one instance of the Repository class exists for the entire SOM Interface Repository. This object does not, itself, reside in the Interface Repository (therefore it does not exhibit any of the behavior defined by the **Contained** class). It is, however, a **Container**, and it holds all **ConstantDefs**, **TypeDefs**, **ExceptionDefs**, **InterfaceDefs**, and **ModuleDefs** 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.

**Accessing Similarly Named TypeDef Objects:** For example, two **TypeDef** objects may have the same name, provided they occur in separate name scopes (**ModuleDefs** or **InterfaceDefs**). 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.

**Obtaining an Interface of the Repository Class:** C or C++ programmers can obtain an instance of the Repository class using the **RepositoryNew** macro. Programmers using other languages (and C/C++ programmers without static linkage to the Repository class) should invoke the method **somGetInterfaceRepository** on the **SOMClassMgrObject**. For example,

For C or C++ (static linkage):

```
#include <repository.h>
Repository repo;

repo = RepositoryNew();
```

From other languages (and for dynamic linkage in C/C++):

1. Use the **somEnvironmentNew** function to obtain a pointer to the **SOMClassMgrObject**, as described in “**SOMClassMgr Class Object and SOMClassMgrObject**” on page 2-3.
2. Use the **somResolve** or **somResolveByName** function to obtain a pointer to the **somGetInterfaceRepository** method procedure.
3. Invoke the method procedure on the **SOMClassMgrObject**, with no additional arguments, to obtain a pointer to the Repository object.

**Using Container Methods to Instantiate Objects in the IR:** After obtaining a pointer to the Repository object, use the methods it inherits from **Container** or its own **lookup\_id** method to instantiate objects in the Interface Repository. As an example, the **contents** method shown in the C fragment below activates every object with global scope in the Interface Repository and returns a sequence containing a pointer to every global object:

```
#include <containd.h> /* Behavior common to all
IR objects */
Environment *ev;
int i;
sequence(Contained) everyGlobalObject;

ev = SOM_CreateLocalEnvironment(); /* Get an environment to use */
printf ("Every global object in the Interface Repository:\n");

everyGlobalObject = Container_contents (repo, ev, "all", TRUE);

for (i=0; i < everyGlobalObject._length; i++) {
 Contained aContained;

 aContained = (Contained) everyGlobalObject._buffer[i];
 printf ("Name: %s, Id: %s\n",
 Contained_get_name (aContained, ev),
 Contained_get_id (aContained, ev));
 SOMObject_somFree (aContained);
}
```

**Accessing Every Object in the Entire IR:** Taking this example one step further, here is a complete program that accesses every object in the entire Interface Repository. It, too, uses the **contents** method, but this time recursively calls the **contents** method until every object in every container has been found:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "irbase.h"
#include "repository.h"
#include "containr.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 *OS/390 SOMobjects Programmer's Reference, Volume 1*.

**JCL to Dump One Class out of the “Payroll” Class Library:** The JCL in Figure 15-4 will dump the *Employee* class from the “Payroll” Class Library.

```
//IRDUMP JOB <job card parameters>
//*
//COMP EXEC PGM=IRDUMP,REGION=40M,PARM=' -o'
//STEPLIB DD DSN=&SOMPRFX..SGOSLOAD,DISP=SHR
// DD DSN=&LEPRFX..SCEERUN,DISP=SHR
//SOMENV DD DSN=&SOMPRFX..SGOSPROF(SGOSENV1),DISP=SHR
//SYSPRINT DD DSN=DSOMMVS.JCSOUTPUT.DATA,DISP=SHR
```

Figure 15-4. JCL to dump the IR output.

The IRDUMP utility can be used from the OpenEdition Shell to display the contents of interface repositories that reside in HFS files.

**Results of Dumping One Class out of the “Payroll” Class Library:**

Figure 15-5 on page 15-17 displays a subset of what the *Employee* class of “Payroll” would look like if you dumped the IR with the above JCL.

```

"payroll": 1 entry found

1 ModuleDef "payroll"
 id: ::payroll
 contains 7 items:

2 InterfaceDef "check"
 id: ::payroll::check
 2 modifiers:
 releaseorder = _get_pay_to_name,_set_pay_to_name,_get_amount,_set_amount,print_check
 filestem = EMPLOYEE
 1 base interface:
 "::SOMObject"
 instanceData: TypeCodeNew (tk_struct, "checkData",
 "pay_to_name", TypeCodeNew (tk_string, 0),
 "amount", TypeCodeNew (tk_float),
 NULL)
 contains 7 items:

3 OperationDef "_get_pay_to_name"
 id: ::payroll::check::_get_pay_to_name
 1 modifier:
 attribute
 mode: NORMAL
 result: TypeCodeNew (tk_string, 0)
3 OperationDef "_set_pay_to_name"
 id: ::payroll::check::_set_pay_to_name
 1 modifier:
 attribute
 mode: NORMAL
 result: TypeCodeNew (tk_void)
 contains 1 item:

4 ParameterDef "pay_to_name"
 id:
 ::payroll::check::_set_pay_to_name::pay_to_name
 mode: IN
 type: TypeCodeNew (tk_string, 0)
3 OperationDef "_get_amount"
 id: ::payroll::check::_get_amount
 1 modifier:
 attribute
 mode: NORMAL
 result: TypeCodeNew (tk_float)
3 OperationDef "_set_amount"
 id: ::payroll::check::_set_amount
 1 modifier:
 attribute
 mode: NORMAL
 result: TypeCodeNew (tk_void)
 contains 1 item:

4 ParameterDef "amount"
 id: ::payroll::check::_set_amount::amount
 mode: IN
 type: TypeCodeNew (tk_float)

3 OperationDef "print_check"
 id: ::payroll::check::print_check
 mode: NORMAL
 result: TypeCodeNew (tk_void)

```

*Figure 15-5 (Part 1 of 2). IR dump output.*

```

3 AttributeDef "pay_to_name"
 id: ::payroll::check::pay_to_name
 mode: NORMAL
 type: TypeCodeNew (tk_string, 0)

3 AttributeDef "amount"
 id: ::payroll::check::amount
 mode: NORMAL
 type: TypeCodeNew (tk_float)

```

Figure 15-5 (Part 2 of 2). IR dump output.

## 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.

**TypeCode “kind” Fields:** TypeCode “kind” fields come in the following data types:

- long
- struct
- union
- enum
- sequence
- array
- objref

Conceptually, every **TypeCode** contains a “kind” field (which classifies it), and one or more parameters that carry descriptive information appropriate for that particular category of **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.)

The **tk\_objref TypeCode** represents an object reference; its parameter is a repository ID that identifies its interface.

**Parameter of All TypeCodes:** A complete table showing the parameters of all possible **TypeCodes** is given in the *OS/390 SOMobjects Programmer's Reference, Volume 1*; see the **TypeCode\_kind** function of the Interface Repository Framework.

**TypeCodes as Pseudo-Objects:** **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 (for example, they can be copied and freed).

### Example of a TypeCode as a Pseudo-Object

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 ostensible similarity to an IDL class, the **TypeCode** programming interfaces can be conveniently defined in IDL as shown in Figure 15-6 on page 15-20.

```

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,
 };

 exception Bounds {};
 // This exception is returned if an attempt is made
 // by the parameter() operation (described below) to
 // access more parameters than exist in the receiving
 // TypeCode.

 boolean equal (in TypeCode tc);
 // Compares the argument with the receiver and returns TRUE
 // if both TypeCodes are equivalent. This is NOT a test for
 // identity.

 TCKind kind ();
 // Returns the type of the receiver as a TCKind.

 long param_count ();
 // Returns the number of parameters that make up the
 // receiving TypeCode.

 any parameter (in long index) raises (Bounds);
 // Returns the indexed parameter from the receiving TypeCode.
 // Parameters are indexed from 0 to param_count()-1.
 //
 // The remaining operations are SOM-unique extensions.
 //

 short alignment ();
 // This operation returns the alignment required for an instance
 // of the type described by the receiving TypeCode.
 TypeCode copy (in TypeCode tc);
 // This operation returns a copy of the receiving TypeCode.

 void free (in TypeCode tc);
 // This operation frees the memory associated with the
 // receiving TypeCode. Subsequently, no further use can be
 // made of the receiver, which, in effect, ceases to exist.

 void print (in TypeCode tc);
 // This operation writes a readable representation of the
 // receiving TypeCode to stdout. Useful for examining
 // TypeCodes when debugging.

 void setAlignment (in short align);
 // This operation overrides the required alignment for an
 // instance of the type described by the receiving TypeCode.
 long size (in TypeCode tc);
 // This operation returns the size of an instance of the
 // type represented by the receiving TypeCode.
 };
}

```

*Figure 15-6. TypeCode interfaces defined within IDL.*

## 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 (that is, unaligned or no alignment) is attached to the struct “abc” and to one particular instance of struct “def” (the instance data item “y”). The following is an example of “align=N” modifier:

```
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;
 };
};
```

### ***Make Certain that the TypeCode Going Into the IR Reflects Your Compiler Alignment:***

**Alignment:** Be aware that assigning the required alignment information to a type does *not* guarantee that instances of that type will actually be aligned as indicated. To ensure that, you must find a way to instruct your compiler to provide the desired alignment. In practice, this can be difficult except in simple cases. Most compilers can be instructed to treat all data as aligned (that is, default alignment) or as unaligned, by using a compile-time option or #pragma. The more important consideration is to make certain that the **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.

**Using Unaligned Instance Data:** If you wish to use unaligned instance data when implementing a class, place an “unattached” align=1 modifier in the implementation section. An unattached align=N modifier is presumed to pertain to the class's instance data structure, and will by implication be attached to all of the instance data items.

When designing your own public types, be aware that the best practice of all (and the one that offers the best opportunity for language neutrality) is to lay out your types carefully so that it will make no difference whether they are compiled as aligned or unaligned!

### Using the ‘tk\_foreign’ TypeCode

TypeCodes can be used to partially describe types that cannot be described in IDL (for example, a FILE type in C, or a specific class type in C++). The SOM-unique extension **tk\_foreign** is used for this purpose. A **tk\_foreign TypeCode** contains three parameters:

1. The name of the type,
2. An implementation context string, and
3. A length.

The implementation context string can be used to carry an arbitrarily long description that identifies the context where the foreign type can be used and understood. If the length of the type is also known, it can be provided with the length parameter. If the length is not known or is not constant, it should be specified as zero. If the length is not specified, it will default to the size of a pointer. A **tk\_foreign TypeCode** can also have alignment information specified, just like any other **TypeCode**.

Using the following steps causes the SOM Compiler to create a foreign **TypeCode** in the Interface Repository:

1. Define the foreign type as a **typedef SOMFOREIGN** in the IDL source file.
2. Use the **#pragma modifier** statement to supply the additional information for the **TypeCode** as modifiers. The implementation context information is supplied using the “impctx” modifier.
3. Compile the IDL file using the **-u** option to place the information in the Interface Repository.

For example:

```
typedef SOMFOREIGN Point;
#pragma modifier Point: impctx="C++ Point class",length=12,align=4;
```

If a foreign type is used to define instance data, structs, unions, attributes, or methods in an IDL source file, it is your responsibility to ensure that the implementation and/or usage bindings contain an appropriate definition of the type that will

satisfy your compiler. You can use the **passthru** statement in your IDL file to supply this definition. However, it is *not* recommended that you expose foreign data in attributes, methods, or any of the public types, if this can be avoided, because there is no guarantee that appropriate usage binding information can be provided for all languages. If you know that all users of the class will be using the same implementation language that your class uses, you may be able to disregard this recommendation.

### TypeCode Constants

**TypeCodes** are actually available in two forms: In addition to the **TypeCode** information provided by the methods of the Interface Repository, **TypeCode** constants can be generated by the SOM Compiler in your C or C++ usage bindings upon request. A **TypeCode** constant contains the same information found in the corresponding IR **TypeCode**, but has the advantage that it can be used as a literal in a C or C++ program anywhere a normal **TypeCode** would be acceptable.

**TypeCode** constants have the form **TC\_ typename**, where *typename* is the name of a type (that is, a **typedef**, **union**, **struct**, or **enum**) that you have defined in an IDL source file. In addition, all IDL basic types and certain types dictated by the OMG CORBA standard come with pre-defined **TypeCode** constants (such as **TC\_long**, **TC\_short**, **TC\_char**, and so on). A full list of the pre-defined **TypeCode** constants can be found in the file “**somcnst.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 “**tccconsts**” modifier with the **-m** option of the **sc** 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**”.

### 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 data 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 data at the time of the access.

**Example of the IDL Basic Type ‘Any’:** Here is an example of a code fragment written in C that illustrates how the casting must be done to extract various values from an **any**:

```
#include <som.h> /* For "any" & "Environment" typedefs */
#include <somtc.h> /* For TypeCode_kind prototype */

any result;
Environment *ev;

printf ("result._value = ");
switch (TypeCode_kind (result._type, ev)) {

 case tk_string:
 printf ("%s\n", *((string *) result._value));
 break;

 case tk_long:
 printf ("%ld\n", *((long *) result._value));
 break;

 case tk_boolean:
 printf ("%d\n", *((boolean *) result._value));
 break;

 case tk_float:
 printf ("%f\n", *((float *) result._value));
 break;

 case tk_double:
 printf ("%f\n", *((double *) result._value));
 break;

 default:
 printf ("something else!\n");
}
```

Note: Of course, an **any** has no restriction, per se, on the type of 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. 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.

**irindex [-f] [irFileName1, irFileName2 &ellipsis.]**

**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.

The name of the index file is built by adding *.irinxxxx.ndx* to the existing Interface Repository name, where xxxx is some random number between 1 and 9999 inclusive.

If you specify the **-f** option as the first option, the index names of any IR files are listed and 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.

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.

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 *filename*** 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**.

The following are examples of the **irindex** command:

1. To create an index for **som.ir**:

```
irindex //som.ir
```

2. To create indexes for files in the **SOMIR** environment variable, in the respective directories:

```
irindex
```

If **SOMIR** is set to `//som.test.ir;//som.ir`, then indexes `som.test.ir.irnnnn.ndx` and `som.ir.irnnnn.ndx` are created.

To list the name of the index for **som.ir**, if one exists:

```
irindex -f //som.ir
```

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
```

To list the names of the index of each of the two **ir** files, if the indexes exist:

```
irindex -f -u-som-test.ir -u-som.ir
```

---

## Chapter 16. Event Management Framework

This chapter addresses the following topics:

- What is the Event Manager Framework?
- The Uses and Benefits of the Event Manager Framework
- Understanding the Event Manager Framework
- Using the Event Manager Framework

---

### What is the Event Manager Framework?

The **Event Management Framework** is a central facility for registering all events of an application. Such a registration facilitates grouping of various application events and waiting on multiple events in a single event-processing loop. This facility is used by DSOM to wait on their respective events of interest. The Event Management Framework must also be used by any interactive application that contains DSOM objects.

---

### Understanding the Event Manager Framework

Understanding the Event Manager Framework consists of the following topics:

- Event Management Basics
- Event Manager Advanced Topics
- Limitations

### Event Management Basics

The Event Management Framework consists of an Event Manager (EMan) class, a Registration Data class and several Event classes. It provides a way to organize various application events into groups and to process all events in a single event-processing loop. The need for this kind of facility is seen very clearly in interactive applications that also need to process some background events (say, messages arriving from a remote process). Such applications must maintain contact with the user while responding to events coming from other sources.

Event Manager basics consists of the following topics:

- Model of EMan usage
- Event types
- Registration
- Unregistering for events
- An example callback proc
- Generating client events
- Example of using a timer event
- Processing events
- Interactive applications

## Model of EMan usage

The programming model of EMan is similar to that of many GUI toolkits. The main program initializes EMan and then registers interest in various types of events. The main program ends by calling a non-returning function of EMan that waits for events and dispatches them as and when they occur. In short, the model includes steps that:

1. Initialize the Event Manager,
2. Register with EMan for all events of interest, and
3. Hand over control to EMan to loop forever and to dispatch events.

The Event Manager is a SOM object and is an instance of the **SOMEEMan** class. Since any application requires only one instance of this object, the **SOMEEMan** class is an instance of the **SOMMSingleInstance** class. Creation and initialization of the Event Manager is accomplished by a function call to **SOMEEmaNew**.

Currently, EMan supports the four kinds of events described in “Event types.” An application can register or unregister for events in a callback routine (explained below) even after control has been turned over to EMan.

## Event types

Event types are categorized as follows:

- **Timer events**  
These can be either one-time timers or interval timers.
- **Sink events** (sockets and file descriptors)
- **Client events** (any event that the application wants to queue with EMan)  
These events are defined, created, processed, and destroyed by the application. EMan simply acts as a place to queue these events for processing. EMan dispatches these client events whenever it sees them. Typically, this happens immediately after the event is queued.
- **Work procedure** events (procedures that can be called when there is no other event)  
These are typically background procedures that the application intends to execute when there are spare processor cycles. When there are no other events to process, EMan calls all registered work procedures.

A work procedure event is called only after all other higher priority events have been called. If **someProcessEvent** method is invoked and there are no other events to be processed, then a work procedure event is not called. If **someProcessEvent** is invoked, then the registered work procedure events will run regardless of whether an event of the requested type was ready to be handled.

The EMan is extendible (that is, other event types can be added to it) through subclassing. The event types currently supported by EMan are at a sufficiently low level so as to enable building other higher level application events on top of them. For example, you can build an X-event handler by simply registering the file descriptor for the X connection with EMan and getting notified when any X-event occurs.

## Registration

This topic illustrates how to register for an event type and consist of the following:

- Callback
- Event classes
- EMan parameters
- Registering for events

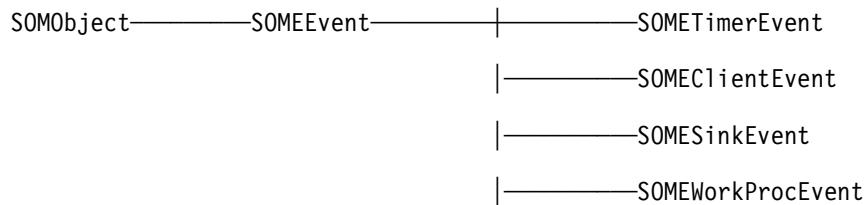
**Callback:** The programmer decides what processing needs to be done when an event occurs and then places the appropriate code either in a procedure or in a method of an object. This procedure or method is called a *callback*. (The callback is provided to EMan at the time of registration and is called by EMan when a registered event occurs.) The signature of a callback is fixed by the framework and must have one of the following three signatures:

```
void EMRegProc(SOMEEvent, void *);
void SOMLINK EMMMethodProc(SOMObject, SOMEEvent, void *);
void SOMLINK EMMMethodProcEv(SOMObject, Environment *Ev,
 SOMEEvent, void *);
```

The three specified prototypes correspond to a simple callback procedure, a callback method using OIDL call style, and a callback method using IDL call style. The parameter type **SOMEEvent** refers to an event object passed by EMan to the callback. Event objects are described below.

NOTE: When the callbacks are methods, EMan calls these methods using **Name-lookup Resolution** (see “Name-lookup Resolution” on page 11-30). One of the implications is that at the time of registration EMan queries the target object’s class object to provide a method pointer for the method name supplied to it. Eman uses this pointer for making event callbacks.

**Event Classes:** All event objects are instances of either the **SOMEEvent** class or a subclass of it. The hierarchy of event classes is as follows:



When called by EMan, a callback expects the appropriate event instance as a parameter. For example, a callback registered for a timer event expects a **SOMETimerEvent** instance from EMan.

**EMan Parameters:** Several method calls in the Event Management Framework make use of bit masks and constants as parameters (for example, **EMSINKEvent** or **EMINPUTREADMASK**). These methods are defined in the include file “eventmsk.h”. When a user plans to extend the Event Management Framework, care must be taken to avoid name and value collisions with the definitions in “eventmsk.h”. For convenience, the contents of the “eventmsk.h” file are shown below.

```

#ifndef H_EVENTMASKDEF
#define H_EVENTMASKDEF

/* Event Types */
#define EMTimerEvent 54
#define EMSignalEvent 55
#define EMSinkEvent 56
#define EMWorkProcEvent 57
#define EMCClientEvent 58

/* Sink input/output condition mask */

#define EMInputReadMask (1L<0)
#define EMInputWriteMask (1L<1)
#define EMInputExceptMask (1L<2)

/* Process Event mask */

#define EMProcessTimerEvent (1L<0)
#define EMProcessSinkEvent (1L<1)
#define EMProcessWorkProcEvent (1L<2)
#define EMProcessClientEvent (1L<3)
#define EMProcessAllEvents (1L<6)

#endif /* H_EVENTMASKDEF */

```

## Registering for Events

In addition to the event classes, the Event Management Framework uses a registration data class (**SOMEEMRegisterData**) to capture all event-related registration information. The procedure for registering interest in an event is as follows:

1. Create an instance of the **SOMEEMRegisterData** class (this will be referred to as a “RegData” object).
2. Set the event type of “RegData.”
3. Set the various fields of “RegData” to supply information about the particular event for which an interest is being registered.
4. Call the registration method of EMan, using “RegData” and the callback method information as parameters. The callback information varies, depending upon whether it is a simple procedure, a method called using OIDL call style, or a method called using IDL call style.

The following code segment illustrates how to register input interest in a socket “sock” and provide a callback procedure “ReadMsg”.

```

data = SOMEEMRegisterDataNew(); /* create a RegData object */
_someClearRegData(data, Ev);
_someSetRegDataEventMask(data,Ev,EMSinkEvent,NULL); /* Event type */
_someSetRegDataSink(data, Ev, sock); /* provide the socket id */
_someSetRegDataSinkMask(data,Ev, EMInputReadMask);
 /*input interest */
regId = _someRegisterProc(some_gEMan,Ev,data,ReadMsg,“UserData”);
/* some_gEMan points to EMan. The last parameter “userData” is any
 data the user wants to be passed to the callback procedure as a
 second parameter */

```

## Troubleshooting Hint

EMan processes new event registrations correctly for a while, but then it seems to ignore all new registrations and client events; for example, enqueued client events are not delivered to the callbacks. If this occurs, you probably called **someUnRegister** with an illegal value, perhaps 0. To resolve this problem, check the source and make sure that all calls to **someUnRegister** are made with registration ids previously returned by the Event Manager.

## Unregistering for Events

One can unregister interest in a given event type at any time. To unregister, you must provide the registration id returned by EMan at the time of registration. Unregistering a non-existent event (such as, an invalid registration id) is a no-op. The following example unregisters the socket registered above:

```
_someUnRegister(some_gEMan, Ev, regId);
```

## An Example Callback Procedure

The following code segment illustrates how to write a callback procedure:

```
void ReadMsg(SOMEEvent event, void *targetData)
{
 int sock;
 printf("Data = %s\n", targetData);
 switch(_somevGetEventType(event)) {
 case EMSinkEvent:
 printf("callback: Perceived Sink Event\n");
 sock = _somevGetEventSink(event);
 /* code to read the message off the socket */
 break;
 default: printf("Unknown Event type in socket callback\n");
 }
}
```

## Generating Client Events

While the other events are caused by the operating system (for example, Timer), by I/O devices, or by external processes, client events are caused by the application itself. The application creates these events and enqueues them with EMan. When client events are dispatched, they are processed in a callback routine just like any other event. The following code segment illustrates how to create and enqueue client events.

```
clientEvent1 = SOMEClientEventNew(); /* create a client event */
_somevSetEventClientType(clientEvent1, Ev, "MyClientType");
_somevSetEventClientData(clientEvent1, Ev,
 "I can give any data here");
/* assuming that "MyClientType" is already registered with EMan */
/* enqueue the above event with EMan */
_someQueueEvent(some_gEMan, Ev, clientEvent1);
```

## Processing Events

After all registrations are finished, an application typically turns over control to EMan and is completely event driven thereafter. Typically, an application main program ends with the following call to EMan:

```
_someProcessEvents(some_gEMan, Ev);
```

An equivalent way to process events is to write a main loop and call **someProcessEvent** from inside the main loop, as indicated:

```
while (1) { /* do forever */
 _someProcessEvent(some_gEMan, Ev, EMProcessTimerEvent
 EMProcessSinkEvent
 EMProcessClientEvent
 EMProcessWorkProcEvent);
 /*** Do other main loop work, as needed. ***/
}
```

The second way allows more precise control over what type of events to process in each call. The example above enables all four types to be processed. The required subset is formed by logically ORing the appropriate bit constants (these are defined in "eventmsk.h"). Another difference is that the second way is a non-blocking call to EMan. That is, if there are no events to process, control returns to the main loop immediately, whereas **someProcessEvents** is a non-returning blocking call. For most applications, the first way of calling EMan is better, since it does not waste processor cycles when there are no events to process.

### Example of Using a Timer Event

The sample program below illustrates the creation and use of a timer event:

```

#include <eman.h>
#include <eventmsk.h>

typedef struct /* for callback procedure */
{
 SOMEEMan *eman; /*Ptr to the SOMEEMan object to use */
 Environment *env;
} CallbackData;

/* callback procedure */
void timerCallbackProc(SOMEEvent *eventP, void *dataP)
{
 CallbackData *cb;

 somPrintf("\ntimerCallbackProc entered\n");
 cb = (CallbackData *)dataP;
 _someShutdown(cb->eman,cb->env);
 return;
}

int main(int argc, char *argv[])
{
 SOMEEMan *some_gEMan;
 CallbackData *d;
 SOMEEMRegisterData *data;
 long regId;

 Environment *ev = somGetGlobalEnvironment();
 some_gEMan= SOMEEManNew(); /* Create EMan object*/
 /* allocate and initialize data for callback procedure */
 d = (CallbackData *)SOMMAlloc(sizeof(CallbackData));
 d->eman=some_gEMan;
 d->env=ev;
 /* Create and initialize registration data */
 data = SOMEEMRegisterDataNew();
 _someClearRegData(data, ev);
 _someSetRegDataEventMask(data, ev, EMTimerEvent, NULL); /* timer */
 _someSetRegDataTimerInterval(data, ev, 1); /* .001 sec */
 /* register procedure with EMan */
 regId = _someRegisterProc(some_gEMan, ev, data, timerCallbackProc,
 d);
 if(regId >= 0)
 {
 _someProcessEvents(some_gEMan, ev);
 _someUnRegister(some_gEMan, ev, regId);
 }
 SOMFree(d);
 return 0;
}

```

## Interactive Applications

Interactive applications need special attention when coupled with EMan. Once control is turned over to EMan by calling `someProcessEvents` method, a single-threaded application has no way of responding to keyboard input. The user must register interest in `stdin` with EMan and provide a callback function that handles keyboard input.

## Event Manager Advanced Topics

The following are the Event Management Framework advanced topics:

- Writing an X or MOTIF application
- Extending EMan
- Using EMan from C++
- Using EMan from other languages
- Tips on using EMan

### Writing an X or MOTIF application

Although the Event Manager does not recognize X events, an X or MOTIF application can be integrated with EMan as follows. First, the necessary initialization of X or MOTIF should be performed. Next, using the Xlib macro “ConnectionNumber” or the “XConnectionNumber” function, you can obtain the file descriptor of the X connection. This file descriptor can be registered with EMan as a sink. It can be registered for both input events and exception events. When there is any activity on this X file descriptor, the developer-provided callback is invoked. The callback can receive the X-event, analyze it, and do further dispatching.

### Extending EMan

The current event manager can be extended without having access to the source code. The use of EMan in an X or MOTIF application mentioned above is just one such example. Several other extensions are possible. For example, new event types can be defined by subclassing either directly from **SOMEEvent** class or from any of its subclasses in the framework. There are three main problems to solve in adding a new event type:

- How to register a new event type with EMan?
- How to make EMan recognize the occurrence of the new event?
- How to make EMan create and send the new event object (a subclass of **SOMEEvent**) to the callback when the event is dispatched?

Because the registration information is supplied with appropriate “set” methods of a RegData object, the RegData object should be extended to include additional methods. This can be achieved by subclassing from **SOMEEMRegisterData** and building a new registration data class that has methods to “set” and “get” additional fields of information that are needed to describe the new event types fully. To handle registrations with instances of new registration data subclass, we must also subclass from **SOMEEMan** and override the **someRegister** and the **someUnRegister** methods. These methods should handle the information in the new fields introduced by the new registration data class and call parent methods to handle the rest.

Making EMan recognize the occurrence of the new event is primarily limited by the primitive events EMan can wait on. Thus the new event would have to be wrapped in a primitive event that EMan can recognize.

The third problem of creating new event objects unknown to EMan can be easily done by applying the previous technique of wrapping the new event in terms of a known event. In a callback routine of the known event, we can create and dispatch the new event unknown to EMan. Of course, this does introduce an intermediate callback routine which would not be needed if EMan directly understood the new event type.

A general way of extending EMan is to look for newly defined event types by overriding **someProcessEvent** and **someProcessEvents** in a subclass of EMan.

### Using EMan from C++

The Event Management framework can be used from C++ just like any other framework in SOMobjects. You must ensure that the C++ usage bindings (that is, the .xh files) are available for the Event Management Framework classes. These .xh files are generated by the SOM Compiler when the **-s** option includes an **xh** emitter.

### Using EMan from Other Languages

The event manager and the other classes can be used from other languages, provided usage bindings are available for them. These usage bindings are produced from .idl files of the framework classes by the appropriate language emitter.

### Tips on Using EMan

The following are some tips for using EMan:

- Eman callback procedures or methods must return quickly. You cannot wait for long periods of time to return from the callbacks. If such long delays occur, then the application may not notice some subsequent events in time to process them.
- It follows from the previous tip that you should not do independent “select” system calls on file descriptors while inside a callback. In general, a callback should not do any blocking of system calls. If an application must do this, then it must be done with a small timeout value.
- Since EMan callbacks must return quickly, no callback should wait on a semaphore indefinitely. If a callback has to obtain some semaphores during its processing, then the callback should try to acquire all of them at the very beginning, and should be prepared to abort and return to EMan if it fails to acquire the necessary semaphores.
- EMan callback methods are called using name-lookup resolution. Therefore, the parameters to an EMan registration call must be such that the class object of the object parameter must be able to provide a pointer to the method indicated by the method parameter. Although this requirement is satisfied in a majority of cases, there are exceptions. For example, if the object is a proxy (in the DSOM sense) to a remote object, then the “real” class object cannot provide a meaningful method pointer. Also note that, when **somDspatch** is overridden, the effect of such an override will not apply to the callback from EMan. Use a procedure callback in these situations.



---

## Chapter 17. Metaclass Framework

Explicit metaclasses are an exciting and powerful part of SOM. They offer the potential to extend object-oriented programming in fundamental ways. However, there is very little academic or industry experience with explicit metaclasses. The SOM development team is committed to bringing SOM customers the full benefit of this exciting technology in a safe and useful manner. The tentative capabilities described in this section are a step in that direction.

This chapter addresses the following topics:

- What is the Metaclass Framework?
- The Uses and Benefits of the Metaclass Framework
- Understanding the Metaclass Framework
- Using the Metaclass Framework

---

### What is the Metaclass Framework?

The Metaclass Framework is a collection of classes that allows you to create a metaclass, introduce new class methods and new class variables.

In SOM, classes are objects. Metaclasses are classes and thus are objects, too. Figure 17-1 on page 17-2 depicts the relationship of these sets of objects. Included are the three primitive class objects of the SOM run time: **SOMClass**, **SOMObject**, and **SOMClassMgr**.

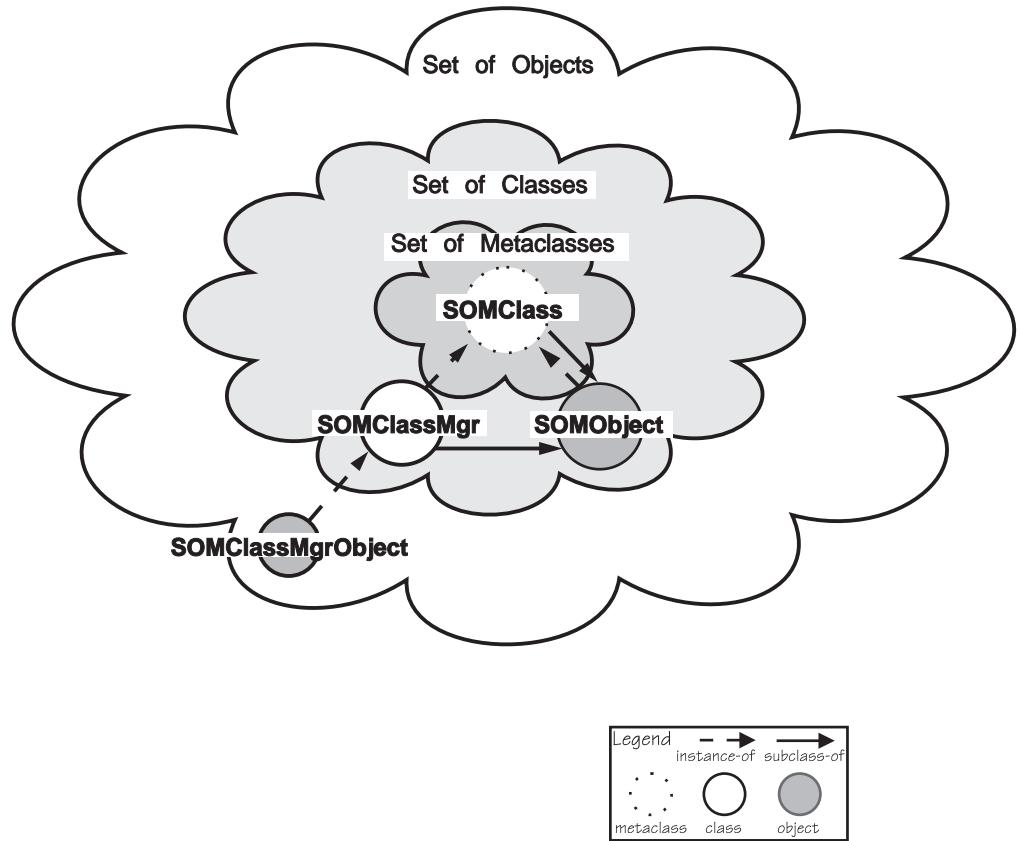


Figure 17-1. The Primitive Objects of the SOM Run Time

The important point to observe here is that any class that is a subclass of **SOMClass** is a metaclass. This chapter describes metaclasses that are available in SOMobjects. There are two kinds of metaclasses:

1. *Framework metaclasses* — metaclasses for building new metaclasses, and
2. *Utility metaclasses* — metaclasses to help you write applications.

Briefly, SOMobjects provides the following metaclasses of each category for use by programmers:

- *Framework metaclasses*:

**SOMMBeforeAfter** Used to create a metaclass that has "before" and "after" methods for *all* methods (inherited, overridden or introduced) invoked on instances of its classes.

- *Utility metaclasses*:

**SOMMSingleInstance**

Used to create a class that may have at most one instance.

**SOMMTraced**

Provides tracing for every invocation of all methods on instances of its classes.

**SOMM\_MVS\_Secure**

Used to protect an object by controlling client access to the method interfaces for the object.

**Note:** The MVS secure metaclass is intended for the DSOM server environment only. If used in a client or in a base SOM application, the security mechanisms can be circumvented.

#### SOMRReplicableObject

Provides an encapsulation of the Replication Framework (see Chapter 9).

#### SOMRReplicable

Provides the Before/After method required by SOMRReplicableObject.

The diagram in Figure 17-2 depicts the relationship of these metaclasses to **SOMClass** (for completeness, Figure 17-2 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.

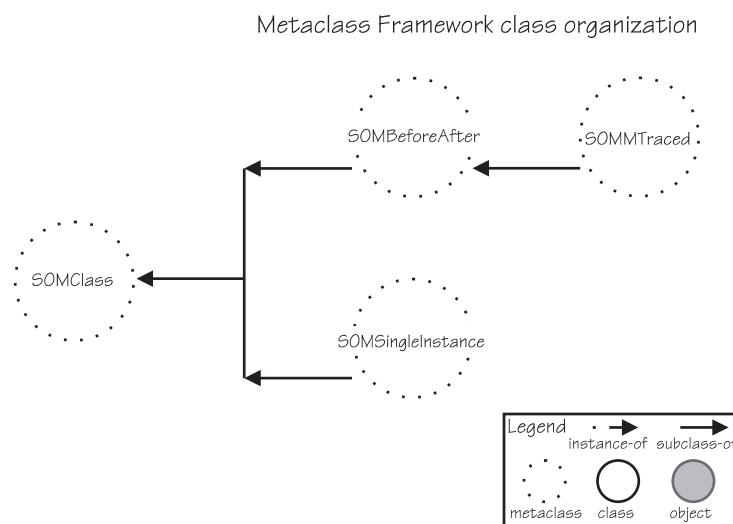


Figure 17-2. Metaclass Framework Class Organization

### A Note about Metaclass Programming

SOM metaclasses are carefully constructed so that they compose (see Section 10.1 below). If you need to create a metaclass, you can introduce new class methods, and new class variables, but you should not override any of the methods introduced by **SOMClass**.

### The SOMMBeforeAfter metaclass

**SOMMBeforeAfter** is a metaclass that allows the user to create a class for which a particular method is invoked *before* each invocation of every method, and for which a second method is invoked *after* each invocation. **SOMMBeforeAfter** defines two methods: **sommBeforeMethod** and **sommAfterMethod**. These two methods are intended to be overridden in the child of **SOMMBeforeAfter** to define the particular "before" and "after" methods needed for the client application.

As further depicted in Figure 17-3 on page 17-4, the "Barking" metaclass overrides the **sommBeforeMethod** and **sommAfterMethod** with a method that emits one bark when invoked. Thus, one can create the "BarkingDog" class, whose instances (such as "Lassie") bark twice when "disturbed" by a method invocation.

## A hierarchy of metaclasses

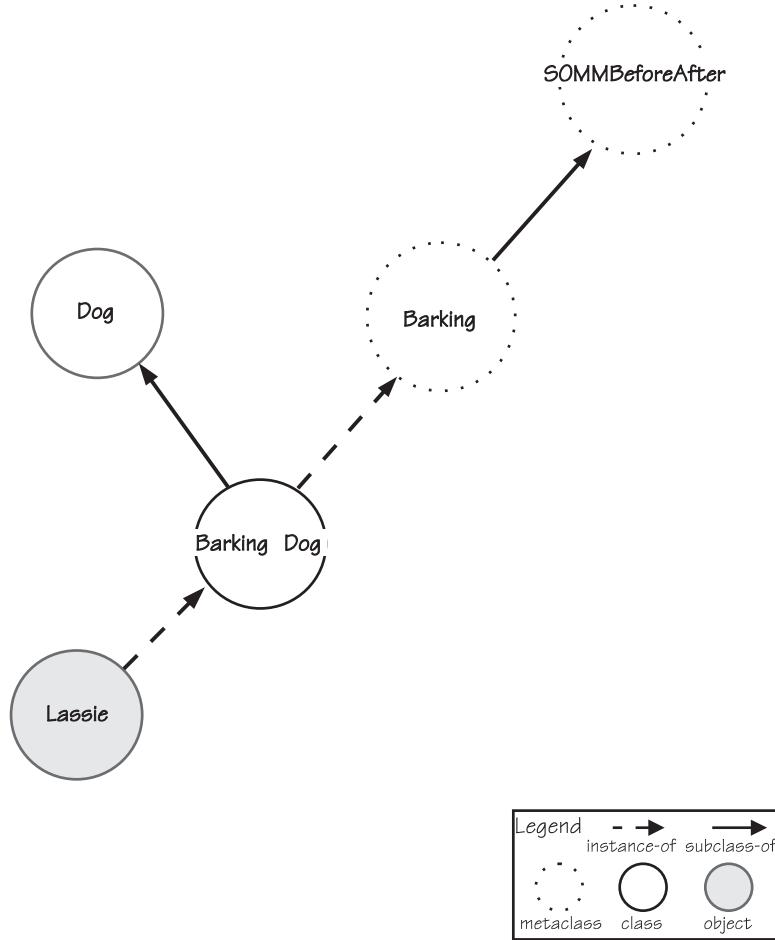


Figure 17-3. A Hierarchy of Metaclasses

The **SOMMBeforeAfter** metaclass is designed to be subclassed; a subclass (or child) of **SOMMBeforeAfter** is also a metaclass. The subclass overrides **sommBeforeMethod** or **sommAfterMethod** or both.

With **SOMMBeforeAfter** methods:

- the method that the client is requesting to use is referred to as the primary method
- the **sommBeforeMethod** receives control before the primary method is invoked, and
- the **sommAfterMethod** receives control after the primary method completes.

That is, they are invoked before and after methods invoked on the ordinary objects that are instances of the class objects that are instances of the subclass of **SOMMBeforeAfter**.

The **sommBeforeMethod** returns a **boolean** value. This allows the "before" method to control whether the "after" method and the primary method get invoked. If **sommBeforeMethod** returns TRUE, normal processing occurs. If FALSE is returned, neither the primary method nor the corresponding **sommAfterMethod** is invoked. In addition, no more deeply nested before/after methods are invoked (see "Composition of Before/After Metaclasses" below). This facility can be used, for

example, to allow a before/after metaclass to provide secure access to an object. The implication of this convention is that, if **sommBeforeMethod** is going to return FALSE, it must do any post-processing that might otherwise be done in the "after" method.

**CAUTION:**

**somInit** and **somFree** are among the methods that get before/after behavior. This implies the following two obligations are imposed on the programmer of a **SOMMBeforeAfter** class. First, the implementation must guard against **sommBeforeMethod** being called before **somInit** has executed, and the object is not yet fully initialized. Second, the implementation must guard against **sommAfterMethod** being called after **somFree**, at which time the object no longer exists (see the example "C implementation for 'Barking' metaclass" below).

The following example shows the IDL needed to create a Barking metaclass. Just run the appropriate emitter to get an implementation binding, and then provide the appropriate "before" behavior and "after" behavior.

### SOM IDL for 'Barking' Metaclass

```
#ifndef Barking_idl
#define Barking_idl

#include <sombacls.idl>
interface Barking : SOMMBeforeAfter
{
#ifdef __SOMIDL__
implementation
{
 //# Class Modifiers
 filestem = barking;
 callstyle = idl;

 //# Method Modifiers
 sommBeforeMethod : override;
 sommAfterMethod : override;
};
#endif /* __SOMIDL__ */
};
#endif /* Barking_idl */
```

The next example shows an implementation of the Barking metaclass in which *no* barking occurs when **somFree** is invoked.

### C implementation for 'Barking' Metaclass

```

#define Barking_Class_Source
#include <barking.ih>

static char *somMN_somFree = "somFree";
static somId somId_somFree = &somMN_somFree;

SOM_Scope boolean SOMLINK sommBeforeMethod(Barking somSelf,
 Environment *ev,
 SOMObject object,
 somId methodId,
 va_list ap)
{
 if (!somCompareIds(methodId, somId_somFree))
 printf("WOOF");
}
SOM_Scope void SOMLINK sommAfterMethod(Barking somSelf,
 Environment *ev,
 SOMObject object,
 somId methodId,
 somId descriptor,
 somToken returnedvalue,
 va_list ap)
{
 if (!somCompareIds(methodId, somId_somFree))
 printf("WOOF");
}

```

## Composition of Before/After Metaclasses

Consider Figure 17-4 on page 17-7 in which there are two before/after metaclasses "Barking" (as before) and "Fierce", which has a **sommBeforeMethod** and **sommAfterMethod** that both growl (that is, both make a "grrrr" sound when executed). The preceding discussion demonstrated how to create a "FierceDog" or a "BarkingDog", but has not yet addressed the question of how to compose these properties of fierce and barking. *Composability* means having the ability to easily create either a "FierceBarkingDog" that goes "grrr woof woof grrr" when it responds to a method call or a "BarkingFierceDog" that goes "woof grrr grrr woof" when it responds to a method call.

## Example for composition of before/after metaclasses

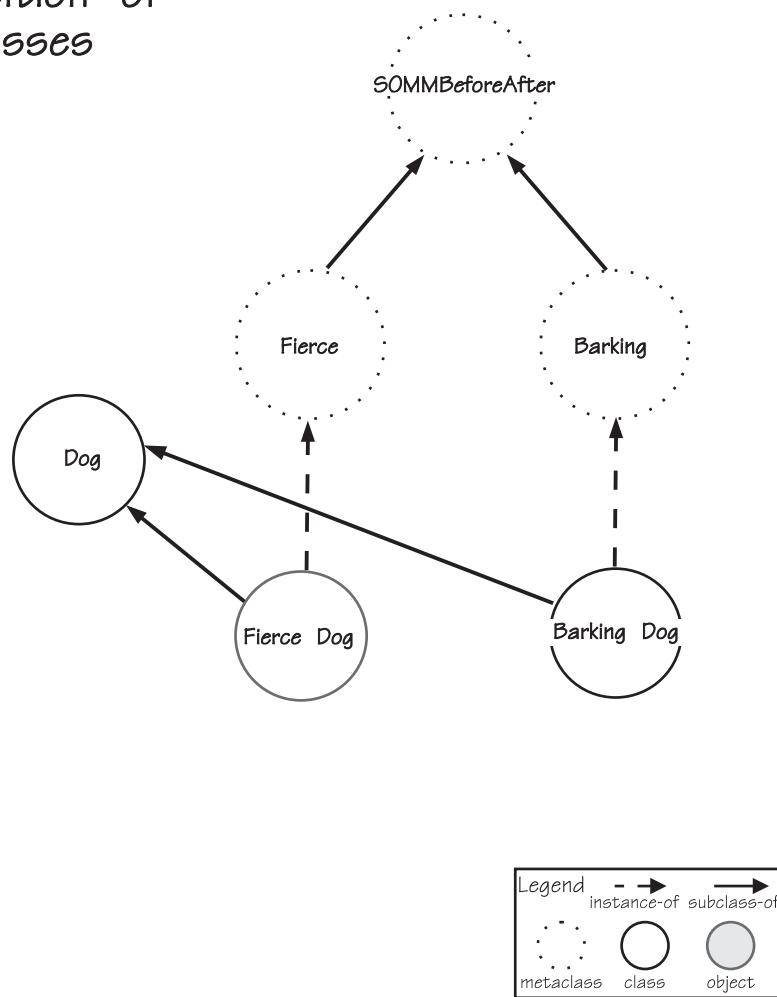


Figure 17-4. Example for Composition of Before/After Metaclasses

There are several ways to express such compositions. Figure 17-5 on page 17-8 depicts SOM IDL fragments for three techniques in which composition can be indicated by a programmer. These are denoted as Technique 1, Technique 2, and Technique 3, each of which creates a "FierceBarkingDog" class, named "FB-1", "FB-2", and "FB-3", respectively, as follows:

- In Technique 1, a new metaclass ("FierceBarking") is created with both the "Fierce" and "Barking" metaclasses as parents. An instance of this new metaclass (that is, "FB-1") is a "FierceBarkingDog" (assuming "Dog" is a parent).
- In Technique 2, a new class is created which has parents that are instances of "Fierce" and "Barking" respectively. That is, "FB-2" is a "FierceBarkingDog" also (assuming "FierceDog" and "BarkingDog" do not further specialize "Dog").
- In Technique 3, "FB-3", which also is a "FierceBarkingDog", is created by declaring that its parent is a "BarkingDog" and that its explicit (syntactically declared) metaclass is "Fierce".

| Technique 1                                                                                                                | Technique 2                                                                                               | Technique 3                                                                                                                |
|----------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <pre>interface FB-1 : Dog {     ...     implementation     {         metaclass = FierceBarking;         ...     }; }</pre> | <pre>interface FB-2 : FierceDog, BarkingDog {     ...     implementation     {         ...     }; }</pre> | <pre>interface FB-3 : BarkingDog {     ...     implementation     {         metaclass = Fierce;         ...     }; }</pre> |

Figure 17-5. Three Techniques for Composing Before/After Metaclasses

Figure 17-6 combines the diagrams for the techniques in Figure 17-5 and shows the actual class relationships. Note that the explicit metaclass in the SOM IDL of "FB-1" is its derived class, "FierceBarking". The derived metaclass of "FB-2" is also "FierceBarking". Lastly, the derived metaclass of "FB-3" is not explicitly specified in the SOM IDL; rather, it too is "FierceBarking".

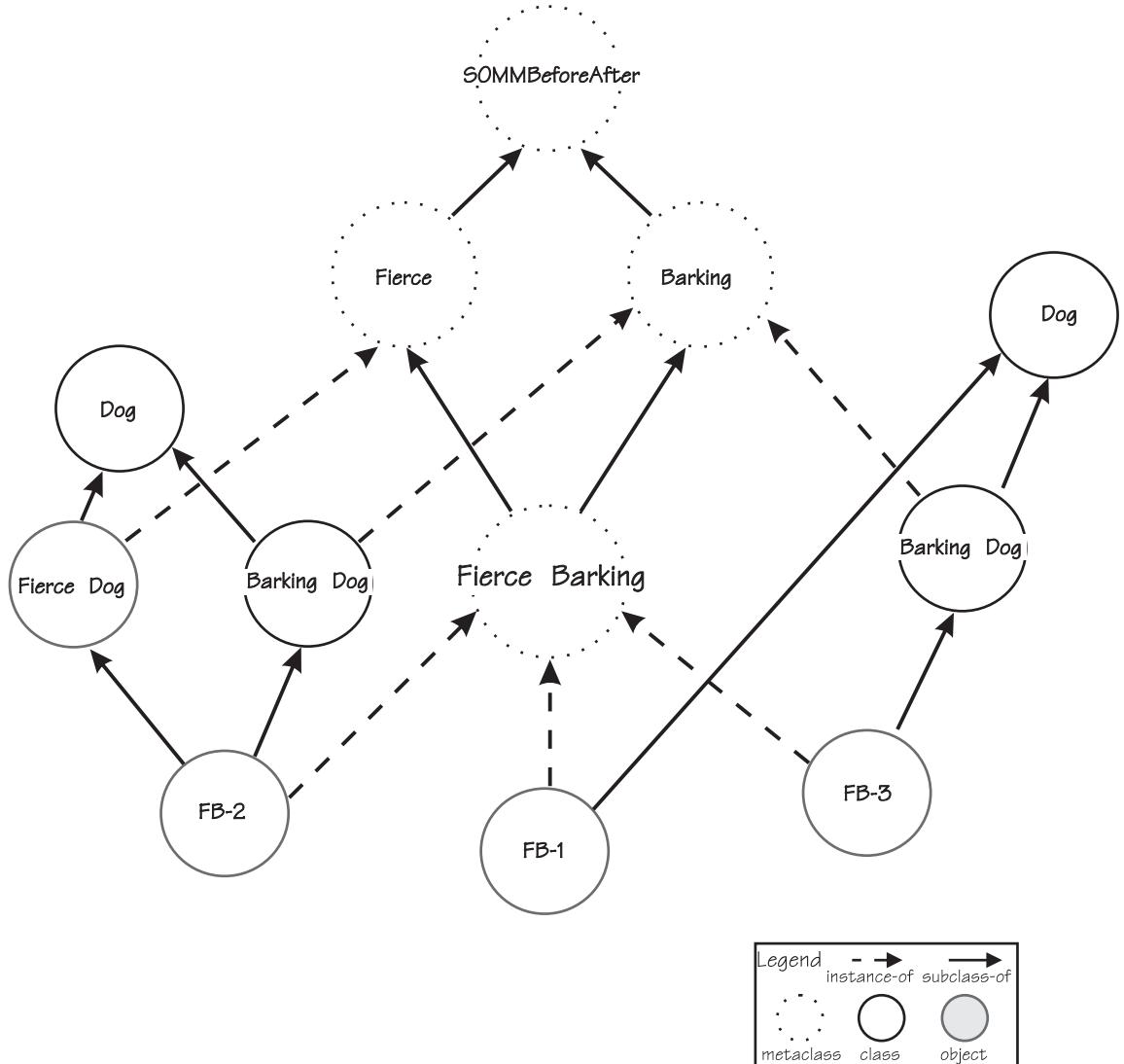


Figure 17-6. Three Techniques for Creating a "FierceBarkingDog"

## Notes and Advantages of ‘Before/After’ Usage

Notes on the dispatching of before/after methods:

- A before (after) method is invoked just once per primary method invocation.
- The dispatching of before/after methods is fast. The time overhead for dispatching a primary method is on the order of  $N$  times the time to invoke a before/after method as a procedure, where  $N$  is the total number of before/after methods to be applied.

In conclusion, consider an example that clearly demonstrates the power of the composition of before/after metaclasses. Suppose you are creating a class library that will have  $n$  classes. Further suppose there are  $p$  properties that must be included in all combinations for all classes. Potentially, the library must have  $n2^p$  classes. Let us hypothesize that (fortunately) all these properties can be captured by before/after metaclasses. In this case, the size of the library is  $n + p$ .

The user of such a library need only produce those combinations necessary for a given application. In addition, note that there is none of the usual programming. Given the IDL for a combination of before/after metaclasses, the SOM compiler generates the implementation of the combination (in either C or C++) with no further manual intervention.

## The **SOMMSingleInstance** Metaclass

Sometimes it is necessary to define a class for which only one instance can be created. This is easily accomplished with the **SOMMSingleInstance** metaclass. Suppose the class "Collie" is an instance of **SOMMSingleInstance**. The first call to **CollieNew** creates the one possible instance of "Collie"; hence, subsequent calls to **CollieNew** return the first (and only) instance.

Any class whose metaclass is **SOMMSingleInstance** gets this requisite behavior; nothing further needs to be done. The first instance created is always returned by the `<className>New` macro.

Alternatively, the *method* **sommGetSingleInstance** does the same thing as the `<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 **somRenew**, **somRenewNoInit**, **somRenewNoInitNoZero**, and **somRenewNoZero** so that a proxy is created in the space indicated in the **somRenew\*** call. This proxy rediscusses all methods to the single instance, which is always allocated in heap storage. Note that all of these methods (**somRenew\***) increment the reference count; therefore, **somFree** should be called on these objects, too. In this case, **somFree** decrements the reference and frees the single instance (and, of course, takes no action with respect to the storage indicated in the original **somRenew\*** call).

If a class is an instance of **SOMMSingleInstance**, all of its subclasses are also instances of **SOMMSingleInstance**. Be aware that this also means that each *subclass* is allowed to have only a single instance. (This may seem obvious. However, it is a common mistake to create a framework class that must have a single instance, while at the same time expecting users of the framework to subclass the single instance class. The result is that two single-instance objects are created: one for the framework class and one for the subclass. One technique that can mitigate this scenario is based on the use of **somSubstituteClass**. In this case, the creator of the subclass must substitute the subclass for the framework class — before the instance of the framework class is created.)

## The SOMMTraced Metaclass

**SOMMTraced** is a metaclass that facilitates tracing of method invocations. The **SOMMTraced** metaclass can be used to trace the paths of your methods. This differs from the DSOMTrace facility, found in *OS/390 SOMobjects Messages, Codes, and Diagnosis*, which can be used to gather trace data that can be merged from different processes (such as client and server).

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 17-7 on page 17-11. 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.

All methods invoked on "Collie"  
are traced

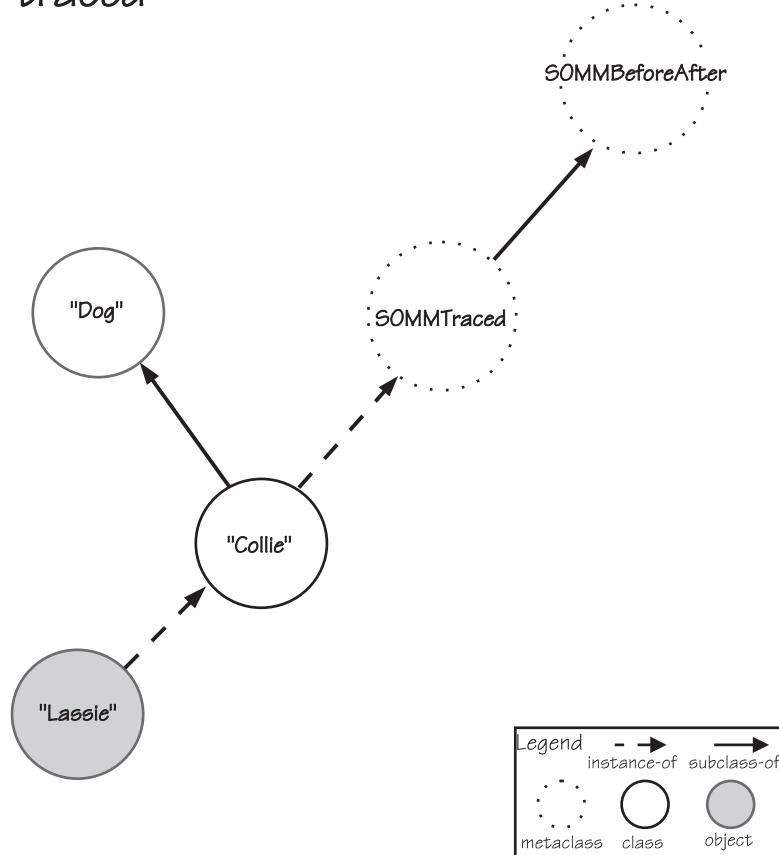


Figure 17-7. All Methods That Are Invoked on "Collie" Are Traced

It is easy to use **SOMMTraced**: Just make a class an instance of **SOMMTraced** in order to get tracing.

There is one more step for using **SOMMTraced**: Nothing prints unless the environment variable **SOMM\_TRACED** is set. If it is set to the empty string, all traced classes print. If **SOMM\_TRACED** is not the empty string, it should be set to the list of 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 (in the OpenEdition shell)
SOMM_TRACED=Collie (in your SOMENV file)
```

**Note:** The **SOMENV** file is either a DDNAME of **SOMENV** in a JCL DD statement, or a file of **SOMENV** in a TSO ALLOCATE command.

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__ */
};

```

## The SOMM\_MVS\_Secure Metaclass

SOMM\_MVS\_Secure is a metaclass that can be used to protect methods by controlling client access to the method interfaces of an object. The encapsulation of object attributes and enabling of method protection for a class allow for secure classes and methods to be created.

Access to protected methods is administered through the use of the Resource Access Control Facility (RACF). Only clients that provide a valid OS/390 user ID and password (or PassTicket) and have the required access authority will be granted permission to use protected methods. Methods to be protected can be defined to RACF in general resource profiles in the SOMDOBJS and GSOMDOBJ general resource classes. For detailed information on setting up these profiles, see *OS/390 SOMobjects Configuration and Administration Guide*.

SOMM\_MVS\_Secure is a subclass of the SOMMBeforeAfter class. SOMM\_MVS\_Secure has methods that override the sommBeforeMethod and sommAfterMethod classes, although the sommAfterMethod override does not contribute to method protection. Recall the following about SOMMBeforeAfter methods:

- the method that the client is requesting to use is referred to as the primary method,
- the sommBeforeMethod receives control before the primary method is invoked, and
- the sommAfterMethod receives control after the primary method completes.

A client request to use a protected method will cause the sommBeforeMethod (overridden by SOMM\_MVS\_Secure) to be invoked prior to invocation of the primary method. The sommBeforeMethod invokes RACF to determine if the client is permitted to use the method. If RACF indicates that the client is authorized to use the method, then the primary method is invoked and the sommAfterMethod is invoked. If RACF indicates that the client is not authorized to use the method, then the primary and sommAfterMethod methods are not invoked and a security exception is raised to indicate that the client is not authorized to use the method. The security exception is returned to the client program, which must check for security exceptions and take whatever action is appropriate.

Specifying SOMM\_MVS\_Secure as the metaclass of a class will enable RACF authorization checking for all methods that can be invoked against instances of the class. All classes derived from a class for which method protection has been

enabled will inherit method protection, causing all methods in the derived class to be protected.

IBM provides the SOM\_MVS\_Secure class (as depicted in Figure 17-8), which specifies SOMM\_MVS\_Secure as its metaclass. The SOM\_MVS\_Secure class has no attributes and no methods. The SOM\_MVS\_Secure class is designed to be subclassed. All subclasses of SOM\_MVS\_Secure will inherit method protection. Note that method protection will also be enabled for any class that specifies a metaclass derived from SOMM\_MVS\_Secure.

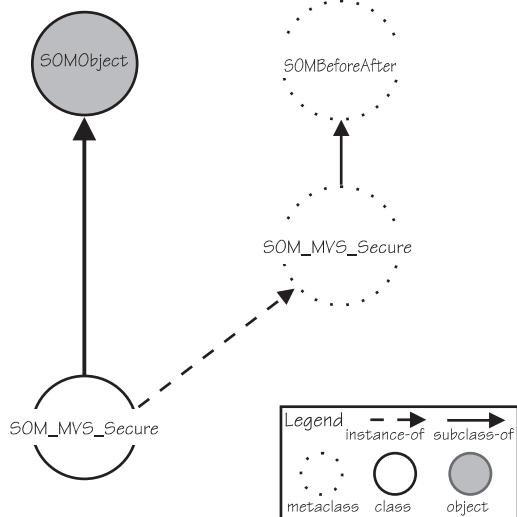


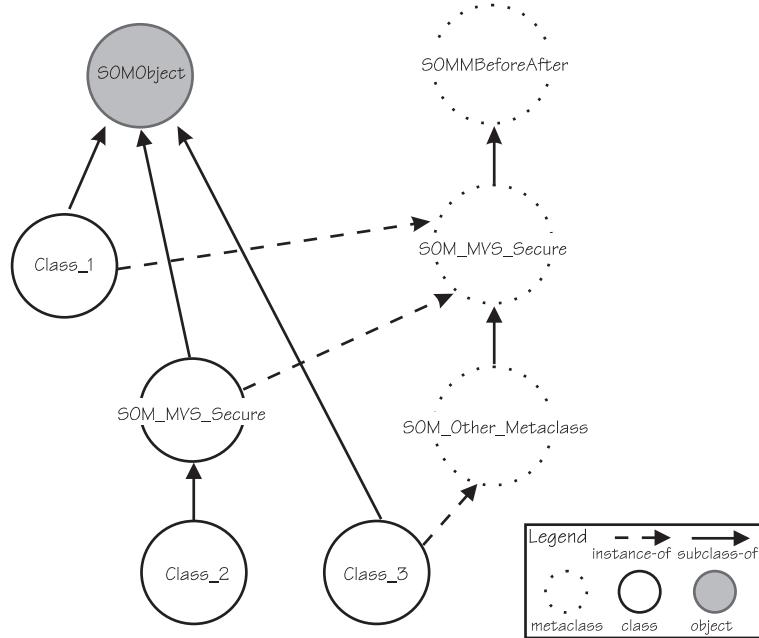
Figure 17-8. The SOMM\_MVS\_Secure Metaclass

There are three ways that class relationships can be arranged to protect the methods of a class. These are listed below and depicted in Figure 17-9 on page 17-14.

1. When a class inherits from SOM\_MVS\_Secure along the leftmost inheritance path. In other words, if you start from the current class and follow up along the leftmost inheritance path for that class, you would find SOM\_MVS\_Secure. Note that this ordering is very important. Even when you inherit from SOM\_MVS\_Secure, but do not inherit according to this criterion, the before method that checks for authorization will not be invoked and one ends up unprotected. The inheritance ordering is important for this to work.

**Note:** It is recommended to use the SOM\_MVS\_Secure class instead of the SOMM\_MVS\_Secure metaclass.

2. When a class specifies a metaclass that is derived from SOMM\_MVS\_Secure.
3. When a class specifies SOMM\_MVS\_Secure as its metaclass.



*Figure 17-9. Three Techniques for Creating a Secure Class*

The IDL for Class\_1, a subclass of SOMObject, contains the statement

```
metaclass = SOMM_MVS_Secure;
```

Therefore, the methods of Class\_1 will be protected. The IDL for Class\_2 does not explicitly specify a metaclass, but is a subclass of SOM\_MVS\_Secure. Thus, Class\_2 will inherit method protection from SOM\_MVS\_Secure. (Recall that SOM\_MVS\_Secure explicitly specifies SOMM\_MVS\_Secure as its metaclass.) The IDL for Class\_3 explicitly specifies a metaclass that is a subclass of SOMM\_MVS\_Secure. Hence, method protection is enabled for Class\_3.

### An Example

Consider the following IDL for the GuardDog class, introducing the methods attack, bite, and heel.

```

#ifndef GuardDog_idl
#define GuardDog_idl

#include <dog.idl>
#include <somdsec.idl>
interface GuardDog : SOM_MVS_Secure, Dog
{
 void attack();
 void bite();
 void heel();

#ifdef __SOMIDL__
implementation
{
 //# Class Modifiers
 filestem = guarddog;
 callstyle = idl;

};

#endif /* __SOMIDL__ */
};

#endif /* GuardDog_idl */

```

The GuardDog methods are not intended for use by the general public. Unlike normal Dog methods, it is necessary to strictly control who uses the attack, bite, and heel methods. To provide this control, the GuardDog class is derived from the Dog and SOM\_MVS\_Secure classes. GuardDog inherits attributes and methods from the Dog class and method protection from the SOM\_MVS\_Secure class.

The somdsec.idl must be included in the IDL in order for SOMM\_MVS\_Secure metaclass to work.

If the GuardDog does not require a metaclass that provides for a particular function, then SOMM\_MVS\_Secure metaclass can be designated as the metaclass of GuardDog and allow for its protection. This is because a class is allowed to have only one explicit metaclass during its definition. This would also enable method protection for all GuardDog methods.

```

#ifndef GuardDog_idl
#define GuardDog_idl

#include <dog.idl>
interface GuardDog : Dog
{
 void attack();
 void bite();
 void heel();

#ifdef __SOMIDL__
implementation
{
 /* Class Modifiers
 filestem = guarddog;
 callstyle = idl;
 metaclass = SOMM_MVS_secure;

};

#endif /* __SOMIDL__ */
};

#endif /* GuardDog_idl */

```

## Security Administration for Protected Methods

Access to protected methods is administered through the use of RACF. For a detailed description of security administration using RACF, please refer to *OS/390 SOMobjects Configuration and Administration Guide*.

Each method to be protected can be defined to RACF in a general resource profile in the SOMDOBJS or GSOMDOBJ class. Profile names in the SOMDOBJS class are of the form *classname.methodname*, where *classname* is the SOM class of the object receiving the request to invoke the method identified by *methodname*. In the SOMDOBJS class, generic profile names (for example, including \*), can be used to protect methods and classes that have similar names. For methods and classes that do not have similar names, you can create profiles in the GSOMDOBJ class.

If a level of RACF which includes the CBIND, SERVER, and SOMDOBJS security classes is not installed, then authorization processing will not be performed.

**Note:** The secure metaclass processing will be performed even if the class is deployed in a non-secure server. This is to protect a class for which security controls are desired from being deployed in a non-secure server.

## An Example of Inherited Method Protection

Consider the simple class relationships depicted in Figure 17-10 on page 17-17. If method protection is enabled for Class\_C and a client requests use of inherited method Meth\_A1 for a receiving object of class Class\_C, then the profile CLASS\_C.METH\_A1 could be used to determine if the client has the authority to use method Meth\_A1. If the profile did not exist and Meth\_A1 did not fall within the scope of any generic profile, then the client would be denied use of Meth\_A1. For more information on setting up security, see *OS/390 SOMobjects Configuration and Administration Guide*.

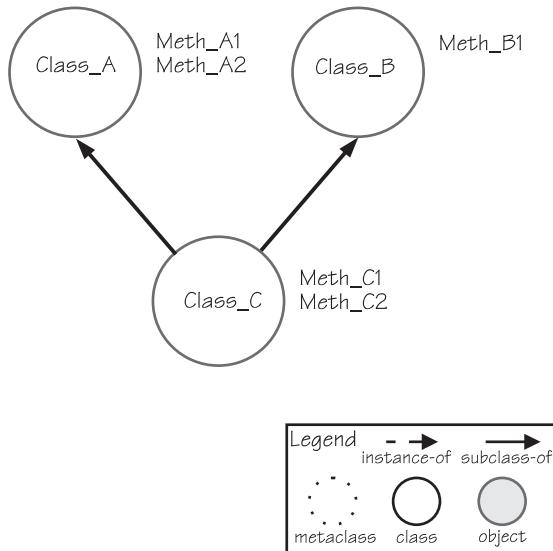


Figure 17-10. Inherited Method Protection

### An Example Illustrating the Use of Generic and Discrete Profiles

All rules that apply to RACF general resource classes apply to the SOMDOBJS class. For example, a generic profile can be defined to protect several methods of a single SOMObjects class that have the same security requirements. Consider the following SOM IDL for the MoodyDog class.

```

#ifndef MoodyDog_idl
#define MoodyDog_idl

#include <dog.idl>
interface MoodyDog : Dog
{
 void howl();
 void snarl();
 void yap();
 void yelp();
 void wagtail();
 void whimper();
 void whine();

#ifdef __SOMIDL__
implementation
{
 //## Class Modifiers
 filestem = moodydog;
 callstyle = idl;
 metaclass = SOMM_MVS_Secure;

};

#endif /* __SOMIDL__ */
};

#endif /* MoodyDog_idl */

```

All MoodyDog methods could be protected with a profile named MOODYDOG.\* in the SOMDOBJS class. The SOMDOBJS profile MOODYDOG.WHI\*

could be used to protect only the whimper and whine methods. Note that the RACF profiles are not case sensitive.

Discrete profiles could also be used to protect the methods of MoodyDog. For example, to protect each method of MoodyDog with its own discrete profile (allowing individual control of which clients use each method), the following SOMDOBJS profiles could be defined:

```
MOODYDOG.HOWL
MOODYDOG.SNARL
MOODYDOG.YAP
MOODYDOG.YELP
MOODYDOG.WAGTAIL
MOODYDOG.WHIMPER
MOODYDOG.WHINE
```

If the MoodyDog class has one or more parent classes (for example, Dog and Animal), then the generic profile MOODYDOG.\* would be used to check a client's authorization to all methods inherited from the Dog and Animal classes for which more-specific profiles (such as discrete profiles) do not exist.

---

## Appendix A. Setting up Configuration Files

This section describes configuration files, gives the syntax for configuration files, and describes the stanzas and keyword variables for SOMobjects *as they apply to the application programmer and end user*. For a complete description of configuration files, including system-wide information, see *OS/390 SOMobjects Configuration and Administration Guide*.

---

### About Configuration Files

**Note:** The configuration file is also known as the environment file, the configuration file, and the profile data set.

### Configuration File Variable Settings

SOMobjects uses variable settings specified by a configuration file. The configuration file is specified in one of two ways:

- For batch jobs, started tasks, and TSO/E users, the configuration file is an MVS data set specified on the SOMENV DD statement or using the TSO/E ALLOCATE FILE(SOMENV) command. If more than one data set is specified, the first occurrence of any one SOMobjects environment variable is used.

For example, the following illustrates how to specify configuration files through the SOMENV DD statement:

```
//SOMENV DD DSN=SOMMVS.SOMENV.INI,DISP=SHR
or
//SOMENV DD DSN=MYINI.SOMENV.INI,DISP=SHR
// DD DSN=SOMMVS.SOMENV.INI,DISP=SHR
```

- For jobs or users running in an OpenEdition environment, the SOMENV environment variable, if specified, is used to define which file or files are to be used as the configuration file. If the SOMENV environment variable is not specified, the SOMENV DD statement or TSO/E ALLOCATE command is used.

For example, the following illustrates how to specify configuration files through the SOMENV OpenEdition environment variable:

```
SOMENV='//somenv.somenv.ini'
or
SOMENV='myini.somenv.ini';//somenv.somenv.ini'
```

**Note:** If both the SOMENV environment variable and the SOMENV DD statement (or TSO/E ALLOCATE command) are specified, the SOMENV environment variable is used and the other is ignored.

The configuration file is organized into stanzas that contain keywords. For example, the SOMDPORT keyword, specified in the [somed] stanza:

```
[somed]
SOMDPORT=9393
```

The combination of stanza and keyword creates a unique variable setting. The same keyword specified in a different stanza represents a different variable setting. For example:

```
[somd]
SOMDPORT=9393
and
[SOMD_TCPIP]
SOMDPORT=9393
represent two distinct variable settings.
```

**Note:** Stanza and keyword names are case sensitive.

When you specify more than one configuration file, the variable settings are combined, starting with the first file or data set. In the case of a duplicate variable specification, the first occurrence of a given variable specification takes precedence.

For example, if data set 'MYINI.SOMENV.INI' contains:

```
[somd]
SOMDPORT=9393
```

and 'SOMMVS.SOMENV.INI' contains

```
[somd]
SOMDPORT=5001
```

and your configuration file specification is

SOMENV='//MYINI.SOMENV.INI';//SOMMVS.SOMENV.INI', the effective SOMDPORT setting for the [somd] stanza is 9393.

## Comparison of Global and Local Configuration File Values

The SOM subsystem processes a configuration file (called the "global configuration file") during its initialization. The variable settings from that configuration file are available to all SOMobjects applications running on the same system (or in the same sysplex) as the SOM subsystem. The SOM subsystem's configuration file is known as the global configuration file.

Each SOMobjects application can also specify its own configuration file (called the local configuration file). The local configuration file is optional.

When a SOMobjects application specifies a local configuration file, the variable settings in the local configuration file override duplicate variable settings from the global configuration file. For example, if a SOMobjects application's local configuration file contains:

```
[somc]
SMINCLUDE=//MYAPP.IDL;//SOMMVS.SGOSIDL;//SOMMVS.SGOSEFW';
```

and the global configuration file contains:

```
[somc]
SMINCLUDE=//SOMMVS.SGOSIDL;//SOMMVS.SGOSEFW';
```

the effective SMINCLUDE setting for that SOMobjects application would be:

```
SMINCLUDE=//MYAPP.IDL;//SOMMVS.SGOSIDL;//SOMMVS.SGOSEFW';
```

## Configuration File Syntax

There are two elements present in a configuration file:

- stanza name
- keyword variable name and setting.

These are the syntax rules for the configuration file:

- Comment lines begin with a semicolon (;) in the first position of the line.
- Each stanza begins with the name of the stanza enclosed in square brackets ([ and ]) at the beginning of a line. The stanza names are case sensitive.  
**Note:** The hexadecimal representation of the left and right square brackets used by SOMobjects is X'AD' and X'BD', respectively.
- Blanks and spaces can appear anywhere in the file.
- Each keyword variable is associated with the stanza name preceding it in the file.
- Keyword variable settings are of the form **Keyword=value**. The keyword variable names are case sensitive. If you wish to continue the value on the subsequent line, put a backward slash (\) as the last non-blank character on the line to be continued and begin the continuation in the first position of the next line.

---

## Stanzas and Keyword Variables

The following sections describe the SOMobjects environment variables. For each variable, the following information is provided:

|                |                                                                                                                                                                                                                                                                                                                                                                               |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Usage</b>   | Must the variable be explicitly specified in the configuration file? What are the special considerations that would cause it to be required?                                                                                                                                                                                                                                  |
| <b>Scope</b>   | System, local, or both. System means that the variable is defined in the global configuration file and it is not intended that individual SOMobjects processes or users modify such variables.<br><br>Local means that individual SOMobjects processes or users may find it useful to specify the variable, and the variable isn't intended to affect all processes or users. |
| <b>Default</b> | The value assigned to the variable if no value is specified in the configuration file.                                                                                                                                                                                                                                                                                        |

### [somc] Stanza (SOMobjects Compiler)

This stanza specifies the settings for the SOMobjects compiler.

#### **SMOE=YES**

Indicates that the SOMobjects compiler should interpret any ambiguous file names as HFS files (files in the hierarchical file system). The default is for ambiguous file names to be interpreted as MVS data set names.

**Note:** If you are planning to develop SOMobjects applications completely within the OpenEdition shell and using the hierarchical file system, then you should export the SMOE environment variable in your setup or profile shell script. For more information, see *OS/390 SOMobjects Programmer's Guide*.

**Usage:** Required if running in the OpenEdition shell; not required otherwise.

**Scope:** Local

**Default:** NO

**SMINCLUDE=***name1;name2;...namen*

Specifies where the SOMobjects compiler should look for IDL members #included by the IDL data set being compiled.

The values specified for *name1*, *name2*, and so forth can be any of the following:

- MVS data set names (PDSs only, without member names), either fully-qualified or not.
- HFS directories
- A dot (indicates the current directory).

Semicolons separate names. You can specify both HFS directories and MVS data set names, in any order, in the same SMINCLUDE setting. When specifying *name* for the SMINCLUDE variable, the following rules apply:

- To specify a fully-qualified MVS data set name, specify // before the name and put quotes around the name, such as:  
`//'SOMMVS.DSNAME'`
- To use the current userid (or prefix) as the high-level qualifier, specify // before the name and omit the quotes, such as:  
`//DSNAME`
- If the IDL member is an HFS file, omit // and specify the path to the directory where the file resides, such as:  
`/u/user1/`  
or, for the current directory, specify a dot ( . ), such as:  
`.`

**Examples:**

To look first in the current directory, then in the IBM-supplied data sets 'SOMMVS.SGOSIDL' and 'SOMMVS.SGOSEFW', specify:

```
SMINCLUDE=.;//'SOMMVS.SGOSIDL';//'SOMMVS.SGOSEFW';
```

To look first in a data set named MYAPP.IDL allocated under the current userid or prefix, then in the IBM-supplied data sets 'SOMMVS.SGOSIDL' and 'SOMMVS.SGOSEFW', specify:

```
SMINCLUDE=/MYAPP.IDL;//'SOMMVS.SGOSIDL';//'SOMMVS.SGOSEFW';
```

**Usage:** Required only if a configuration file is input to the SOMobjects compiler.

**Scope:** Both

**Default:** No default

**NEWEMITEFW=h/q**

Sets the high-level qualifiers found on the profile NEWEMITEFW record into the symbol table referenced by the emitter framework utilities. This value, named "TemplateHLQs" is substituted in the GEN@EMIT template to specify the high-level qualifiers of the user's template.

**Usage:** Required if running the NEWEMIT utility

**Scope:** Local

**Default:** Null

#### **SMNOPRINTNAME=SUPPRESS**

Suppresses the printing of processing messages.

**Usage:** Optional

**Scope:** Local

**Default:** No default

#### **SMEMITPREFIX=emitter-prefix**

Emitter prefix.

**Usage:** Optional

**Scope:** Local

**Default:** emit

#### **SMEMIT=emitters**

Specifies which output files the SOMobjects compiler produces. Specify a list of items separated by colons. Each item designates an emitter to execute. For example, the statement:

SMEMIT=c:h:ih

indicates the C, H and IH emitters will be run. This directs the SOMobjects compiler to produce the C implementation template (*dataset\_stem.c(payroll)*), and the C language bindings (*dataset\_stem.h(payroll)*) and *dataset\_stem.ih(payroll)*) from the *dataset\_stem.idl(payroll)* IDL.

By comparison:

SMEMIT=xc:xh:xih

indicates the XC, XH and XIH emitters will be run. This directs the SOMobjects compiler to produce the C++ implementation template (*dataset\_stem.cxx(payroll)*), and the C++ language bindings (*dataset\_stem.xh(payroll)* and *dataset\_stem.xih(payroll)*) from the *dataset\_stem.idl(payroll)* IDL.

**Usage:** Optional

**Scope:** Local

**Default:** h:ih

#### **SMADDSTAR**

When defined, causes all interface references to have a \* added to them for the C emitters (c, h, ih). However, the command line SOM compiler options **-maddstar** and **-mnoaddstar** supersede and override the SMADDSTAR setting.

**Note:** The SMADDSTAR environment variable does not affect output from the C++ emitters (xc, xh, xih, hh). These C++ emitters cause all interface references to have a \* added to them.

**Usage:** Optional

**Scope:** Local

**Default:** None (\* not added to C)

## [somd] Stanza (Distributed SOMobjects)

This stanza specifies the settings for distributed SOMobjects (DSOM).

### **SOMDCOMMDEBUGFLAG=*flag***

If SOM\_TraceLevel=2 or higher is specified, SOMDCOMMDEBUGFLAG specifies the amount of communications debugging information provided by SOMobjects. Normally, this variable would not be set.

Specifying SOMDCOMMDEBUGFLAG=1 enables a display of TRANSPORT information. TRANSPORT information is data associated with the communication socket such as "Create Listener" and "Sending Message".

Specifying SOMDCOMMDEBUGFLAG=2 enables a display of DATASTREAM information. DATASTREAM information is the actual data sent via a distributed request.

Specifying SOMDCOMMDEBUGFLAG=4 enables a display of DISPATCH information. DISPATCH information is the dispatch of a distributed method request by a SOMobjects server.

To trace multiple types of debugging information, add the flags mathematically, and specify the total. For example, specifying SOMDCOMMDEBUGFLAG=3 enables a display of both TRANSPORT and DATASTREAM information. Specifying SOMDCOMMDEBUGFLAG=7 enables a display of all information above.

**Usage:** Optional

**Scope:** Local

**Default:** If SOM\_TraceLevel is 2 or higher, the default is 7. Otherwise, the default is 0.

### **SOMDRECVWAIT=*number-of-seconds***

The number of seconds to wait for a socket to become readable before generating a communications timeout error.

**Usage:** Optional

**Scope:** Both

**Default:** 30

### **SOMDSENDWAIT=*number-of-seconds***

The number of seconds to wait for a socket to become writable before generating a communications timeout error.

**Usage:** Optional

**Scope:** Both

**Default:** 30

### **MAXIMUMHOPS=*number-of-hops***

The maximum number of location forwarding requests that will be processed before raising an exception.

**Usage:** Optional

**Scope:** Local

**Default:** 1

**SOMDTIMEOUT=number-of-seconds**

The number of seconds to wait for a socket to become readable before generating a communications timeout error.

**Note:** This variable is supplied for compatibility purposes only. IBM strongly encourages you to specify SOMDRECVWAIT instead of this variable. If SOMDRECVWAIT is not set and SOMDTIMEOUT is set, SOMDTIMEOUT will be the timeout value. If SOMDRECVWAIT is specified, then that value will be the timeout value, whether or not SOMDTIMEOUT is set.

**Usage:** Optional

**Scope:** Both

**Default:** 30

## [somir] Stanza (Interface Repository)

This stanza specifies the settings for the SOMobjects Interface Repository.

**SOMIR=name1;name2;...namen**

The name (or list of names) of the files that contain the Interface Repository. The SOMIR keyword variable can reference an ordered list of separate IR files, which process from left to right. Taken as a whole, this gives the appearance of a single, logical interface repository. When running the SOMobjects compiler using the IR emitter, the rightmost data set in the list will be updated. Each data set specification can be either an MVS data set or a file in the hierarchical file system.

When specifying *name* for the SOMIR variable, the following rules apply:

- To specify a fully-qualified MVS data set name, specify // before the name and put quotes around the name, such as:  
`//'SOMMVS.DSNAME'`
- To use the current userid (or prefix) as the high-level qualifier, specify // before the name and omit the quotes, such as:  
`//DSNAME`
- For an HFS file, omit // and specify complete path, such as:  
`/u/user1/filename`

**Examples:**

To use the IBM-supplied data sets, specify:

```
SOMIR='//SOMMVS.SGOSIR';//SOMMVS.SGOSIRSM';
```

To use an HFS file named localir, then the IBM-supplied data sets, specify

```
SOMIR=/u/user1/localir;//SOMMVS.SGOSIR;//SOMMVS.SGOSIRSM';
```

**Usage:** Required

**Scope:** Both

**Default:** No default.

## [somras] Stanza (Error Log and Trace Facility)

This stanza specifies the settings for the SOMobjects error log and trace facilities.

### SOMErrorLogFile=*name*

The name of the file or MVS data set where error log entries will be stored.

IBM recommends that all processes on a system share one error log file. Each data set specification can be either an MVS data set or a file in the hierarchical file system.

When specifying *name* for the SOMErrorLogFile variable, the following rules apply:

- To specify a fully-qualified MVS data set name, specify // before the name and put quotes around the name, such as:

```
//'SOMMVS.SOMERROR.LOG'
```

- To use the current userid (or prefix) as the high-level qualifier, specify // before the name and omit the quotes, such as:

```
//SOMERROR.LOG
```

- For an HFS file, omit // and specify complete path, such as:

```
/u/sommvs/somerror.log
```

**Usage:** Optional

**Scope:** Local

**Default:** somerror.log

**Note:** If the default is taken, the specification is treated as an MVS data set named SOMMVS.SOMERROR.LOG

### SOMErrorLogSize=*size*

The size, in kilobytes, of the error log file. The default allows space for several hundred average-sized log entries.

**Usage:** Optional

**Scope:** Local

**Default:** 128

### SOMErrorLogControl=*filter*

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.

**Usage:** Optional

**Scope:** Local

**Default:** WARNING ERROR MAPPED\_EXCEPTION

### SOMErrorLogDisplayMsgs=NO | YES

Indicates whether or not error log messages should be displayed to the standard output device in addition to the error log specified in the SOMErrorLogFile. IBM recommends that you specify YES during initial installation and when trying to debug a problem. At other times, specify NO.

**Usage:** Optional

**Scope:** Local

**Default:** YES

**MVSTraceLog=*h/q***

The high-level qualifier of the trace data set. This variable must be specified to get trace records written to a data set. If SOM\_TraceLevel is greater than zero and MVSTraceLog is specified, a SOMobjects trace data set is allocated for each process (address space) during initialization of that process. If you specify a value for MVSTraceLog, IBM recommends specifying MVSTraceLog=SOMMVS. The name of the SOMobjects trace data set is a unique name generated by SOMobjects. The name is reported to the user in message GOS20007I.

**Usage:** Optional

**Scope:** Local

**Default:** No default. If this variable is not specified, tracing will be done, but only to an in-memory trace buffer.

**MVSTraceLogSize=*nnn sss***

The size of the SOMobjects trace data set, where:

*nnn* is a decimal value.

*sss* is either TRK or CYL, for tracks and cylinders, respectively.

**Usage:** Optional

**Scope:** Local

**Default:** 100 times the value of SOM\_TraceLevel, in units of TRK (tracks).

## [somsec] Stanza (Security Service)

This stanza specifies the settings for the SOMobjects security service.

**Note:** If RACF PassTicket support is used, you need only specify the DISABLE\_AUTHN variable. In this case, the target application server does not need to be running on the OS/390 platform.

**USER=*userid***

The user ID used to authenticate a distributed SOMobjects client. If the USER variable is not set, the userid associated with the address space under which the client is executing will be used. Note that the system on which the security server runs must have a definition (a RACF user profile) for this user ID.

Never specify the USER keyword in the SOM subsystem's configuration file. Since the SOM subsystem's configuration file values are global, the user ID would be exposed to all SOMobjects applications.

**Usage:** Optional

**Scope:** Local

**Default:** If you do not specify the USER keyword, it will default to the user ID associated with the address space in which the SOMobjects application is executing.

**PASSWD=*password***

The password used to authenticate a distributed SOMobjects client. Corresponds to the user ID that identifies the user of the SOMobjects application. Note that the system on which the security server runs must have a definition for this user ID/password pair. For a SOMobjects application communicating

only with OS/390 servers, PASSWD may be omitted if RACF PassTicket support is enabled. PassTickets offer greater security. For information on enabling PassTicket support, see the “PassTicket” section in the *OS/390 SOMobjects Configuration and Administration Guide*.

Never specify the PASSWD keyword in the SOM subsystem's configuration file. Since the SOM subsystem's configuration file values are global, the password would be exposed to all SOMobjects applications. If PassTicket support is not available, you are establishing a connection with a non-OS/390 server, the best strategy is to specify an appropriate PASSWD value in the local configuration file of each SOMobjects application.

**Usage:** Optional

**Scope:** Local

**Default:** Null

#### **DISABLE\_AUTHN=TRUE | FALSE**

Set DISABLE\_AUTHN to TRUE to turn off authentication. Authentication is enabled by default.

**Note:** A secure server rejects requests from unauthenticated clients.

**Usage:** Optional

**Scope:** Local

**Default:** TRUE

## **[SOM\_POSSOM] Stanza (Persistent Object Service)**

This stanza specifies the settings for the persistent object service environment.

#### **POS\_POMDATA=name**

The path and file name of the POM (persistent object manager) data file. Each data set specification can be either an MVS data set or a file in the hierarchical file system.

When specifying *name* for the SOM\_POSSOM variable, the following rules apply:

- To specify a fully-qualified MVS data set name, specify // before the name and put quotes around the name, such as:  
`//'SOMMVS.SGOSMISC(GOSPMDAT)'`
- To use the current userid (or prefix) as the high-level qualifier, specify // before the name and omit the quotes, such as:  
`//SGOSMISC(GOSPMDAT)`
- For an HFS file, omit // and specify complete path, such as:  
`/u/sommvs/gospmdat`

**Usage:** Optional

**Scope:** Both

**Default:** `//'SOMMVS.SGOSMISC(GOSPMDAT)'`

#### **POSIX\_StreamCreation=LOCAL | NONLOCAL**

The location of the stream for the POSIX protocol. To create the stream in the same process as the POM (persistent object manager), specify NONLOCAL.

The persistent object manager is a class within the SOM persistence framework.

To create the stream in the same process as the PDS (persistent data store), specify LOCAL. The persistent data store is a class within the SOM persistence framework.

**Usage:** Optional

**Scope:** Both

**Default:** NONLOCAL

**BTREE\_StreamCreation=LOCAL | NONLOCAL**

The location of the stream for the BTREE protocol. To create the stream in the same process as the POM, specify NONLOCAL. To create the stream in the same process as the PDS, specify LOCAL.

**Usage:** Optional

**Scope:** Both

**Default:** NONLOCAL



---

## Appendix B. SOM IDL Language Grammar

```
specification : [comment] definition+
definition : type_dcl ; [comment]
 | const_dcl ; [comment]
 | interface ; [comment]
 | module ; [comment]
 | pragma_stm
module : module identifier [comment]
 { [comment] definition+ }
interface : interface identifier
 | interface_dcl
interface_dcl : interface identifier [inheritance] [comment]
 { [comment] export* } [comment]
inheritance : scoped_name {, scoped_name}*
export : type_dcl ; [comment]
 | const_dcl ; [comment]
 | attr_dcl ; [comment]
 | op_dcl ; [comment]
 | implementation_body ; [comment]
 | pragma_stm
scoped_name : identifier
 | :: identifier
 | scoped_name :: identifier
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+
 : [const_expr]

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 : identifier
 | _identifier
stars : *+
```

## Glossary

This glossary defines important terms and abbreviations used across the products that deliver Objects on MVS. This glossary does not contain general data processing terms that can be found in other published works. If you do not find the term you are looking for, refer to the Index or to the *Dictionary of Computing*, SC20-1699.

This glossary includes terms and definitions from:

- The *American National Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.
- The American Production and Inventory Control Society, Inc., *APICS Dictionary*, Sixth Edition, 1987. Definitions are identified by the symbol (APICS) after the definition.

Note: In the following definitions, words shown in *italics* are terms for which separate glossary entries are also defined.

## A

**abstract class (SOM definition).** A *class* that is not designed to be instantiated, but serves as a *base class* for the definition of subclasses. Regardless of whether an abstract class inherits *instance data* and *methods* from *parent classes*, it will always introduce methods that must be *overridden* in a *subclass*, in order to produce a class whose objects are semantically valid.

**abstract class (C++ definition).** (1) A class with at least one pure virtual function that is used as a base class for other classes. The abstract class represents a concept; classes derived from it represent implementations of the concept. You cannot have a direct object of an abstract class. (See also *base class*.) (2) A class that allows polymorphism. There can be no objects of

an abstract class; they are only used to derive new classes.

**access.** Determines whether or not a class member is accessible in an expression or declaration.

**accessory.** A choice or option offered to customers for customizing the end product. An accessory is typically shipped with the end product, not built into it. Examples of accessories are product documentation, car floor mats, and camera cases. One accessory may be usable with more than one end product. (Product documentation may be usable for a product family.) Accessories may be *mandatory* (documentation) or *optional* (camera cases). When more than one choice is offered for an accessory, the choices are referred to as *variants*. For example, product documentation variants would typically provide national language support, variants for camera cases would provide multiple price or function options. See also *feature*.

**address.** A name, label, or number identifying a location in storage.

**addressing mode (AMODE).** In MVS, a program attribute that refers to the address length that a program is prepared to handle upon entry. *IBM*.

**aggregate.** (1) An array or a structure. (2) A compile-time option to show the layout of a structure or union in the listing. (3) An array or a class object with no private or protected members, no constructors, no base classes, and no virtual functions. (4) In programming languages, a structured collection of data items that form a data type. *ISO-JTC1*.

**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).

**alignment.** The storing of data in relation to certain machine-dependent boundaries.

**American National Standards Institute (ANSI).** An organization consisting of producers, consumers, and general interest groups that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. (A)

**AMODE (addressing mode).** In MVS, a program attribute that refers to the address length that a program is prepared to handle upon entry. *IBM*.

**ancestor class.** A *class* from which another class inherits *instance methods*, *attributes*, and *instance variables*, either directly or indirectly. A direct descendant

of an ancestor class is called a *child class*, *derived class*, or *subclass*. A direct ancestor of a class is called a *parent class*, *base class*; or *superclass*.

**angle brackets.** The characters < (left angle bracket) and > (right angle bracket). When used in the phrase "enclosed in angle brackets", the symbol < immediately precedes the object to be enclosed, and > immediately follows it. When describing these characters in the portable character set, the names <less-than-sign> and <greater-than-sign> are used. *X/Open*.

**argument.** (1) A parameter passed between a calling program and a called program. *IBM*. (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called *parameter*. (3) In the shell, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open*.

**array.** In programming languages, an aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting. *IBM*.

**attribute.** Generically, the data that defines a property or characteristic of an object. In IDL, a specialized syntax for declaring "set" and "get" methods. Method names corresponding to attributes always begin with "\_set\_" or "\_get\_". An attribute name is declared in the body of the *interface statement* for a class. Method procedures for get/set methods are automatically defined by the *SOM Compiler* unless an attribute is declared as "noget/noset". Likewise, a corresponding *instance variable* is automatically defined unless an attribute is declared as "nodata". IDL also supports "readonly" attributes, which specify only a "get" method.

## B

**base class.** See *parent class*.

**base class (C++ definition).** A class from which other classes are derived. May itself be derived from another base class. (See also *abstract class*.)

**based on.** The use of existing classes for implementing new classes.

**batch.** See *batch job*.

**batch job.** A job submitted as a predefined series of actions to be performed with little or no interaction between user and system.

**behavior (of an object).** The *methods* that an *object* responds to. These methods are those either introduced or inherited by the *class* of the object. See also *state*.

**bindings.** Language-specific macros and procedures that make implementing and using SOM classes more convenient. These bindings offer a convenient interface to SOM that is tailored to a particular programming language. The SOMobjects for MVS Compiler generates binding files for C and C++. These binding files include an *implementation template* for the class and two header files, one to be included in the class's implementation file and the other in client programs.

**block (in C++).** (1) In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. *ISO-JTC1*. (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. *ISO Draft*. (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

**braces.** The characters { (left brace) and } (right brace), also known as *curly braces*. When used in the phrase "enclosed in (curly) braces" the symbol { immediately precedes the object to be enclosed, and } immediately follows it. When describing these characters in the portable character set, the names <left-brace> and <right-brace> are used. *X/Open*.

**brackets.** The characters [ (left bracket) and ] (right bracket), also known as *square brackets*. When used in the phrase "enclosed in (square) brackets" the symbol [ immediately precedes the object to be enclosed, and ] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open*.

**built-in.** (1) A function which the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. *ISO-JTC1*. Synonymous with *predefined*. *IBM*.

## C

**C library.** A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions.

**C set+.** An object-oriented application development

package for building OS/2 and AIX solutions in C/MVS and C++/MVS.

**case label.** The word case followed by a constant expression and a colon. When the selector evaluates the value of the constant expression, the statements following the case label are processed.

**character.** (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. ANSI. (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multi-byte character), where a single-byte character is a special case of the multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. X/Open. ISO.1.

**character set.** (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, "7-bit Coded Character Set for Information Processing Interchange". ISO Draft. (2) All the valid characters for a programming language or for a computer system. IBM. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. IBM.

**child class (SOM definition).** A class that inherits *instance methods*, *attributes*, and *instance variables* directly from another class, called the *parent class*, *base class*, or *superclass*, or indirectly from an *ancestor class*. A child class may also be called a *derived class* or *subclass*.

**class (SOM definition).** A way of categorizing *objects* based on their behavior (the *methods* they support) and shape (memory layout). A class is a definition of a generic object. In SOM, a class is also a special kind of object that can manufacture other objects that all have a common shape and exhibit similar behavior. The specification of what comprises the shape and behavior of a set of objects is referred to as the "definition" of a class. New classes are defined in terms of existing classes through a technique known as *inheritance*. See also *class object*.

**class (C++ definition).** (1) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. Classes may be defined hierarchically, allowing one class to be derived from another, and may restrict access to its members. (2) A user-defined data type that can contain both data representations (data members) and functions (member functions).

**class library.** A collection of C++ classes.

**class manager.** An *object* that acts as a run-time registry for all SOM *class objects* that exist within the current process and which assists in the dynamic loading and unloading of class libraries. A class implementor can define a customized class manager by subclassing *SOMClassMgr* class to replace the SOM-supplied *SOMClassMgrObject*. This is done to augment the functionality of the default class-management registry (for example, to coordinate the automatic quiescing and unloading of classes).

**class method.** A class method is a *method* that a *class object* responds to (as opposed to an *instance method*). A class method that class <X> responds to is provided by the *metaclass* of class <X>. Class methods are executed without requiring any *instances* of class <X> to exist, and are frequently used to create instances of the class.

**class object.** The run-time *object* representing a SOM *class*. In SOM, a class object can perform the same behavior common to all *objects*, inherited from *SOMObject*.

**class variable.** *Instance data* of a *class object*. All instance data of an *object* is defined (through either *introduction* or *inheritance*) by the object's class. Thus, class variables are defined by *metaclasses*.

**client.** (1) A user. (2) A functional unit that receives shared services from a server.

**client code.** (Or *client program* or *client*.) An application program, written in the programmer's preferred language, which invokes *methods* on *objects* that are *instances* of SOM *classes*.

**CLIST.** An executable list of TSO commands.

**collection.** (1) An abstract class without any ordering, element properties, or key properties. All abstract classes are derived from collection. (2) In a general sense, an implementation of an abstract data type for storing elements.

**column.** (1) One of two or more vertical arrangements of lines, positioned side by side on a page or screen. (T) (2) A vertical arrangement of characters or other expressions. (A) (3) A character position within a print line or on a display. The positions are numbered from 1, by 1, starting at the leftmost character position and extending to the rightmost position. (4) In SQL\*, the vertical part of a table. A column has a name and a particular data type; for example, character, decimal, or integer.

**comment.** (1) Any descriptive information whose format depends on the context in which the comment exists (a source code comment, a folder comment, etc.). (2) In programming languages, a language con-

struct for the inclusion of text in a program and having no impact on the execution of the program. Comments are used to explain certain aspects of the program. (I) (3) A statement used to document a program or file. Comments include information that may be helpful in running a job or reviewing an output listing. (4) Synonymous with computer program annotation, note, remark. Comments serve as documentation instead of as instructions. They are not processed by a compiler.

**completion status.** A code that indicates a review or deliverable has completed its cycle.

**const.** (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change.

**constant.** (1) In programming languages, a language object that takes only one specific value. *ISO-JTC1*. (2) A data item with a value that does not change. *IBM*.

**constant expression.** An expression having a value that can be determined during compilation and that does not change during program execution.

**constructor (C++ definition).** A class member function with the same name as its class, used to construct class objects and sometimes to initialize them.

**context expression.** An optional expression in a method's IDL declaration, specifying identifiers whose value (if any) can be used during SOM's *method resolution* process and/or by the *target object* as it executes the *method procedure*. If a context expression is specified, then a related Context parameter is required when the method is invoked. (This Context parameter is an *implicit parameter* in the IDL specification of the method, but it is an explicit parameter of the method's procedure.) No SOM-supplied methods require context parameters.

**conversion (C++ definition).** (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost due to conversion since accuracy of data representation varies among different data types. *ISO-JTC1*. (2) The process of changing from one method of data processing to another or from one data processing system to another. *IBM*. (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM*. (4) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

**CORBA.** The Common Object Request Broker Architecture established by the Object Management Group.

IBM's *Interface Definition Language* used to describe the *interface* for SOM classes is fully compliant with CORBA standards.

**current working directory.** (1) A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash. *X/Open ISO-1*. (2) In DOS, the directory that is searched when a filename is entered with no indication of the directory that lists the filename. DOS assumes that the current directory is the root directory unless a path to another directory is specified. *IBM*. (3) In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM*. (4) In the AIX operating system, a directory that is active and that can be displayed. Relative path name resolution begins in the current directory. *IBM*.

**customize.** To adapt or modify an application to fit a unique environment using build-time facilities provided as part of the application. Two kinds of customizing are available. The information services department customizes the application to fit the information processing system facilities available; for example, libraries, databases, workstations, communications facilities, and so on. The application specialists customize enterprise and application profiles to match the business policies and practices applicable to all users in the enterprise, or all users of one application. An example is changing a user interface dialog or panel. Users may personalize applications to further suit their unique needs. See also *personalize*.

## D

**data definition (DD).** (1) In C/MVS and C++/MVS, a definition that describes a data object, reserves storage for a data object, and can provide an initial value for a data object. A data definition appears outside a function or at the beginning of a block statement. *IBM*. (2) A program statement that describes the features of, specifies relationships of, or establishes context of, data. *ANSI*. (3) A statement that is stored in the environment and that externally identifies a file and the attributes with which it should be opened.

**data structure.** The internal data representation of an implementation.

**data type.** The properties and internal representation that characterize data.

**database.** A collection of data with a given structure for accepting, storing, and providing, on demand, data for multiple users. (T)

**debug.** To detect, locate, and correct errors in a program.

**declaration.** (1) In C/MVS and C++/MVS, a description that makes an external object or function available to a function or a block statement. *IBM*. (2) Establishes the names and characteristics of data objects and functions used in a program.

**declarator.** Designates a data object or function declared. Initializations can be performed in a declarator.

**declare.** To identify the variable symbols to be used at preassembly time.

**default.** A value, attribute, or option that is assumed when no alternative is specified by the programmer.

**default implementation.** One of several possible implementation variants offered as the default for a specific abstract data type.

**define directive.** A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

**definition.** (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

**delete.** (1) The keyword `delete` identifies a free store deallocation operator. (2) The `delete` operator is used to destroy objects created by `new`.

**derivation.** In C++, to derive a class, called a derived class, from an existing class, called a base class.

**derived class (SOM definition).** See *subclass* and *subclassing*.

**derived class (C++ definition).** A class that inherits the proper base class become members of a derived class object. You can add additional data members and member functions to the derived class. A derived class object can be manipulated as if it is a base class object. The derived class can override virtual functions of the base class.

**derived metaclass.** (Or *SOM-derived metaclass*.) A *metaclass* that SOM creates automatically (often even when the *class* implementor specifies an explicit metaclass) as needed to ensure that, for any code that executes without *method-resolution* error on an *instance* of a given class, the code will similarly execute without *method-resolution* error on instances of any *subclass* of the given class. SOM's ability to derive such metaclasses is a fundamental necessity in order to ensure binary compatibility for client programs despite any subsequent changes in class *implementations*.

**descendant.** Any class that has the current class in its inheritance hierarchy.

**descriptor (SOM definition).** (Or *method descriptor*.) An ID representing the identifier of a *method* definition or an *attribute* definition in the Interface Repository. The IR definition contains information about the method's return type and the type of its arguments.

**descriptor (C++ definition).** PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during run time.

**difference.** Given two sets A and B, the difference (A-B) is the set of all elements contained in A but not in B. For bags, there is an additional rule for duplicates: If bag P contains an element  $m$  times and bag Q contains the same element  $n$  times, then, if  $m > n$ , the difference contains that element  $m-n$  times. If  $m \leq n$ , the difference contains that element zero times.

**directive.** A message (a pre-defined character constant) received by a *replica* from the Replication Framework. Indicates a potential failure situation.

**dispatch-function resolution.** Dispatch-function resolution is the slowest, but most flexible, of the three *method-resolution* techniques SOM offers. Dispatch functions permit method resolution to be based on arbitrary rules associated with an *object's class*. Thus, a class implementor has complete freedom in determining how methods invoked on its *instances* are resolved. See also *dispatch method* and *dynamic dispatching*.

**dispatch method.** A *method* (such as `somDispatch` or `somClassDispatch`) that is invoked (and passed an argument list and the ID of another method) in order to determine the appropriate *method procedure* to execute. The use of dispatch methods facilitates *dispatch-function resolution* in SOM applications. See also *dynamic dispatching*.

**DLL (dynamic link library).** A collection of functions and variables accessed by external programs. A DLL identifies all functions, methods, and attributes of classes.

**double-quote.** The character ", also known as *quotation mark*. X/Open.

**driver.** A system or device that enables a functional unit to operate.

**dump.** To copy data in a readable format from main or auxiliary storage onto an external medium such as tape, diskette, or printer. *IBM*.

**dynamic.** Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM*.

**dynamic link library (DLL).** A collection of functions and variables accessed by external programs. A DLL identifies all functions, methods, and attributes of classes.

## E

**element.** The component of an array, subrange, enumeration, or set.

**emitter.** Generically, a program that takes the output from one system and converts the information into a different form. Using the Emitter Framework, selected output from the *SOM Compiler* (describing each syntactic unit in an *IDL source file*) is transformed and formatted according to a user-defined template. Example emitter output, besides the implementation template and language bindings, might include reference documentation, class browser descriptions, or “pretty” printouts.

**empty string.** (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

**enabler.** A set of build-time tools provided to simplify and control the process of creating programs or data sets.

**encapsulation (SOM definition).** An object-oriented programming feature whereby the implementation details of a class are hidden from client programs, which are only required to know the *interface* of a *class* (the signatures of its *methods* and the names of its *attributes*) in order to use the class's methods and attributes.

**encoder/decoder.** In the Persistence Framework, a *class* that knows how to read/write the persistent object format of a *persistent object*. Every persistent object is associated with an Encoder/Decoder, and an encoder/decoder object is created for each *attribute* and *instance variable*. An Encoder/Decoder is supplied by the Persistence Framework by default, or an application can define its own.

**enterprise.** The entire business organization under discussion. It may consist of one or many establishments, divisions, plants, warehouses, and so on. With respect to systems, enterprise refers to a single installation.

**entry class.** In the Emitter Framework, a *class* that represents some syntactic unit of an *interface* definition in the *IDL source file*.

**entity.** Any concrete or abstract thing of interest, including associations among things; for example, a person, object, event, or process that is of interest in

the context under consideration, and about which data may be stored in a database. (T)

**enumeration type.** A distinct data type that is not an integral type. An enumeration type defines a set of enumeration constants.

**enumerator.** An enumeration constant and its associated value.

**Environment parameter.** A CORBA-required parameter in all *method procedures*, it represents a memory location where exception information can be returned by the *object* of a method invocation. [Certain methods are exempt (when the class contains a modifier of *callstyle=oidl*), to maintain upward compatibility for client programs written using an earlier release.]

**error.** A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

**exception (C++ definition).** (1) Any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called throwing the exception). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA *ISO-JTC1*.

**exception handling.** A type of error handling that allows control and information to be passed to an exception handler when an exception occurs. **Try** blocks, **catch** blocks and **throw** expressions are the constructs used to implement formal exception handling in C++.

**expression.** A representation of a value. For example, variables and constants appearing alone or in combination with operators are expressions.

**extension.** (1) One kind of modification of an application by an enterprise; function is added without changing any function present in the application.

**Note:** next 2 definitions are from C++ (2) An element or function not included in the standard language.  
(3) File name extension.

## F

**feature.** (1) A choice or option available to customize the end product. A feature is built into the end product during production, not just shipped with it. A feature may be *mandatory (standard)*—that is, one of the available choices (*variants*) must be chosen; for example, engines and transmissions in cars; or *optional*—none of

the variants for this feature need be chosen; for example, radios or sun roofs in cars. There are *production features*, which define allowable configurations of a given product, with associated routings and technical specifications necessary, and *sales features*, which define choices or options the customer can order. In some cases, sales and production features are identical, but in others, one sales feature may translate into several production features. See also *accessory*, *variant*. (2) In object-oriented programming, a collective term for methods and attributes.

**filter.** A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output. Typically, its function is to perform some transformation on the data stream. *X/Open*.

**first element.** The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

**framework.** Pre-built class libraries that act as off-the-shelf modules for building applications to solve specific business problems.

**function definition.** The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, parameter declarations, and a block statement (the function body).

## G

**generic class.** See *class templates*.

**given.** Referring to a collection, element or function that is given as a function argument.

## H

**header.** System-defined control information that precedes user data.

**header data.** The data that provides the identification and description of an object.

**header file.** A file that contains system-defined control information that precedes user data. Also known as an *include* file.

**hierarchy (C++ definition).** A structure that has different ranks or levels.

## I

**ID.** See *somId*.

**identifier (C++ definition).** (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI*. (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *ANSI*. (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM*.

**IDL source file.** A user-written .idl file, expressed using the syntax of the *Interface Definition Language* (IDL), which describes the *interface* for a particular *class* (or *classes*, for a *module*). The IDL source file is processed by the *SOM Compiler* to generate the *binding files* specific to the programming languages of the class implementor and the client application. (This file may also be called the “IDL file,” the “source file,” or the “interface definition file.”)

**implementation.** (Or *object implementation*.) The specification of what *instance variables* implement an *object's state* and what *procedures* implement its *methods* (or *behaviors*).

**implementation statement.** An optional declaration within the body of the *interface* definition of a *class* in a SOM *IDL source file*, specifying information about how the class will be implemented (such as, version numbers for the class, overriding of inherited methods, or type of method resolution to be supported by particular methods). This statement is a SOM-unique statement; thus, it must be preceded by the term “#ifdef \_\_SOMIDL\_\_” and followed by “#endif”. See also *interface declaration*.

**implementation template.** A template file containing *stub procedures* for *methods* that a *class* introduces or *overrides*. The implementation template is one of the *binding files* generated by the *SOM Compiler* when it processes the *IDL source file* containing class *interface declarations*. The class implementor then customizes the *implementation*, by adding language-specific code to the *method procedures*.

**IMS (Information Management System).** A database/data communication (DB/DC) system that can manage complex databases and networks. *IBM*.

**include directive.** A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

**incremental update.** A revision to an *implementation template* file that results from reprocessing of the *IDL source file* by the *SOM Compiler*. The updated implementation file will contain new *stub procedures*, added

comments, and revised *method prototypes* reflecting changes made to the *method definitions* in the IDL specification. Importantly, these updates do not disturb existing code that the class implementor has defined for the prior method procedures.

**index.** A list of the contents of a file or of a document, together with keys or references for locating the contents. (I) (A)

**Information Management System (IMS).** A database/data communication (DB/DC) system that can manage complex databases and networks. IBM.

**inheritance (SOM definition).** The technique of defining one *class* (called a *subclass*, *derived class*, or *child class*) as incremental differences from another class (called the *parent class*, *base class*, *superclass*, or *ancestor class*). From its parents, the subclass inherits variables and *methods* for its *instances*. The subclass can also provide additional *instance variables* and methods. Furthermore, the subclass can provide new procedures for implementing inherited methods. The subclass is then said to *override* the parent class's methods. An overriding method procedure can elect to call the parent class's *method procedure*. (Such a call is known as a *parent method call*.)

**inheritance (C++ definition).** An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

**inheritance hierarchy.** The sequential relationship from a root class to a subclass, through which the subclass inherits *instance methods*, *attributes*, and *instance variables* from all of its ancestors, either directly or indirectly. The root class of all SOM classes is SOMObject.

**install.** (1) To add a program, program option, or software to a system in such a manner that it runs and interacts properly with all affected programs in the system. (2) To connect hardware to a system.

**installation.** In system development, preparing and placing a functional unit in position for use. (T)

**instance (SOM definition).** (Or *object instance* or just *object*.) A specific object, as distinguished from a *class* of objects. See also *object*.

**instance (C++ definition).** An object-oriented programming term synonymous with 'object'. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class box is previously defined, two instances of a class box could be instantiated with the declaration:

```
box box1, box2;
```

**instance data.** A general term for the set of instance data variables specified for an object.

**instance variable.** Variables declared for access by *method procedures* of a *class*. An instance variable is declared within the body of the *implementation statement* in a SOMobjects for MVS

*IDL source file.* An instance variable is "private" to the class and should not be accessed by a client program.

**instantiate.** To create or generate a particular instance (or object) of a data type or class of objects.

**integer.** A positive or negative whole number, that is, an optional sign followed by a number that does not contain a decimal place or zero.

**interface.** The information that a *client* must know to use a *class* namely, the names of its *attributes* and the signatures of its *methods*. The interface is described in a formal language (the *Interface Definition Language*, IDL) that is independent of the programming language used to implement the class's methods.

**interface declaration.** (Or *interface statement*.) The statement in the *IDL source file* that specifies the name of a new class and the names of its *parent class(es)*. The "body" of the interface declaration defines the *signature* of each new *method* and any *attribute(s)* associated with the class. In SOM IDL, the body may also include an *implementation statement* (where *instance variables* are declared or a *modifier* is specified, for example to *override* a method).

**Interface Definition Language (IDL).** The formal language (independent of any programming language) by which the *interface* for a *class of objects* is defined in a .idl file, which the *SOM Compiler* then interprets to create an *implementation template* file and *binding* files. SOM's Interface Definition Language is fully compliant with standards established by the Object Management Group's Common Object Request Broker Architecture (CORBA).

**Interface Repository (IR).** The database that SOM optionally creates to provide persistent storage of objects representing the major elements of *interface* definitions. Creation and maintenance of the IR is based on information supplied in the *IDL source file*. The SOM IR Framework supports all interfaces described in the CORBA standard.

**Interface Repository Framework.** A set of *classes* that provide *methods* whereby executing programs can access the persistent objects of the *Interface Repository* to discover everything known about the programming *interfaces* of SOM classes.

**interoperability.** Under MVS, resources existing in parent processes do not exist in a forked child process.

## J

**job class.** Any one of a number of job categories that can be defined. By classifying jobs and directing initiator/terminators to initiate specific classes of jobs, it is possible to control a mixture of jobs that can be performed concurrently.

## K

**key.** A data member that is used to identify an element. A key only is a part of a complete element that can be used, for example, to establish the order of elements in a collection.

**level.** In a tree-structured object, the level defines the relation between the root and a particular node. The root is the highest level, nodes adjacent to the root form the next level, and so on.

**library (C++ definition).** (1) A collection of functions, calls, subroutines, or other data. *IBM*. (2) A set of object modules that can be specified in a link command.

**line.** A sequence of zero or more non-newline characters plus a terminating newline character. *X/Open*.

**literal (C++ Definition).** (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, "APRIL" represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1*. (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM*. (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal "CHARACTERS" has the value CHARACTERS. *IBM*.

**local.** (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. *ISO-JTC1*. (2) Pertaining to that which is defined and used only in one subdivision of a computer program. *ANSI*.

**local variable.** A variable that is only visible and usable in a single method.

**location.** (1) The site responsible for an object. (2) An identified place of significance to an enterprise, such as a work place, plant, or cost center. Manufacturing locations, cost-estimating locations, and engineering locations can be identified.

## M

**macro (SOM definition).** 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*.

**macro (C++ definition).** An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor `#define` directive.

**map.** An unordered flat collection that uses keys and has element equality.

**mapping.** In a database, the establishing of correspondences between a given logical structure and a given physical structure. (T)

**member.** A data object or function in a structure, union or class. Members can also be classes, enumerations, bit fields and type names.

**memory leak.** Occurs when dynamic memory that has been allocated to an application is not freed by the application when the memory is no longer needed.

**message.** (1) The primary mechanism for objects to communicate with each other. (2) Text displayed to the user to supply important information or to prompt action. Messages are defined in message members using the <MSG> and <VARSUB> dialog tags.

**metaclass.** A *class* whose *instances* are class objects. In SOM, any class descended from *SOMClass* is a metaclass. The *methods* a class inherits from its metaclass are called *class methods*.

**metaclass incompatibility.** A situation where a *subclass* does not include all of the *class variables* or respond to all of the *class methods* of its *ancestor classes*. This situation can easily arise in *OOP* systems that allow programmers to explicitly specify *metaclasses*, but is not allowed to occur in SOM. Instead, SOM automatically prevents this by creating and using *derived metaclasses* whenever necessary.

**method (SOM definition).** A combination of a *procedure* and a name, such that many different procedures can be associated with the same name. In object-oriented programming, invoking a method on an *object* causes the object to execute a specific *method procedure*. The process of determining which method procedure to execute when a method is invoked on an object is called *method resolution*. (The CORBA standard uses the term "operation" for method invocation). SOM supports two different kinds of methods: static methods and dynamic methods. See also *static method* and *dynamic method*.

**method (C++ definition).** Method is an object-oriented programming term synonymous with member function.

**method descriptor.** See *descriptor*.

**method ID.** A number representing a zero-terminated string by which SOM uniquely represents a *method* name. See also *somId*.

**method procedure.** A function or procedure, written in an arbitrary programming language, that implements a *method* of a *class*. A method procedure is defined by the class implementor within the *implementation template* file generated by the *SOM Compiler*.

**method resolution.** The process of selecting a particular *method procedure*, given a *method* name and an *object instance*. The process results in selecting the particular function/procedure that implements the abstract method in a way appropriate for the designated object. SOM supports a variety of method-resolution mechanisms, including *offset method resolution*, *name-lookup resolution*, and *dispatch-function resolution*.

**method table.** A table of pointers to the *method procedures* that implement the *methods* that an *object* supports. See also *method token*.

**method token.** A value that identifies a specific *method* introduced by a *class*. A method token is used during *method resolution* to locate the *method procedure* that implements the identified method. The two basic method-resolution procedures are *somResolve* (which takes as arguments an *object* and a method token, and returns a pointer to a procedure that implements the identified method on the given object) and *somClassResolve* (which takes as arguments a *class* and a method token, and returns a pointer to a procedure that implements the identified method on an instance of the given class).

**mode.** A collection of attributes that specifies a file's type and its access permissions. *X/Open. ISO.1.*

**model.** A representation of a process or system that attempts to relate the most important variables in the system in such a way that analysis of the model leads to insights into the system. The model is typically used to anticipate the result of some particular strategy in the real system.

**modifier.** Any of a set of statements that control how a *class*, an *attribute*, or a *method* will be implemented. Modifiers can be defined in the *implementation statement* of a SOM *IDL source file*. The implementation statement is a SOM-unique extension of the *CORBA* specification. [User-defined modifiers can also be specified for use by user-written emitters or to store information in the *Interface Repository*, which can then be accessed via methods provided by the *Interface Repository Framework*.]

**module (SOM definition).** 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.

**module (C++ definition).** A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

**multibyte character.** A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

**multiple inheritance (SOM definition).** The situation in which a *class* is derived from (and inherits *interface* and *implementation* from) multiple parent classes.

**multiple inheritance (C++ definition).** An object-oriented programming technique implemented in C++ through derivation, in which the derived class inherits members from more than one base class. See also *inheritance*.

## N

**name space.** A category used to group similar types of identifiers.

**naming scope.** See *scope*.

**network.** An arrangement of nodes and connecting branches. (T)

**new.** A keyword identifying a free store allocation operator. The *new* operator may be used to create class objects.

**node.** In a tree structure, a point at which subordinate items of data originate. *ANSI*.

**NULL.** In C/MVS and C++/MVS, a pointer that does not point to a data object. *IBM*.

## O

**object (SOM definition).** (Or *object instance* or just *instance*.) An entity that has *state* (its data values) and *behavior* (its *methods*). An object is one of the elements of data and function that programs create, manipulate, pass as arguments, and so forth. An object is a way to encapsulate state and behavior.

*Encapsulation* permits many aspects of the *implementation* of an object to change without affecting client programs that depend on the object's behavior. In SOM, objects are created by other objects called *classes*.

**object (C++ definition).** (1) A region of storage. An object is created when a variable is defined or new is invoked. An object is destroyed when it goes out of scope. (See also *instance*.) (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. *IBM*. (3) An instance of a class.

**object implementation.** See *implementation*.

**object-oriented programming (C++ definition).** A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished but instead on what data objects comprise the problem and how they are manipulated.

**object reference (SOM definition).** A CORBA term denoting the information needed to reliably identify a particular *object*.

**object request broker (ORB).** See *ORB*.

**OIDL.** The original language used for declaring SOM classes. The acronym stands for Object Interface Definition Language. OIDL is still supported by SOM release 2, but it does not include the ability to specify *multiple inheritance* classes.

**OOP.** An acronym for “object-oriented programming.”

**operating system.** Software that controls the execution of programs and that may provide services such as resource allocation, scheduling, input/output control, and data management. Although operating systems are predominantly software, partial hardware implementations are possible. (T)

**operation.** See *method*.

**operator.** A symbol (such as +, -, \*) that represents an operation (in this case, addition, subtraction, multiplication).

**overflow.** (1) That portion of an operation's result that exceeds the capacity of the intended unit of storage. (2) That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM*.

**override.** (Or *overriding method*.) The technique by which a *class* replaces (redefines) the *implementation* of a *method* that it inherits from one of its *parent classes*. An overriding method can elect to call the parent class's *method procedure* as part of its own implementation. (Such a call is known as a *parent method call*.)

## P

**parameter (C++ term).** (1) In C/MVS and C++/MVS, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open*. (2) Data passed between programs or procedures. *IBM*.

**parameter declaration.** A description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

**parent.** A node that has children in a tree structure.

**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*.

**parser.** See *dialog tag parser*.

**pattern.** A sequence of characters used either with regular expression notation or for pathname expansion, as a means of selecting various characters strings or pathnames, respectively. The syntaxes of the two patterns are similar, but not identical. *X/Open*.

**persistent object.** An *object* whose *state* can be preserved beyond the termination of the *process* that created it. Typically, such objects are stored in a persistent store such as files, a database, etc..

**personalize.** To change an application so as to apply only to an individual's use of the application; for example, setting language and time/date formats. See also *customize*.

**pointer.** A variable that holds the address of a data object or function.

**polymorphism (SOM definition).** An object-oriented programming feature where a method name can denote more than one *method procedure*. For example, when a SOM class overrides a parent class definition of a method to change its behavior. The term literally means “having many forms.”

**polymorphism (C++ definition).** The technique of taking an abstract view of an object or function and using any concrete objects or arguments that are derived from this abstract view.

**precedence.** The priority system for grouping different types of operators with their operands.

**preprocessor.** A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

**primitive.** A data type that is supported by the C programming language. A primitive cannot receive message calls.

**principal.** The user on whose behalf a particular (remote) *method* call is being performed.

**private.** Pertaining to a class member that is only accessible to member functions and friends of that class.

**private IDL.** That part of an IDL specification that a class builder has identified as being hidden from a user.

**procedure.** A small section of code that executes a limited, well-understood task when called from another program. In SOM, a *method procedure* is often referred to as a procedure. See also *method procedure*.

**process (C++ definition).** (1) An instance of an executing application and the resources it uses. (2) An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the *fork()* function. The process that issues the *fork()* function is known as the parent process, and the new process created by the *fork()* function is known as the child process. *X/Open. ISO.1.*

**process ID.** The unique identifier representing a process. A process ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid\_t*.) A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A process that is not a system process will not have a process ID of 1. *X/Open. ISO.1.*

**protected.** A protected member of a class is accessible to member functions and friends of that class, or member functions and friends of classes derived from that class.

**prototype.** A function declaration or definition that includes both the return type of the function and the types of its parameters. See *function prototype*.

**public.** A public member of a class is accessible to all functions.

**public IDL.** That part of an IDL specification that a class builder has identified as being accessible by a user.

## Q

**qualified name.** Used to qualify a nonclass type name such as a member by its class name.

**queue.** A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

## R

**receiver.** See *target object*.

**relationship.** A definition of the properties that exist between two entities.

**replica.** When an object is replicated among a set of processes (using the Replication Framework), each process is said to have a replica of the object. From the view point of any application model, the replicas together represent a single object.

**replicated object.** An *object* for which *replicas* (copies) exist. See *replica*.

**return code.** A value returned to a program to indicate the results of an operation requested by that program.

**root.** (1) A node that has no parent. All other nodes of a tree are descendants of the root. (2) In the AIX operating system, the user name for the system user with the most authority. *IBM*. (3) In the OS/2 operating system, the base directory.

## S

**S-name.** An external non-C++ name in an object module produced by compiling with the *NOLONGNAME* option. Such a name is up to 8 characters long and single case.

**scope (SOM definition).** (Or *naming scope*.) That portion of a program within which an identifier name has "visibility" and denotes a unique variable. In SOM, an *IDL source file* forms a scope. An identifier can only be defined once within a scope; identifiers can be redefined within a nested scope. In a *.idl* file, modules, interface statements, structures, unions, methods, and exceptions form nested scopes.

**scope (C++ definition).** (1) That part of a source program in which a variable is visible. (2) That part of a source program in which an object is defined and recognized.

**sequence.** A sequentially ordered flat collection.

**services.** Routines complementary to the application transaction support.

**set.** An unordered flat collection with element equality.

**shadowing.** In the Emitter Framework, a technique that is required when any of the *entry classes* are subclassed. Shadowing causes instances of the new subclass(es) (rather than instances of the original entry classes) to be used as input for building the object graph, without requiring a recompile of emitter framework code. Shadowing is accomplished by using the macro SOM\_SubstituteClass.

**signal.** (1) A condition that may or may not be reported during program execution. For example, SIGFPE is the signal used to represent erroneous arithmetic operations such as a division by zero. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open. ISO.1.* (3) In AIX operating system operations, a method of inter-process communication that simulates software interrupts. *IBM.*

**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).

**Sockets class.** A class that provides a common communications interface to distributed SOMobjects, the Replication Framework, and the Event Management Framework. The Sockets class provides the base interfaces (patterned after TCP/IP sockets); the subclasses TCPIPSockets, NB.Sockets, and IPX.Sockets provide actual implementations for TCP/IP, Netbios, and Netware IPX/SPX, respectively.

**SOM Compiler.** A tool provided by the SOM Toolkit that takes as input the interface definition file for a class (the .idl file) and produces a set of *binding files* that make it more convenient to implement and use SOM classes.

**SOMClass.** One of the three primitive *class objects* of the SOM run-time environment. SOMClass is the root (meta)class from which all subsequent *metaclasses* are derived. SOMClass defines the essential *behavior* common to all SOM *class objects*.

**SOMClassMgr.** One of the three primitive *class objects* of the SOM run-time environment. During SOM initialization, a single *instance (object)* of SOMClassMgr is created, called SOMClassMgrObject. This object

maintains a directory of all SOM classes that exist within the current process, and it assists with dynamic loading and unloading of class libraries.

**SOM-derived metaclass.** See *derived metaclass*.

**somId.** A pointer to a number that uniquely represents a zero-terminated string. Such pointers are declared as type somId. In SOM, somId's are used to represent *method names*, *class names*, and so forth.

**SOMObject.** One of the three primitive *class objects* of the SOM run-time environment. SOMObject is the root class for all SOM (sub)classes. SOMObject defines the essential *behavior* common to all SOM *objects*.

**somSelf.** Within *method procedures* in the *implementation* file for a class, a parameter pointing to the *target object* that is an *instance* of the *class* being implemented. It is local to the *method procedure*.

**somThis.** Within *method procedures*, a local variable that points to a data structure containing the *instance variables* introduced by the *class*. If no instance variables are specified in the SOM *IDL source file*, then the somThis assignment statement is commented out by the *SOM Compiler*.

**stack.** A sequence with restricted access in which elements are added to the top and removed from the top. A stack is characterized by last-in, first-out behavior and reverse chronological order.

**standard output.** (1) An output stream usually intended to be used for primary data output. *X/Open.* (2) In the AIX operating system, the primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM.*

**state (of an object).** The data (*attributes*, *instance variables* and their values) associated with an *object*. See also *behavior*.

**static.** A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

**status.** The condition or situation of an object. The status of an object determines the activities that may be performed on that object.

**status code.** Indicates the status of an object, for example, created, distributed, recalled, deleted, etc.

**string.** In dialog tags, a series of characters processed as a single item. Strings with multiple words require quotation marks around them and are referred to as *quoted strings*.

**structure.** A construct (a class data type) that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

**stub procedures.** *Method procedures* in the *implementation template* generated by the *SOM Compiler*. They are procedures whose bodies are largely vacuous, to be filled in by the implementor.

**subclass.** A *class* that inherits *instance methods*, *attributes*, and *instance variables* directly from another *class*, called the *parent class*, *base class*, *superclass*, or indirectly from an *ancestor class*. A subclass may also be called a *child class* or *derived class*.

**subclassing.** The process whereby a new *class*, as it is created (or *derived*), inherits *instance methods*, *attributes*, and *instance variables* from one or more previously defined *ancestor classes*. The immediate *parent class(es)* of a new class must be specified in the class's *interface declaration*. See also *inheritance*.

**subset.** Given two sets A and B, B is a subset of A if and only if all elements of B are also elements of A.

**suffix.** The part of a dialog tag that follows the > in the tag. Suffixes are usually used to code literal information.

**switch statement.** A C++ language statement that causes control to be transferred to one of several statements depending on the value of an expression.

**symbol.** In the Emitter Framework, any of a (standard or user-defined) set of names (such as, className) that are used as placeholders when building a text template to pattern the desired *emitter* output. When a template is emitted, the symbols are replaced with their corresponding values from the emitter's symbol table. Other symbols (such as, classSN) have values that are used by section-emitting methods to identify major sections of the template (which are correspondingly labeled as "classS" or by a user-defined name).

## T

**table.** An array of data each item of which can be unambiguously identified by means of one or more arguments.

**target object.** (Or *receiver*.) The object responding to a *method* call. The target object is always the first

formal parameter of a *method procedure*. For SOM's C-language bindings, the target object is the first argument provided to the method invocation macro, \_methodName.

**task.** (1) In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. *ISO-JTC1. ANSI*. (2) A routine that is used to simulate the operation of programs. Tasks are said to be *nonpreemptive* because only a single task is executing at any one time. Tasks are said to be *lightweight* because less time and space are required to create a task than a true operating system process.

**template.** A family of classes or functions with variable types.

**template class.** A class instance generated by a class template.

**this.** A keyword that identifies a special type of pointer that references in a member function the class object with which the member function was invoked.

**type.** The description of the data and the operations that can be performed on or by the data. Also see *data type*.

**type code.** A code that identifies the kind of object. Type codes for routings include conceptual, master, and individual.

**type definition.** A definition of a data type.

## U

**union.** (1) In C/MVS or C++/MVS, a variable that can hold any one of several data types, but only one data type at a time. *IBM*. (2) For bags, there is an additional rule for duplicates: If bag P contains an element m times and bag Q contains the same element n times, then the union of P and Q contains that element m+n times.

**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.

**using object.**

## V

**value logging.** In the Replication Framework, a technique for maintaining consistency among *replicas* of a replicated object, whereby the new value of the object is distributed after the execution of a method that updates the object.

**variable.** In SOMobjects for MVS, the data that defines a class of objects and that holds (or contains) a value in an instance of that class. A data variable can define a class and its instances (objects) or a metaclass and its instances (class objects). They are sometimes referred to, respectively, as instance variables and class variables

**variable value.** In SOMobjects for MVS, the data that is contained in an instantiation of a data variable as a program is executing.

**variant.** One of the allowable choices for an accessory or a feature. See also *accessory, feature*.

## W

**warning message.** In SAA Common User Access architecture, a message that tells a user that a requested action has been suspended because something undesirable could occur. A user can continue the requested action, withdraw the requested action, or get help.

**white space.** (1) Space characters, tab characters, form-feed characters, and new-line characters. (2) A sequence of one or more characters that belong to the space character class as defined via the LC\_CTYPE category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), newline characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open*.

**+0 (—0).** An algebraic sign provides additional information about any variable that has the value zero. Although all precisions have distinct representations for +0, -0, +Inf, and -Inf, the signs are significant in some circumstances, such as division by zero, and not in others. *X/Open*.



# Index

## Special Characters

-maddstar compiler option 4-13  
'new' operator in C++ client programs 11-15  
new' operator in C++ client programs 6-11  
<className>\_Class\_Source symbol 8-18  
<className>\_MajorVersion constant 6-21  
<className>\_MinorVersion constant 6-21  
<className>New macro 6-11  
<className>NewClass procedure  
    for creating class objects 6-21  
'any' IDL type 3-7  
\*  
    adding to interface references  
        SMADDSTAR keyword in [somc] stanza A-5  
||>\_||> macro 6-13  
||>New macro 6-13  
#ifdef \_\_SOMIDL\_\_ statement 3-40, 8-11  
#include directive in implementation templates 3-2,  
    8-18  
    IDL syntax of 3-5

## A

abstract modifier 3-27  
Abstract syntax graph 14-3  
abstractparents modifier 3-27  
activate\_impl\_failed method  
activation policies  
    distributed SOMobjects servers 7-40  
add\_arg method  
add\_class\_toImpldef method  
add\_Impldef method  
add\_item method  
addcmt compiler option 4-13  
addstar compiler option 4-13, 6-7  
Advanced server programming in DSOM  
After methods  
Aggregate type 15-18  
alignment method 15-20  
Ancestor class 11-26  
Ancestor initialization with somDefault  
    method 11-9  
ARG\_INOUT flag value  
ARG\_OUT flag value  
Array declarations in IDL 3-13  
Atomic type 15-18  
Attribute declarator entry 14-16  
Attribute entry 14-17  
AttributeDef class 15-8  
Attributes  
    accessing from client programs 6-16

## Attributes (continued)

    get methods for 6-16  
    private attributes 3-40  
    readonly attributes 6-16  
    set methods for 6-16  
    syntax for declarations 3-19

## authentication

    DISABLE\_AUTHN keyword in [somsec]  
        stanza A-10

## B

Base class 11-1  
Base class entry 14-15  
Base proxy classes  
baseproxyclass modifier 3-27  
Before methods  
Binary compatibility of SOMobjects classes 1-7  
Binding data sets for client programs 6-3  
Binding files for SOM classes 3-2  
Binding files for SOMobjects classes 1-8, 1-9  
Bindings for SOMobjects classes 4-3  
    porting to another platform 4-7  
BOA (Basic Object Adapter) class 7-36  
Boolean IDL type 3-7  
Bounds exception 15-20  
BTREE\_StreamCreation keyword in  
    [SOMD\_POSSOM] stanza A-11  
Building Class Libraries 5-1

## C

C/C++ bindings for SOMobjects classes 4-4, 4-5  
    limitations of 4-6  
C/C++ Methods and Functions 11-25  
C/C++ usage bindings 6-3  
C/C++ binding files for SOM classes 3-2  
C/C++ binding files for SOMobjects classes 1-9  
C++ classes converted to SOMobjects classes 8-23  
    METHOD\_MACROS for 8-23  
Callback procedures/methods 16-3  
caller\_owns\_parameters modifier  
caller\_owns\_results modifier  
callstyle = oidl modifier 6-13, 6-14  
Casted method resolution 6-16  
change\_id method 12-45  
char IDL type 3-6  
Character output 11-25  
    from SOM methods/functions 11-25  
Child class 11-1  
Class categories  
    base class 11-1

**Class categories (continued)**

child class 11-1  
metaclass 2-2  
parent class 11-1  
parent class vs metaclass 11-1  
primitive class 2-2  
subclass 11-1

**Class data structure 2-10, 6-15****Class entry 14-14****Class libraries 17-1**

loading 6-22

**Class name, getting 11-26****Class names as types 3-13****Class objects 11-25, 11-26**

creating from a client program 6-20  
getting information about 11-25  
methods for 11-25  
getting the class of an object 6-20  
size of, getting 11-26  
using 6-20

**Class shadowing 14-4, 14-28, 14-30****Class variables 3-35****classinit modifier 3-28****classNameNew macro 6-7****classNameRenew macro 6-7****client authentication**

DISABLE\_AUTHN keyword in [somsec]  
stanza A-10

**Client events 16-2****client initialization in distributed SOMObjects 7-18****client programming in distributed SOMObjects 7-16**

client initialization 7-18  
client termination 7-32  
compiling and linking 7-32  
finding an object factory 7-20  
memory allocation and ownership 7-27  
default allocation responsibilities 7-27  
functions used for 7-29  
inout parameters 7-29  
out parameters and return results 7-27

method invocation 7-24, 7-25

object creation from a factory 7-22

object lifecycle service 7-18

object references 7-18

initial 7-18

ORB object creation 7-18

remote method calls 7-24

remote objects

creation of 7-19

destruction of 7-31

method calls on 7-24

somdCreate usage 7-24

**Client programming in DSOM 12-18, 12-19, 12-21,****12-24, 12-33, 12-66**

'stub' DLLs in 12-51

clients that are also servers 12-33

**Client programming in DSOM (continued)**

creating objects 12-21  
using metaclasses 12-21  
Direct-to-SOM (DTS) program 12-66  
finding existing objects 12-21  
memory allocation and ownership 12-24  
advanced options 12-24  
object references 12-19  
duplicating or testing 12-19  
saving/restoring 12-19  
remote objects 12-18  
implementation of, getting 12-18

**Client programs 3-2, 6-3**

compiling, prelinking, and linking 6-33  
creating objects in 6-7, 11-15  
header data set members 6-3  
header files 3-2  
initializer methods in 11-15  
method invocations 3-20

**Cobol examples**

language neutrality examples 9-22

**Collection classes 13-2, 13-3, 13-4, 13-5, 13-6, 13-7,****13-8, 13-9, 13-10**

abstract classes 13-3  
choosing the best class 13-7  
class inheritance versus element inheritance 13-2  
IsSame versus IsEqual comparisons 13-2  
iterator classes 13-8  
main collection classes 13-4, 13-5, 13-6, 13-7  
    somf\_TDeque class 13-6  
    somf\_TDictionary class 13-5  
    somf\_THashTable class 13-4  
    somf\_TPrimitiveLinkedList class 13-6  
    somf\_TPriorityQueue class 13-7  
    somf\_TSet class 13-5  
    somf\_TSofitedSequence class 13-6  
mixin classes 13-9  
naming conventions 13-3  
object\_initializer methods 13-2  
supporting classes 13-10

**Comment substitution in emitter template 14-11****Comments within IDL**

syntax of 3-38

**communications debugging**

SOMDCOMMDEBUGFLAG keyword in [somd]  
stanza A-6

**Compiling and linking**

distributed SOMObjects client programs 7-32  
DSOM servers 12-51

**Compiling, prelinking, and linking 6-33****const modifier 3-31****Constant declarations in IDL 3-18****Constant entry 14-17****ConstantDef class 15-8****Constructed IDL types**

enum 3-7

**Constructed IDL types (continued)**

- struct 3-7
- union 3-9

**Contained class** 15-7

**Container class** 15-8

**Context class** 7-37

**Context expression in method declarations** 3-23, 6-13, 6-15
 

- context parameter in method calls 6-13, 6-15

**copy method** 15-20

**CORBA compliance**

- distributed SOMobjects 7-35

**CORBA compliance of SOM system** 15-2

**CORBA compliance of SOMObjects system** 1-8
 

- SOMobjects IDL syntax 3-2

**create method** 7-39, 12-43

**create\_constant method** 12-43, 12-45

**create\_factory method** 12-48

**create\_list method**

**create\_operation\_list method**

**create\_request method**

**create\_request\_args method**

**create\_SOM\_ref method** 12-43, 12-44

**Creating objects in client programs** 6-7

**creating remote objects** 7-19

**customization and debugging of SOMobjects** 11-28

**Customization features of SOM**

- class objects initialized/uninitialized 11-24

**Customization features of SOMObjects**

- class loading and unloading 11-28
- error handling 11-33
- memory management 11-31
- objects initialized/uninitialized 11-8
- output processing 11-32

**customizing DLL loading**

- customizing loading/unloading 11-28
- DLL loading 11-28

**customizing DLL unloading**

- customizing loading/unloading 11-29
- DLL unloading 11-29

## D

**Data entry** 14-16

**deactivate\_impl method**

**Debugging**

- SOMDCOMMDEBUGFLAG keyword in [somd]
  - stanza A-6
- statements in stub procedures 8-20
- with SOMMTraced metaclass 17-10

**Deinitialization of objects** 11-24

**delete operator**

- use after 'new' operator, in C++ 11-16

**delete operator, use after ' new' operator, in C++** 6-10

**deleteImpldef method**

**Deque class (somf\_TDeque)** 13-6

**Derived metaclasses** 11-3

**destroying remote objects** 7-31

**Dictionary class (somf\_TDictionary)** 13-5

**Direct-call procedures** 11-7

**Direct-to-SOM (DTS)**

**Direct-to-SOM (DTS) program** 12-66

**directinitclasses modifier** 3-28, 11-9

**DISABLE\_AUTHN keyword in [somsec]**

- stanza A-10

**Dispatch methods** 6-19

**Dispatch-function method resolution** 6-19

**Dispatch-function method resolution** 11-7

**Dispatch-<#106>function method resolution** 11-30

**distributed applications, configuring**

- registering class interfaces 7-15
- updating implementation repository 12-22

**Distributed SOM**

- tutorial example 9-1

**distributed SOMObjects**

- Advanced topics 12-1
- benefits and features of 7-2
- classes
- client process flow steps 7-16
- client programming 7-16
- configuring applications 7-15
- constraints 7-34
- existing objects, finding 7-12
- existing SOM libraries, using 7-12
- exiting a client program 7-32
- factories 7-10
- factory proxies 7-12
- frameworks 7-1
- generic server role 7-33
- initialization of distributed SOMObjects client programs 7-18
- memory allocation and ownership 7-29
  - inout parameters 7-29
- Name service 7-11
- object references 7-18
- ORB object creation 7-18
- overview 7-3
- proxy objects 7-10
- runtime scenario 7-13
- who should read this chapter 7-1, 12-1

**Distributed SOMObjects (DSOM)** 1-10, 12-18, 12-24, 12-26, 12-27, 12-33, 12-48, 12-66
 

- base proxy classes, customizing 12-55
- clients that are also servers 12-33
- Direct-to-SOM (DTS) program 12-66
- Dynamic Invocation Interface 12-64
- factory creation, customized 12-48
- introduction to 1-10
- memory allocation and ownership 12-24, 12-26, 12-27
  - advanced options 12-24

**Distributed SOMObjects (DSOM) (continued)**

- memory allocation and ownership (continued)
  - of dual-owned parameters 12-26
  - of method parameters 12-24
  - of object-owned parameters 12-24
  - suppressing inout parameter freeing 12-27
- object references 12-19
  - duplicating or testing 12-19
- remote objects 12-18
  - implementation of, getting 12-18
- user-supplied proxies 12-52

**distributed SOMObjects (DSOM) advanced topics**

- Dynamic Invocation Interface 12-59
- EMan used with 12-56, 12-57
  - potential deadlocks of 12-57
- peer processes 12-56
- proxy classes (default base classes) 12-55

**distributed SOMObjects (DSOM) managing objects**

- in a server 12-42, 12-43, 12-44, 12-45
  - application-specific object refer 12-45
  - object references (SOMDOBJECTS) 12-42
  - ReferenceData 12-43
  - SOM object references 12-44

**distributed SOMObjects (DSOM) method dis-patching, customized** 12-49

**distributed SOMObjects (DSOM) object references** 12-19

- saving/restoring 12-19

**distributed SOMObjects (DSOM) server programming** 12-38

- example of 12-38

**distributed SOMObjects method arguments**

- strings
  - inout 7-34
- supported and unsupported types 7-35

**DLL**

- description 5-4
- understanding 5-4
- uses and benefits of 5-4
- using 5-4

**DLL Considerations** 7-15

**DLL loading** 6-22

**dllname modifier** 3-28, 6-23

**double IDL type** 3-6

**dual\_owned\_parameters modifier** 12-26

**dual\_owned\_result modifier** 12-26

**duplicate method** 7-39, 12-19

**Dynamic class loading** 6-22

**Dynamic dispatching** 6-19

**Dynamic Invocation Interface (DII)** 7-35, 7-40

**Dynamic methods** 11-7

**Dynamically linked library (DLL)**

- customizing loading 11-28

## E

**EMan event manager** 16-1

**Emitter class (SOMTEmitC)** 14-4, 14-5, 14-7

**Emitter Framework** 1-11

- description 14-1
- emitter class (SOMTEmitC) 14-4, 14-5, 14-7
- entry classes
  - class descriptions of 14-12
  - hierarchy of 14-12
  - introduction 14-3, 14-4
- entry objects 14-3
- error handling 14-31
- introduction to 1-11
- object graph builder 14-3
- structure of 14-1, 14-3
- table of section names/methods 14-38
- template class (SOMTTemplateOutputC) 14-4, 14-5, 14-10
- uses and benefits 14-2
- using 14-19
- writing an emitter
  - advanced topics 14-26
  - basics 14-19

**emitter framework utilities**

- NEWEMITEFW keyword in [somc] stanza A-4

**Emitter name** 14-7

**Emitter output**

- designing 14-23
- section names 14-23
- sections of 14-10, 14-23
- somtGenerateSections method 14-24

**emitter prefix**

- SMEMITPREFIX keyword in [somc] stanza A-5

**Emitter template** 14-10

- epilog sections 14-8, 14-23
- prolog sections 14-8, 14-23
- repeating sections of 14-8, 14-23
- standard sections of 14-7

**Emitters** 4-6

- for C bindings (c 4-4
  - h 4-4
- for C++ bindings (xc 4-5
  - XH 4-5
- imod emitter 4-6
- ir emitter 4-6, 15-12
- newemit facility 14-2, 14-6
- NEWEMIT utility 14-2, 14-19
- pdl emitter 4-5
- SMEMIT keyword in [somc] stanza A-5
- use by SOM Compiler 14-2, 14-5

**Entry classes**

- class descriptions of 14-12
- hierarchy of 14-12
- introduction 14-3, 14-4

**Entry objects** 14-3  
**Entry type** 14-13  
**Enum entry** 14-18  
**enum IDL type** 3-7  
**Enumerator name entry** 14-18  
**Environment structure** 6-12  
 in distributed SOMobjects 7-30  
**Environment variables** 4-12  
 -m options of SOM Compiler command, 4-12  
 SOMIR environment variable 15-12  
 SOMM\_TRACE environment variable 17-11  
**Epilog section of a template** 14-8, 14-23  
**equal method** 15-20  
**Error handling** 14-31  
 customizing 11-33  
**error log**  
 SOMErrorLogControl keyword in [somras]  
 stanza A-8  
 SOMErrorLogDisplayMsgs keyword in [somras]  
 stanza A-8  
 SOMErrorLogFile keyword in [somras] stanza A-8  
 SOMErrorLogSize keyword in [somras] stanza A-8  
**Event classes of Event Management**  
**Framework** 16-3  
**Event Management Framework** 16-1, 16-7  
 ConnectionNumber' macro 16-8  
 eventmsk.h' include file 16-3  
 advanced topics 16-8  
 basics of 16-1  
 callback procedures/methods 16-3  
 client events  
     generating 16-5  
 EMan parameters 16-3  
 event classes 16-3  
 event types  
     client events 16-2  
     sink events 16-2  
     timer events 16-2  
     work procedure events 16-2  
 extending EMan 16-8  
 interactive applications 16-7  
 message queues 16-2  
 MOTIF applications 16-8  
 processing events 16-6  
 RegData object 16-4  
 registering for events 16-4  
 SOMEEM class 16-2  
 SOMEEMRegisterData class 16-4  
 thread safety 16-8  
 tips on using EMan 16-9  
 unregistering for events 16-5  
**Examples**  
 C, C++, C++ Direct-to SOM (DTS), COBOL 10-1  
 expanded "Payroll" example 8-2  
 language neutrality 10-1  
 SOMobjects examples 8-1

**Exception entry** 14-18  
**exception IDL declarations** 3-14, 3-18  
 table of standard CORBA exceptions 3-17  
**exception\_free function** 7-30  
**ExceptionDef class** 15-8  
**exceptions, freeing**  
 in distributed SOMobjects 7-30  
**execute\_next\_request method** 12-41  
**execute\_request\_loop method**  
**externalize\_to\_stream method** 12-28

## F

**Factories**  
 finding a SOM object factory 7-20  
 somdCreate function 7-10, 7-24  
**factory modifier** 3-28  
**filestem modifier** 3-28  
**Filter methods** 14-9  
**find\_all\_impldefs method**  
**find\_any method** 7-11, 7-20  
**find\_Impldef method**  
**find\_Impldef\_by\_alias method**  
**find\_Impldef\_by\_class method**  
**find\_Impldef\_classes method**  
**float IDL type** 3-6  
**Floating point IDL types**  
 double 3-6  
 float 3-6  
**Foreign data types** 12-29  
 marshaling of 12-29  
**Forward references** 3-38  
 to non-IDL classes 3-38  
**Frameworks** 1-10, 1-11  
 as SOMobjects class libraries 1-10  
 Distributed SOMobjects (DSOM) 1-10  
 Emitter Framework 1-11  
 Event Management Framework 16-8  
 Interface Repository 2-14  
 Interface Repository Framework 1-10, 15-1  
 Metaclass Framework 1-11, 17-1  
 usage in client programs 2-17  
**free method** 15-20  
**free\_memory method**  
**functionprefix modifier** 3-29, 4-13  
**Functions for generating output** 11-25

## G

**Generating output** 11-25  
 customization of 11-32  
 from SOM methods/functions 11-25  
**generic server program (somdsvr)** 7-33  
**generic servers** 7-33  
**get method** 6-16

**get\_count method**  
**get\_id method** 12-45  
**get\_implementation method** 12-18  
**get\_item method**  
**get\_response method** 12-63  
**get\_SOM\_object method** 12-44  
**Global modifier** 4-12  
**Global modifiers** 14-7  
**global variables** 12-35  
    SOMDImplDefObject 12-35, 12-39  
    SOMDImplRepObject 12-22, 12-39  
    SOMDORBOObject 7-37  
    SOMDServerObject 12-40  
    SOMDSOMOAObject 12-39  
**Grammar of SOM IDL syntax** B-1

## H

**Hash table class (somf\_THashTable)** 13-4  
**Header data sets for SOMobjects classes** 3-5  
**Header files for distributed SOMobjects**  
    library files 7-32  
**Header files for SOM classes** 3-2  
**Header files for SOMobjects classes** 8-18  
**hops**  
    MAXIMUMHOPS keyword in [somd] stanza A-6

## I

**IBM COBOL for OS/390 and VM Version 2 Release 1**  
    compiling 2-18  
    Direct-to-SOM 5-1  
**Identifier names**  
    naming scope restrictions 3-41  
**IDL**  
    description 3-1  
    syntax 3-3  
    syntax (keywords, directives, and declarations) 3-5  
    uses and benefits 3-3  
    using IDL (with examples) 3-45  
**IDL syntax**  
    initializer methods 11-10  
**imod emitter** 4-6  
**impctx modifier** 3-31, 12-29  
**impl\_is\_ready method**  
**impldef\_prompts modifier** 3-31  
**implementation of objects** 7-40  
**Implementation Repository** 7-40  
    SOMIR keyword in [somir] stanza A-7  
**Implementation statement** 3-40, 8-10  
    syntax of 3-24  
**Implementation templates** 1-9, 3-2, 11-28  
    <className>MethodDebug procedure in 8-20  
    #define <className>\_Class\_Source  
        statement 8-18  
    #include header data set 3-5

## Implementation templates (*continued*)

#include header file 3-2, 8-18  
accessing internal instance variables 8-21  
bindings 1-9, 3-2, 4-3  
customizing the stub procedures 8-21  
GetData usage 8-19  
incremental updates of 4-3, 8-17, 8-23  
method procedures 8-18  
parent<#106>method calls in 8-22  
somSelf usage 8-19  
somThis usage 8-19  
syntax of SOMobjects Compiler output 8-17  
syntax of stub procedures for initializer  
    methods 11-13  
syntax of stub procedures for methods 8-18

**ImplementationDef class** 7-40, 12-18

**ImplementationDef object** 7-37

**implementing SOM classes**

    using SOM classes 7-33

**Implicit method parameter** 6-12

**ImplRepository class** 7-40

**in and out parameters** 3-21

**Incremental updates of implementation template** 4-3, 8-17, 8-23

**indirect modifier** 3-31

**Inheritance** 2-4, 11-1

**Inherited methods** 2-5

    overriding 2-5

**init modifier** 3-31

**initialization**

    of DSOM client programs 7-18

**Initializer methods**

    declaring new initializers 11-10

    implementing initializers in .idl file 11-13

    non-default initializer calls 11-14

    somDefaultInit method 11-8

    use in client programs 11-15

**Instance variable declarators** 3-35

    syntax of 3-35

**Instance variables**

    accessing in method procedures 8-21

**Integral IDL types** 3-6

    long 3-6

    short 3-6

    unsigned short or long 3-6

**Interface Definition Language** 1-7, 3-2

    introduction 3-1

    SOM classes defined in 3-2

**Interface names as types** 3-13

**Interface Repository** 1-10, 2-14, 15-1

    accessing objects in 15-13

    classes 15-7

    emitter 15-12

    files 15-4

    memory management in 15-10

    objects 15-7

**Interface Repository** (*continued*)

private information 15-6

**Interface Repository Framework** 1-10, 15-1

environment variables 15-5, 15-12

introduction to 1-10

**Interface Repository index** 15-24**Interface statement**

multiple interfaces defined 3-41

syntax of 3-17

**Interface vs implementation** 3-2**InterfaceDef class** 15-8**InterfaceDef object** 7-37**internalize\_from\_stream** method 12-28**invoke** method 12-63**Invoking methods** 6-12, 6-14, 6-15

from C++ client programs 6-14

from other client programs 6-15

initializer methods 11-15

**ir emitter** 4-6, 15-12**IR Framework**

description 15-1

understanding 15-4

uses and benefits 15-3

using 15-12

using the SOM Compiler to build an IR 15-12

**irindex command** 15-24**irindex examples** 15-26**irindex return codes** 15-25**is\_constant** method 12-45**is\_nil** method 7-39, 12-19**is\_proxy** method 12-19**is\_SOM\_ref** method 12-44, 12-45**IsSame versus IsEqual comparisons** 13-2**Iterator classes** 13-8**K****kind** method 15-20**L****Language bindings** 1-9, 3-2, 4-3**length** modifier 12-29**Libraries**

creating import library 6-33

**life cycle service** 7-40**Linked list class (somf\_TPrimitiveLinkedList)** 13-6**Linking** 6-33

distributed SOMobjects client programs 7-32

DSOM servers 12-51

**List substitution for template** 14-11**list\_initial\_services** method 7-18**Loading classes and DLLs** 11-28**location forwarding requests**

MAXIMUMHOPS keyword in [somd] stanza A-6

**long IDL type** 3-6**lookup\_id** method 15-14**M****Macros** 6-13, 6-17, 6-29, 6-30

\&gt;\_\&gt; 6-13

\&gt;\_lookup\_\&gt; 6-30

\&gt;New 6-13

lookup\_\&gt; 6-29, 6-30

SOM\_GetClass 6-20

SOM\_Resolve 6-17

SOM\_ResolveNoCheck 6-17

**maddstar compiler option** 6-7**Main collection classes** 13-4**Major and minor version numbers** 6-21**majorversion** modifier 3-29**marshal** method 12-29**Marshaling of foreign data types** 12-29**MAXIMUMHOPS keyword in [somd] stanza** A-6**maybe\_by\_value** modifier 12-28**maybe\_by\_value\_parameters** modifier 3-31**maybe\_by\_value\_result** 3-31**maybe\_by\_value\_result** modifier 12-28**memory allocation and ownership in distributed****SOMobjects** 7-27

default allocation responsibilities 7-27

functions used for 7-29

inout parameters 7-29

management by the client 7-29

out parameters and return results 7-27

**Memory allocation/ownership in DSOM** 12-24,**12-26**

advanced options 12-24

for method parameters 12-24

for object-owned parameters 12-26

**Memory management customization features** 11-31

SOMCalloc global variable 11-31

SOMFree global variable 11-31

SOMMalloc global variable 11-31

SOMRealloc global variable 11-31

**Message queues** 16-2**Metaclass entry** 14-15**Metaclass Framework** 1-11, 17-1

before/after behavior 17-3

introduction to 1-11

SOMM\_MVS\_Secure metaclass 17-12

SOMMBeforeAfter metaclass 17-3

SOMMSingleInstance metaclass 17-9

SOMMTraced metaclass 17-10

**metaclass** modifier 3-29**Metaclasses** 2-2

metaclass incompatibility 11-4

SOM&lt;#106&gt;derived 11-3

use in DSOM 12-21

**method declarations** 3-20  
**Method declarations in IDL**  
  context expression 3-23  
  in 3-21  
  out 3-21  
  initializer methods 11-10  
  oneway keyword 3-21  
  parameter list 3-21  
  raises expression 3-23  
  syntax of 3-20  
**Method entry** 14-17  
**Method invocations** 6-12, 6-13, 6-14, 6-15, 6-17, 6-19, 6-24  
  Context parameters 6-13, 6-15  
  dynamic dispatching 6-19  
  Environment variable 6-12  
  for client programs in C++ 6-14  
  for client programs in other languages 6-15  
  for initializer methods 11-15  
  format of 3-20, 6-12  
  implicit method parameters 6-12  
  method name/signature unknown at compile time 6-19  
  obtaining method procedure pointers 6-17  
  receiving object of 6-12  
  short form vs long form 6-13  
  va\_list methods 6-24  
**method modifier** 3-32, 11-7  
**Method procedure pointers** 6-17, 6-19  
  obtaining with name- lookup method 6-19  
  obtaining with offset method resolution 6-17  
**Method procedures** 8-18  
**Method resolution** 6-15, 6-17, 6-19, 6-29  
  by kinds of SOM methods 11-6  
  dispatch- function resolution 6-19  
  dispatch<#106>function resolution 11-30  
  introduction to 2-9  
  method procedure pointers 6-17  
  name- lookup resolution 6-19, 6-29  
  name-lookup resolution 11-7  
  name<#106>lookup resolution 11-30  
  offset resolution 2-10, 6-15, 6-17, 6-29  
**Method table** 2-10  
**Method tokens** 2-10, 6-15, 6-16  
**Method tracing**  
**METHOD\_MACROS for C++ bindings** 8-23  
**Methods** 6-12, 11-25, 11-26  
  class methods vs instance methods 2-2  
  customizing stub procedures in implementation templates 8-21  
  direct-call procedures 11-7  
  dynamic methods 11-7  
  for generating output 11-25  
  four kinds of SOM methods 11-6  
  getting the number of 11-26  
  initializer methods 11-8

**Methods (continued)**  
  invoking in client programs 6-12  
  modifiers 3-25, 3-40, 8-10  
  nonstatic methods 11-7  
  overriding 11-10, 11-24  
  static methods 11-7  
  stub procedures in implementation template 8-18  
  syntax of IDL method declarations 3-20  
**Methods and functions, language- neutral** 11-25  
**migrate modifier** 3-32  
**migration considerations** 1-11  
**minorversion modifier** 3-29  
**Mixin classes** 13-9  
**Modifier statements** 3-25, 3-36, 4-13, 12-24, 15-12  
  *See also* modifiers  
  attribute modifiers 3-32, 3-33  
    nodata 3-32  
    noget 3-32  
    noset 3-33  
  attributes modifiers 3-31  
    impldef\_prompts 3-31  
    indirect 3-31  
  class modifiers 3-25, 3-27, 3-28, 3-29, 3-30, 3-34  
    abstract 3-27  
    abstractparents 3-27  
    baseproxyclass 3-27  
    callstyle 3-28  
    classinit 3-28  
    directinitclasses 3-28  
    dllname 3-28  
    factory 3-28  
    filestem 3-28  
    functionprefix 3-29  
    majorversion 3-29  
    metaclass 3-29  
    minorversion 3-29  
    releaseorder 3-34  
    somallocate 3-29  
    somdeallocate 3-30  
  method modifiers 3-31, 3-32, 3-33, 3-36, 12-24  
    caller\_owns\_parameters 3-30  
    caller\_owns\_result 3-31  
    caller\_owns\_results 3-31  
    const 3-31  
    impldef\_prompts 3-31  
    init 3-31, 3-32  
    maybe\_by\_value\_parameters 3-31  
    maybe\_by\_value\_result 3-31  
    method 3-32  
    migrate 3-32  
    namelookup 3-32  
    nocall 3-32  
    noenv 3-32  
    nonstatic 3-32  
    nooverride 3-32  
    noself 3-33  
    object\_owns\_parameters 12-24

**Modifier statements (continued)**

- method modifiers (continued)
  - object\_owns\_result 12-24
  - offset 3-33
  - override 3-33
  - pass\_by\_copy 3-36
  - pass\_by\_copy\_parameters 3-33
  - pass\_by\_copy\_result 3-33
  - procedure 3-32
  - reintroduce 3-34
  - qualified 3-25, 3-30
  - SOM Compiler -m modifiers 4-13
  - syntax of 3-25
  - type modifiers 3-30, 3-31, 3-33, 3-34
    - impctx 3-31
    - length 3-31
    - pointer 3-33
    - staticdata 3-30
    - struct 3-34
    - unqualified 3-25, 3-27
- modifiers**
  - class modifiers 12-55
    - baseproxyclass 12-55
  - method modifiers 3-33
    - procedure 3-33
    - select 3-34
  - SOM IDL 14-4, 14-13
  - variables modifier
    - staticdata 3-34
- Module entry** 14-16
- Module statement**
  - syntax of 3-24, 3-41
- ModuleDef class** 15-8
- Modules**
  - handling 14-31
  - somtopenEmitFile function 14-31
  - target module 14-31
- Multiple inheritance** 2-5
- Multiple interfaces in a SOMObjects IDL data set**
  - syntax of 3-41
- MVSTraceLog keyword in [somras] stanza** A-9
- MVSTraceLogSize keyword in [somras] stanza** A-9

## N

**Name service**

- context of 7-11
- use in distributed SOMObjects 7-11

**Name- lookup method resolution** 6-19, 6-29

**Name-lookup method resolution** 11-7

**Name<#106>lookup method resolution** 3-41, 11-30

**NamedValue structure**

**Naming scopes** 3-41

**new operator in C++ client programs** 6-9

**new operator in C++ client programs** 6-10

**New\_ macro** 11-15

**newemit facility** 14-2, 14-6

**NEWEMIT utility** 14-2, 14-19

**NEWEMITEFW keyword in [somc] stanza** A-4

**nocall modifier** 3-32

**nodata modifier** 3-32

**noenv modifier** 3-32

**noget modifier** 3-32

**Nonstatic methods** 11-7

**nonstatic modifier** 3-32, 11-7

**nooverride modifier** 3-32

**noself modifier** 3-33

**noset modifier** 3-33

**Number of methods, getting** 11-26

**NVList class** 7-37

## O

**Object Adapter** 7-41

**Object graph builder** 14-3

**Object pseudo-class** 7-39

**Object references in distributed SOMobjects** 7-18, 7-38
 

- finding initial object references 7-18
- proxy classes
  - constructing 7-39
  - releasing 7-31

**Object references in DSOM** 12-19, 12-42
 

- creating in the SOMOA 12-42
- duplicating 12-19
- saving and restoring 12-19
- testing if NIL 12-19
- testing if proxy 12-19
- working with 12-19

**Object variables**

- declaring in client programs 6-6
- object type 6-6

**object\_owns\_parameters modifier** 12-24

**object\_owns\_result modifier** 12-24

**object\_to\_string method** 7-40, 12-19

**objects, customizing initialization/uninitialization**

- initializer methods 11-8
- initializing 11-8
- new initializers declared 11-10
- somDefaultInit method 11-8
- somInit method 11-8

**octet IDL type** 3-7

**Offset method resolution** 2-10, 6-15, 6-17, 6-29, 6-30, 11-7
 

- vs name- lookup method resolution 6-29

**offset modifier** 3-33

**oneway keyword of method declarations** 3-21

**Oneway messages in DSOM**

**OpenEdition Shell**

- SOMObjects Compiler examples 4-16

**OperationDef class** 15-8  
**ORB (Object Request Broker)**  
  CORBA compliance 7-35  
  ORB object creation 7-18  
**ORB class** 7-37, 7-39  
**ORBfree method** 7-30  
**OS/390 SOMobjects classes**  
  customizing loading/unloading 11-28  
  inheritance 11-1  
  parent class vs metaclass 11-1  
**out parameter** 3-21  
**Output file**  
  opening 14-31  
**Output processing**  
  customizing 11-32  
**Overloaded method** 2-5  
**override modifier** 3-33, 11-7  
**OVERRIDING OF METHODS** 2-5  
  somDefaultInit 11-10

## P

**param\_count method** 15-20  
**Parameter entry** 14-17  
**parameter memory management**  
  in distributed SOMobjects 7-27  
**parameter method** 15-20  
**ParameterDef class** 15-8  
**Parent class vs metaclass** 11-1  
**Parent class, getting** 11-26  
**pass\_by\_copy modifier** 3-36, 12-28  
**pass\_by\_copy\_parameters modifier** 3-33  
**pass\_by\_copy\_result modifier** 3-33, 12-28  
**Passing foreign data types** 12-29  
**Passing objects by copying** 12-28  
**Passing parameters by copying** 3-36  
**Passthru entry** 14-16  
**passthru statement** 3-37, 3-38  
  for non- IDL types or classes 3-38  
  syntax of 3-37  
  with multiple #includes 3-38

**PASSWD keyword in [somsec] stanza** A-9  
**pdl emitter** 4-5  
**pdl utility**  
  command syntax and options 4-19  
**PDS (persistent data store)**  
  POSIX\_StreamCreation keyword in  
  [SOMD\_POSSOM] stanza A-10  
**Peer processes in DSOM**  
**persistent data store (PDS)**  
  POSIX\_StreamCreation keyword in  
  [SOMD\_POSSOM] stanza A-10  
**persistent object manager (POM)**  
  POS\_POMDATA keyword in [SOMD\_POSSOM]  
  stanza A-10  
  POSIX\_StreamCreation keyword in  
  [SOMD\_POSSOM] stanza A-10

**Persistent object server in DSOM** 12-45  
**Persistent servers** 7-40  
**pointer modifier** 3-33  
**Pointer SOMobjects IDL declarations** 3-13  
**POM (persistent object manager)**  
  POS\_POMDATA keyword in [SOMD\_POSSOM]  
  stanza A-10  
  POSIX\_StreamCreation keyword in  
  [SOMD\_POSSOM] stanza A-10  
**Porting classes to another platform** 4-7  
**POS\_POMDATA keyword in [SOMD\_POSSOM]**  
  stanza A-10  
**POSIX\_StreamCreation keyword in**  
  [SOMD\_POSSOM] stanza A-10  
**primitive class** 2-2  
**Primitive Linked List class**  
  (somf\_TPrimitiveLinkedList) 13-6  
**Principal class** 7-37  
**print method** 15-20  
**printing of processing messages**  
  SMNOPRINTNAME keyword in [somc] stanza A-5  
**Printing output** 11-25  
  customization of 11-32  
  from SOM methods/functions 11-25  
**printing to standard output** 7-34  
**Priority Queue class (somf\_TPriorityQueue)** 13-7  
**Private methods and attributes**  
  syntax of 3-40  
**procedure modifier** 3-32, 3-33, 11-7  
**Prolog section of a template** 14-8, 14-23  
**Proxy classes**  
  customizing default base classes 12-55  
  user-supplied 12-52  
**proxy objects (in distributed SOMobjects)** 7-5,  
  7-10, 7-39  
**proxy objects in distributed SOMobjects** 7-38  
**Pseudo-objects** 15-19

## Q

**Qualified modifiers** 3-25, 3-30  
**Queue class (somf\_TPriorityQueue)** 13-7

## R

**raises expression in method declarations** 3-23  
**Receiving object** 6-12  
**ReferenceData type** 12-43  
  in DSOM 12-43  
**RegData objects** 16-4  
**reintroduce modifier** 11-7  
**release method** 7-10, 7-31, 7-39  
**releaseorder modifier** 3-34  
**remote method calls** 7-24  
**remote objects** 12-18  
  creation of

**remote objects** (*continued*)  
     destroying 7-31  
     implementation of, getting 12-18  
     method calls on 7-24  
**remove\_class\_from\_all method**  
**remove\_class\_fromImplDef method**  
**Repeating sections of a template** 14-8, 14-23  
**Repository class** 15-13  
**Repository ID** 15-13  
**Request class** 7-37  
**resolve method** 7-20  
**resolve\_initial\_references method** 7-11, 7-18  
**RESP\_NO\_WAIT flag** 12-63  
**Run-time environment** 1-10  
**Run<#106>time environment** 2-13, 2-17  
     initialization of 6-22

## S

**sc command**  
     -m option 14-7  
     -s option 14-7, 14-25  
**sc command to run SOMobjects Compiler**  
     compiler options 4-11  
**Scanning methods** 14-9, 14-25, 14-35  
**Scoping in IDL** 3-41  
**Section names** 14-23  
     changing 14-29  
     section<#106>name symbols 14-38  
     table of initial values and related methods 14-38  
**Section<#106>emitting methods** 14-9, 14-35, 14-38  
     customizing 14-29  
**Section<#106>name symbols** 14-38  
     values & methods 14-38  
     table of initial values & methods 14-38  
**Sections of a template** 14-10, 14-23  
**select modifier** 3-34  
**send method** 12-63  
**Sequence entry** 14-18  
**sequence IDL type** 3-10  
**Server activation (in DSOM)**  
**Server implementation definition (in DSOM)**  
     implementation definitions 12-35  
**Server objects (in DSOM)**  
**server programming in distributed**  
     SOMobjects 7-32  
**Server programming in DSOM** 12-34, 12-38, 12-42,  
     12-44, 12-45, 12-48, 12-49, 12-50  
     application<specific object refer 12-45  
     authentication 12-50  
     compiling and linking servers 12-51  
     example program 12-38  
     generic server program (somdsvr) 12-34, 12-45  
     object references 12-42  
     server implementation definition 12-35  
     server objects 12-37

**Server programming in DSOM** (*continued*)  
     servers 12-39, 12-40, 12-41, 12-42, 12-48, 12-49  
         customized factories 12-48  
         dispatching methods 12-49  
         initialization 12-39  
         managing objects 12-42  
         processing requests 12-40  
         termination 12-41  
     SOM object adapter (SOMOA class) 12-36, 12-39  
         initializing 12-39  
     SOM object references 12-44  
         subclassing SOMDServer 12-45  
**server-per-method servers** 7-40  
**servers** 12-34, 12-38, 12-42, 12-45  
     activation and deactivation 7-33, 12-37  
     activation policies  
         activation policies 7-40  
         compiling and linking 12-51  
         example server program 12-38  
         generic (somdsvr) 7-33, 7-40, 12-34  
         initializing the SOMOA 12-39  
         managing objects 12-42  
         persistent 7-40, 12-45  
         server-per-method 7-40  
         shared 7-40  
         SOMDServer server-object class 7-34  
         unshared 7-40  
**Set class (somf\_TSet)** 13-5  
**set method** 6-16  
**set\_item method**  
**setAlignment method** 15-20  
**Shadowing** 14-4, 14-28, 14-30  
**shared servers** 7-40  
**short IDL type** 3-6  
**Sink events** 16-2  
**size method** 15-20  
**Size of objects, getting** 11-26  
**SMADDSTAR keyword in [somc] stanza** A-5  
**SMEMIT keyword in [somc] stanza** A-5  
**SMEMITPREFIX keyword in [somc] stanza** A-5  
**SMINCLUDE keyword in [somc] stanza** A-4  
**SMNOPRINTNAME keyword in [somc] stanza** A-5  
**SMOE keyword in [somc] stanza** A-3  
**SOM bindings** 3-2  
     for SOM classes 3-2  
**SOM classes** 2-4, 2-5, 3-2  
     implementation 7-40  
     inheritance 2-4  
     interface vs implementation 3-2  
     multiple inheritance 2-5  
     subclassing 2-5  
     using with DSOM 7-33  
**SOM classes, implementing** 3-2  
     header files 3-2  
     implementation templates 3-2  
     interface definition file (.idl file) 3-2

**SOM classes, implementing** (*continued*)  
 Interface Definition Language (IDL) 3-2  
 interface vs implementation 3-2

**SOM classes, usage in client programs** 6-12, 6-24, 11-25  
 Environment structure 6-12  
 generating output, methods/functions for 11-25  
 getting information about 11-25  
   a class, methods for 11-25  
 method invocations 6-12  
 va\_list methods 6-24

**SOM Compiler**  
 and Interface Repository 15-12  
 structure of 2-14  
 use of emitters 14-2, 14-5  
 -m option of sc command 14-7  
 -s option of sc command 14-7, 14-25

**SOM IDL language grammar** B-1

**SOM IDL modifiers** 14-4, 14-13

**SOM IDL syntax** 3-35, 3-37  
 grammar of IDL B-1  
 instance variables 3-35  
 modifier statements 15-12  
 module statement definition 3-41  
 passthru statement 3-37  
 staticdata variables 3-35

**SOM object adapter** 7-39  
 SOMOA class 7-34, 12-36, 12-39

**SOM objects, customizing**  
**initialization/uninitialization**  
 'new' operator, in C++ 11-15  
 customizing class objects 11-24  
 example 11-17  
 non-default initializer calls 11-14  
 somDefaultInit method 11-24  
 somDestruct method 11-16, 11-24  
 somFree method 11-16  
 somInitMIClass method 11-24  
 somUninit method 11-13  
 uninitialized 11-16

**SOM system** 2-4, 3-2  
 bindings (language bindings) 3-2  
 inheritance 2-4

**SOM\_GetClass macro** 6-20

**SOM\_InterfaceRepository macro** 15-14

**SOM\_NoTest symbol** 6-18

**SOM\_NoTrace macro** 8-20

**SOM\_Resolve macro** 6-17

**SOM\_ResolveNoCheck macro** 6-17

**SOM\_SubstituteClass macro** 14-30

**SOM\_TestOn symbol** 6-18

**SOM/MVS classes**  
 implementing 3-5, 3-20, 3-25  
 header data sets 3-5  
 method invocations 3-20  
 modifiers 3-25

**SOM/MVS classes** (*continued*)  
 metaclasses 2-2

**som.h header data set members for C**  
 programs 6-3

**som.xh header data set members for C++**  
 programs 6-3

**SOM<#106>derived metaclasses** 11-3

**somAddDynamicMethod method** 11-7

**somallocate modifier** 3-29

**somApply function** 6-19

**somc stanza** A-3

**SOMCalloc function** 11-31

**SOMClass metaclass** 2-2

**somClassDispatch method** 6-19

**somClassFromId method** 6-24

**SOMClassMgr class** 2-3

**SOMClassMgrObject** 2-3, 6-22

**somClassResolve procedure** 6-16

**somd stanza** A-6

**SOMD\_ImplDefObject global variable** 12-35

**SOMD\_ImplRepObject global variable**

**SOMD\_Init function** 7-10, 7-18

**SOMD\_NO\_WAIT flag**

**SOMD\_NoORBfree function** 7-30

**SOMD\_ObjectMgr global variable** 7-18

**SOMD\_ORBObject global variable** 7-37

**SOMD\_POSSOM stanza** A-10

**SOMD\_RegisterCallback function**

**SOMD\_ServerObject global variable**

**SOMD\_SOMOAObject global variable**

**SOMD\_Uninit function** 7-10, 7-32

**SOMD\_WAIT flag**

**SOMDClientProxy class** 7-38

**SOMDCOMMDEBUGFLAG keyword in [somd]**  
 stanza A-6

**somdCreate function** 7-10, 7-24

**somdDispatchMethod** 12-49

**somdeallocate modifier** 3-30

**somDefaultInit method** 6-10, 11-12, 11-15, 11-24  
 use by 'new' operator 6-10

**somDestruct method** 6-9, 11-16, 11-24  
 use after SOMMalloc function 6-9

**somdExceptionFree function** 7-30

**SOMDForeignMarshaler class** 12-29

**somDispatch method** 6-19, 6-24  
 relating to va\_list 6-24

**SOMDOBJECT class** 7-37, 7-38, 7-39

**SOMDRECVWAIT keyword in [somd] stanza** A-6

**somdRefFromSOMObj method** 12-43, 12-45

**somdReleaseResources method** 12-26

**SOMDSENDWAIT keyword in [somd] stanza** A-6

**SOMDServer class** 7-34

**somdSOMObjFromRef method** 12-43, 12-45

**somdsvr program** 7-33

**somdsvr program (in DSOM)** 12-34, 12-45

**somdTargetFree** method 7-31  
**SOMDTIMEOUT** keyword in [somd] stanza A-7  
**SOMEEMan** class 16-2  
**SOMEEMRegisterData** class 16-4  
**SOMEEvent** class 16-3  
**somEnvironmentNew** function 6-22  
**SOMEError** global variable 11-33  
**SOMEErrorLogControl** keyword in [somras] stanza A-8  
**SOMEErrorLogDisplayMsgs** keyword in [somras] stanza A-8  
**SOMEErrorLogFile** keyword in [somras] stanza A-8  
**SOMEErrorLogSize** keyword in [somras] stanza A-8  
**somf\_TDeque** class 13-6  
**somf\_TDictionary** class 13-5  
**somf\_THashTable** class 13-4  
**somf\_TPrimitiveLinkedList** class 13-6  
**somf\_TPriorityQueue** class 13-7  
**somf\_TSet** class 13-5  
**somf\_TSortedSequence** class 13-6  
**SOMFACTORYNC** environment variable 7-20  
**somFindClass** method 6-10, 6-22  
**somFindCIsIn** File method 6-22, 6-23  
**somFindMethod** method 6-19, 6-30  
**somFindMethodOK** method 6-19, 6-30  
**SOMFOREIGN** data type 12-29  
**SOMFree** function 6-9, 11-31  
    use after SOMMalloc function 6-9  
    use with Renew macro 6-12  
**somFree** method 6-10, 7-31, 11-16  
    use after 'new' operator, in C++ 6-10  
    use after <className>New macro 6-11  
    use after classNameNew macro 6-7  
**somGetClass** method 6-20  
**somGetInstanceSize** method  
    use with <className>Renew macro 6-11  
**somGetInterfaceRepository** method 15-14  
**somGetMethodData** method 6-19  
**somInit** method 11-12  
    overriding 3-40, 8-10  
**somInitCtrl** data structure 11-9  
**somInitMIClass** method 11-24  
**SOMIR** environment variable 15-5, 15-12  
**SOMIR** keyword in [somir] stanza A-7  
**somir** stanza A-7  
**somLocateClassFile** method 6-23  
**somLookupMethod** method 6-19  
**SOMM\_MVS\_Secure** metaclass 17-12  
**SOMM\_TRACED** environment variable  
**sommAfterMethod** method  
**SOMMMalloc** function 11-31  
**SOMMBeforeAfter** metaclass 17-3  
**sommBeforeMethod** method  
**sommGetInstance** method  
**SOMMSingleInstance** metaclass 17-9

**SOMMTraced** metaclass 17-10  
**SOMMVS.SGOSIR Interface Repository** file 15-4  
**somNew** compared with somNewNoInit 7-22  
**somNew** method 6-13, 7-11, 7-22, 11-15  
    for creating instances 6-10, 6-11  
    invalid as first C method argument 6-13  
    use in C/C++ 6-10  
**somNewNoInit** method 7-22, 11-15, 11-16  
    use by new operator 6-10  
**SOMOA (SOM object adapter)** class 7-34, 7-39  
    use in method dispatching 7-41  
**SOMObject** class 2-2  
**SOMobjects** 1-7, 1-10  
    frameworks of, introduction to 1-10  
    introduction to 1-7  
**SOMobjects bindings** 1-8, 1-9  
    for C/C++ client programs 6-3  
    for SOMobjects classes 4-3  
**SOMobjects classes**  
    implementing 3-40, 4-7, 8-3, 8-10, 8-18  
        header files 8-18  
        modifiers 3-40, 8-10  
        porting classes to another platform 4-7  
        steps required 8-3  
        tutorial 8-3  
        usage in client programs 6-3, 6-4, 6-6, 6-7, 6-10,  
            6-20  
        C/C++ usage bindings 6-3  
        creating class objects 6-10, 6-20  
        creating instances 6-7, 6-9, 6-10  
        example program 6-4  
        freeing instances 6-7  
        getting the class of an object 6-20  
        object variables 6-6  
        SOMobjects header data set members for  
            C/C++ 6-3  
**SOMobjects Compiler** 1-9  
    actions of 8-16  
    bindings generated 4-3  
    C bindings 4-4  
    C++ bindings 4-5  
    compiler examples 4-15  
    compiler examples in the OpenEdition Shell 4-16  
    description 4-2  
    environment variables 4-10  
    implementation template created 8-16  
    incremental updates of implementation  
        template 4-3, 8-17, 8-23  
    introduction to 1-9  
    options 4-11  
    sc command, syntax and options 4-11  
    setting up to run from TSO, ISPF or batch 4-9  
    Tailoring the SOMobjects Profile 4-8  
    understanding 4-3, 14-3  
    uses and benefits of 4-2  
    using 4-9

**SOMobjects IDL syntax**

#ifdef \_\_SOMIDL\_\_ statement 3-40, 8-11  
#include directive 3-5  
attribute declarations 3-19  
comments 3-38  
constant declarations 3-18  
exception declarations 3-14, 3-18  
implementation statement 3-24, 3-40, 8-10  
interface declarations 3-17  
method declarations 3-20  
modifier statements 3-25  
module definition 3-24  
multiple interfaces 3-41  
name resolution 3-41  
naming scopes 3-41  
private methods and attributes 3-40  
procedure modifier 3-33  
scopes 3-41  
type declarations 3-18

**SOMobjects system 1-7, 1-8, 1-9, 1-10**  
binary compatibility of SOMobjects classes 1-7  
bindings (language bindings) 1-8, 1-9, 4-3  
class libraries from 1-7  
CORBA compliance 1-8, 3-2  
Interface Definition Language (IDL) 1-7  
language- neutral characteristics 1-8, 1-10  
language<#106>neutral characteristics 2-13, 2-17  
run<#106>time library 2-13, 2-17  
method resolution 2-9  
parent class vs metaclass 11-1  
run- time library of 1-10  
run<#106>time library of 2-13, 2-17  
SOMClass metaclass 2-2  
SOMClassMgr class 2-3  
SOMClassMgrObject 2-3  
SOMObject primitive class 2-2  
SOMobjects Compiler, introduction to 1-9

**somossvr program 7-33**  
**SOMOutCharRoutine global variable 11-25, 11-32**  
**somras stanza A-8**  
**SOMRealloc function 11-31**  
**somRenew method**  
for creating instances in given space 6-10  
**somResolve procedure 6-15**  
without C/C++ bindings 6-15  
**somResolveByName function 6-16, 6-19**  
**somsec stanza A-9**  
**somSelf pointer**  
syntax in implementation template 8-19

**SOMTAttributeEntryC class 14-17**  
**SOMTBaseClassEntryC class 14-15**  
**SOMTClassEntryC class 14-3, 14-14**  
**SOMTCommonEntryC class 14-14**  
**SOMTConstEntryC class 14-17**  
**somTD type definition 6-18**

**SOMTDataEntryC class 14-16**  
**SOMTEmitC class 14-4, 14-5, 14-7**  
**SOMTEntryC class 14-13**  
**SOMTEnumEntryC class 14-18**  
**SOMTEnumNameEntryC class 14-18**  
**somtGenerateSections method 14-24**  
**somThis assignment**  
syntax in implementation template 8-19  
**SOMTMetaClassEntryC class 14-15**  
**SOMTMethodEntryC class 14-3, 14-17**  
**SOMTModuleEntryC class 14-16**  
**somtopenEmitFile function 14-31**  
**SOMTParameterEntryC class 14-3, 14-17**  
**SOMTPassThruEntryC class 14-16**  
**SOMTSequenceEntryC class 14-18**  
**somtSetSymbolsOnEntry method 14-27, 14-35**  
**SOMTStringEntryC class 14-18**  
**SOMTStructEntryC class 14-18**  
**SOMTTemplateOutputC class 14-4, 14-5, 14-10**  
**SOMTTypedefEntry 14-16**  
**SOMTUUnionEntryC class 14-18**  
**SOMTUUserDefinedTypeEntryC class 14-18**  
**somUninit method**  
**Sorted sequence**  
**class(somf\_TSortedSequence) 13-6**  
**Standard sections of a template 14-7**  
**Standard symbols 14-7**  
by entry class availability 14-35  
by section validity 14-32  
section<#106>name symbols 14-38

**Static methods 6-19, 11-7**  
**staticdata modifier 3-30, 3-34**  
**staticdata variable declarators 3-35**  
**stream creation for BTREE protocol**  
BTREE\_StreamCreation keyword in  
[SOMD\_POSSOM] stanza A-11

**String entry 14-18**  
**string IDL type 3-10**  
**string\_to\_object method 7-40, 12-19**  
**Struct entry 14-18**  
**struct IDL type 3-7**  
**Struct member 14-16**  
**Struct member declarator entry 14-16**  
**struct modifier 3-34**  
**Stub procedures 8-20**  
for initializer methods 11-13

**Subclass 2-5, 11-1**  
**suppress\_inout\_free modifier 12-27**

**Symbol names**  
in emitter template 14-11  
section<#106>name symbols 14-38

**Symbol processing**  
comment substitution 14-11  
list substitution 14-11

**Symbols**  
defining new names 14-26

## **Symbols** (*continued*)

- getting values of 14-26
- in emitter template 14-10, 14-11

## **T**

**Tabbing in a template** 14-11

**Tailoring the SOMobjects Profile**

- as SOMobjects Compiler controls 4-8

**Target class entry** 14-14

**Target class of an emitter** 14-7

- standard symbols of 14-7, 14-32

**Target file of an emitter** 14-7

**Target module** 14-7, 14-31

**TCKind enumeration** 15-20

**TCP/IP communications packets**

- SOMDCOMMDEBUGFLAG keyword in [somd]
- stanza A-6

**Template**

- comment substitution in 14-11

- for emitter output 14-10

- list substitution in 14-11

- tabbing in 14-11

**Template class (SOMTTemplateOutputC)** 14-4, 14-5, 14-10

**Template file for an emitter** 14-10, 14-23

**Template object of an emitter** 14-7

**Template output** 14-10

- designing 14-23

- epilog sections 14-8, 14-23

- prolog sections 14-8, 14-23

- repeating sections of 14-8, 14-23

- section names 14-23

- sections of 14-10, 14-23

- somtGenerateSections method 14-24

- standard sections of 14-7

**Template sections** 14-10, 14-23

**Template symbols (symbol names)** 14-11

**Testing**

- with SOMMTraced metaclass 17-10

**timeout error**

- SOMDRECVWAIT keyword in [somd] stanza A-6

- SOMDSENDWAIT keyword in [somd] stanza A-6

**Timer events** 16-2

**tk\_<type> enumerator names** 15-20

**trace log**

- MVSTraceLog keyword in [somras] stanza A-9

- MVSTraceLogSize keyword in [somras] stanza A-9

**Tracing methods**

**Tutorial for implementing SOMobjects classes** 8-3

- #ifdef \_\_SOMIDL\_\_ statement 3-40, 8-11

- implementation statement 3-40, 8-10

- modifiers 3-40, 8-10

**Type declarations in IDL** 3-18

- any 3-7

- array 3-13

## **Type declarations in IDL** (*continued*)

boolean 3-7

char 3-6

constructed types 3-7

double 3-6

enum 3-7

exception 3-14

float 3-6

floating point types 3-6

integral types 3-6

long 3-6

object types 3-13

octet 3-7

pointer 3-13

sequence 3-10

short 3-6

SOM<#106>unique extensions 3-43

string 3-10

struct 3-7

template types 3-10

union 3-9

unsigned short or long 3-6

**TypeCode pseudo-objects** 15-18

alignment' modifier for 15-21

any' type usage 15-23

foreign data types for 15-22

methods for 15-20

TypeCode constants 15-23

**TypeDef class** 15-8

**Typedef entry** 14-16

**Types provided by SOM** 6-18

somMethodProc 6-18

somTD\_<>\_<> 6-18

## **U**

**Uninitialization of objects** 11-16, 11-24

**Union entry** 14-18

**union** IDL type 3-9

**Unloading classes and DLLs** 11-28

**Unqualified modifiers** 3-25, 3-27

**unshared servers** 7-40

**unsigned short or long** IDL type 3-6

**update\_impldef** method

**Updating the implementation template** 4-3, 8-17, 8-23

**Usage bindings** 1-8, 1-9, 3-2, 4-3, 6-3

**USER keyword in [somsec] stanza** A-9

**User<#106>defined type entry** 14-18

**Utility metaclasses** 17-1

## **V**

**va\_list** type 6-24

**Variable argument list** 6-24, 6-30

functions to create va\_lists 6-24

**Variable argument list (*continued*)**

methods using va\_lists 6-24  
using a va\_list in programs 6-30

**Variable argument list (va\_list)**

defining 3-22

**Version numbers 6-21, 11-26**

getting 11-26  
in customizing DLL loading 11-29

**W****wait time for a socket to become readable**

SOMDRECVWAIT keyword in [somd] stanza A-6

**wait time for a socket to become writable**

SOMDSENDWAIT keyword in [somd] stanza A-6

**well-known port 12-1****Work procedure events 16-2****Writing an emitter**

advanced topics 14-26  
basics 14-19



---

# Communicating Your Comments to IBM

OS/390  
SOMobjects  
Programmer's Guide  
Publication No. GC28-1859-03

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing an RCF from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+914+432-9405
- If you prefer to send comments electronically, use this network ID:
  - IBMLink: (United States customers only): KGNVMC(MHVRCFS)
  - IBM Mail Exchange: USIB6TC9 at IBMMAIL
  - Internet e-mail: mhvrcfs@us.ibm.com
  - World Wide Web: <http://www.s390.ibm.com/os390>

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

---

## Reader's Comments — We'd Like to Hear from You

**OS/390  
SOMobjects  
Programmer's Guide  
Publication No. GC28-1859-03**

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

**Note:** Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: \_\_\_\_\_

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

- |                                                        |                                                 |
|--------------------------------------------------------|-------------------------------------------------|
| <input type="checkbox"/> As an introduction            | <input type="checkbox"/> As a text (student)    |
| <input type="checkbox"/> As a reference manual         | <input type="checkbox"/> As a text (instructor) |
| <input type="checkbox"/> For another purpose (explain) |                                                 |
- 
- 

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: \_\_\_\_\_ Comment: \_\_\_\_\_

---

Name \_\_\_\_\_ Address \_\_\_\_\_

Company or Organization \_\_\_\_\_

Phone No. \_\_\_\_\_

**Reader's Comments — We'd Like to Hear from You**  
GC28-1859-03

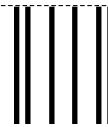


Cut or Fold  
Along Line

Fold and Tape

**Please do not staple**

Fold and Tape



---

NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES

---

A series of eight thick horizontal black bars, likely representing postage indicia.

## BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation  
Department 55JA, Mail Station P384  
522 South Road  
Poughkeepsie NY 12601-5400



Fold and Tape

**Please do not staple**

Fold and Tape

GC28-1859-03

Cut or Fold  
Along Line





Program Number: 5696-822



Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

GC28-1859-03

