SOMobjects Developer Toolkit

Technical Overview

A condensed Users Guide providing an introduction to the System Object Model and its accompanying frameworks

Version 2.0
November 1993

Second Edition (November 1993)

SOMobjects Developer Toolkit Technical Overview

Contents

Notices

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX
Operating System/2
OS/2
OS/2 Workplace Shell
RISC System 6000
SOMobjects
System Object Model

For convenience, the acronym “SOM” is used in this publication to reference the technology of the System Object Model, and the term “SOM Compiler” is used to reference the compiler of the System Object Model.

Each of the following terms used in this publication is a trademark of another company:

EXCEL	Microsoft Corporation
Intel	Intel Corporation
IPX	Novell Corporation
Lotus 1-2-3	Lotus Development Corporation
NetWare	Novell Corporation
Objective-C	The Stepstone Corporation
Smalltalk	Digitalk Inc.
Windows	Microsoft Corporation

The term “ANSI C” used throughout this publication refers to American National Standard X3.159–1989.

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

Chapter 1. Tutorial for Implementing SOM Classes

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

1.1 Basic Concepts of the System Object Model (SOM)

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

- An *object* is an OOP entity that has *behavior* (its *methods* or operations) and *state* (its data values). In SOM, an object is a run-time entity with a specific set of methods and instance variables. The methods are used by a client programmer to make the object exhibit behavior (that is, to do something), and the instance variables are used by the object to store its state. (The state of an object can change over time, which allows the object's behavior to change.) When a method is invoked on an object, the object is said to be the *receiver* or *target* of the method call.

- An object's *implementation* is determined by the procedures that execute its methods, and by the type and layout of its instance variables. The procedures and instance variables that implement an object are normally *encapsulated* (hidden from the caller), so a program can use the object's methods without knowing anything about how those methods are implemented. Instead, a user is given access to the object's methods through its *interface* (a description of the methods in terms of the data elements required as input and the type of value each method returns).

- An interface through which an object may be manipulated is represented by an *object type*. That is, by declaring a type for an object variable, a programmer specifies the interface that is intended to be used to access that object. *SOM IDL* (the **SOM Interface Definition Language**) is used to define object interfaces. The *interface names* used in these IDL definitions are also the type names used by programmers when typing SOM object variables.

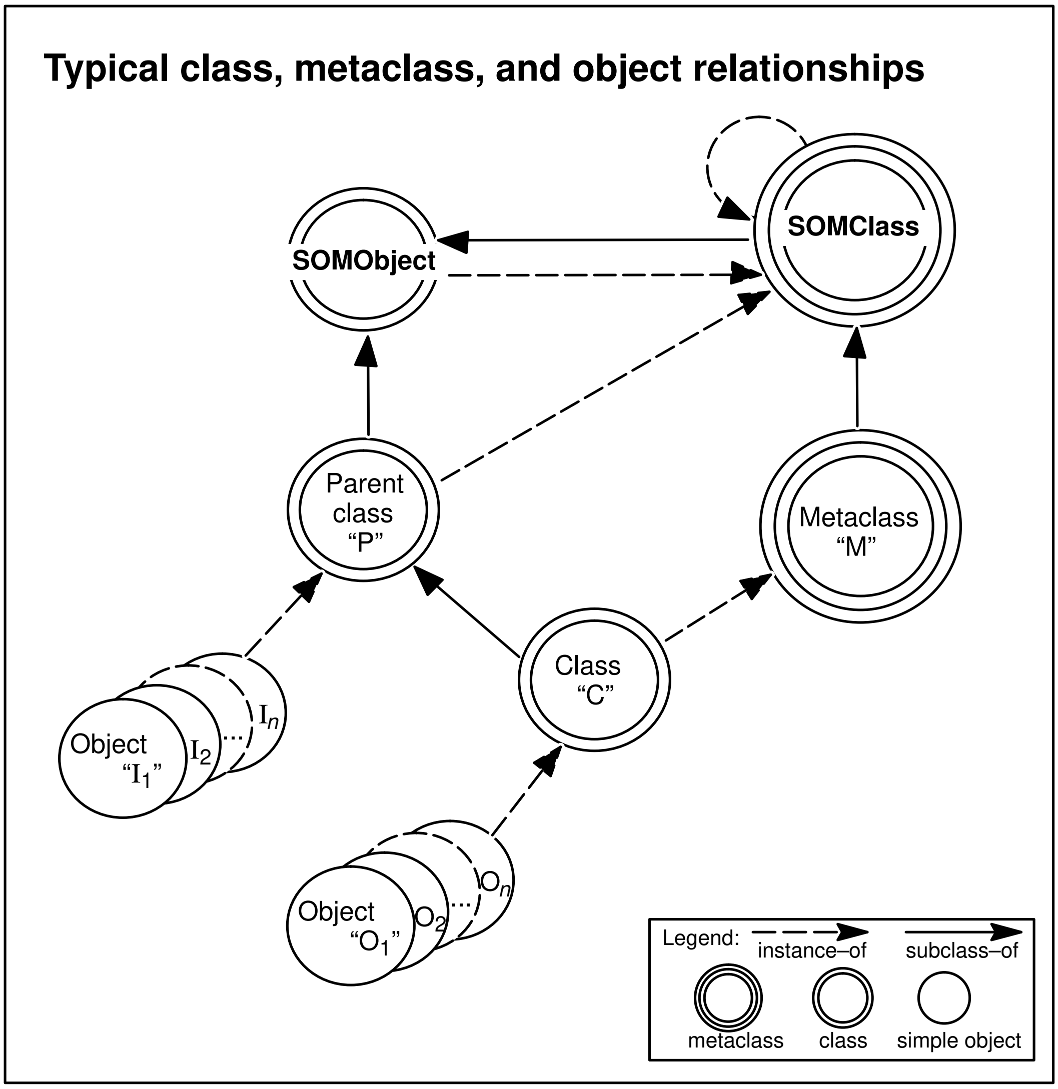
- In SOM, as in most approaches to object-oriented programming, a *class* defines the implementation of objects. That is, the implementation of any SOM object (as well as its interface) is defined by some specific SOM class. A class definition begins with an IDL specification of the interface to its objects, and the name of this interface is used as the class name as well. Each object of a given class may also be called an *instance* of the class, or an *instantiation* of the class.

- *Inheritance,* or *class derivation*, is a technique for developing new classes from existing classes. The original class is called the *base* class, or the *parent* class, or sometimes the direct *ancestor* class. The derived class is called a *child* class or a *subclass*. The primary advantage of inheritance is that a derived class inherits all of its parent's methods and instance variables. Also through inheritance, a new class can *override* (or redefine) methods of its parent, in order to provide enhanced functionality as needed. In addition, a derived class can introduce new methods of its own. If a class results from several generations of successive class derivation, that class “knows” all of its ancestors’s methods (whether overridden or not), and an object (or instance) of that class can execute any of those methods.

- SOM classes can also take advantage of *multiple inheritance,* which means that a new class is jointly derived from two or more parent classes. In this case, the derived class inherits methods from all of its parents (and all of its ancestors), giving it greatly expanded capabilities. In the event that different parents have methods of the same name that execute differently, SOM provides ways for avoiding conflicts.
- In the SOM run time, classes are themselves objects. That is, classes have their own methods and interfaces, and are themselves defined by other classes. For this reason, a class is often called a *class object*. Likewise, the terms *class methods* and *class variables* are used to distinguish between the methods/variables of a class object vs. those of its instances. (Note that the type of an object is *not* the same as the type of its class, which as a “class object” has its own type.)
- A class that defines the implementation of class objects is called a *metaclass*. Just as an instance of a class is an object, so an instance of a metaclass is a class object. Moreover, just as an ordinary class defines methods that its objects respond to, so a metaclass defines methods that a class object responds to. For example, such methods might involve operations that execute when a class (that is, a class object) is creating an instance of itself (an object). Just as classes are derived from parent classes, so metaclasses can be derived from parent metaclasses, in order to define new functionality for class objects.
- The SOM system contains three primitive classes that are the basis for all subsequent classes:

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

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



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

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

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

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

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

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

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

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

Development of the Tutorial examples

- **Example 1 — Implementing a simple class with one method**
  Prints a default message when the "sayHello" method is invoked on an object of the "Hello" class.
- **Example 2 — Adding an attribute to the Hello class**
  Defines a "msg" attribute for the "sayHello" method to use. The client program "sets" a message; then the "sayHello" method "gets" the message and prints it. (There is no defined message when an object of the "Hello" class is initialized.)
- **Example 3 — Overriding an inherited method**
  Overrides the required **somInit** method so the "msg" attribute will be "set" with default text as each instance of the "Hello" class is initialized. The client program can use that message and/or "set" it to a different one. As before, the "sayHello" method then "gets" the message and prints it.
- **Example 4 — Using metaclasses**
  Defines the metaclass "M_Hello" of the "Hello" class, plus a new metaclass method ("HelloCreate") that is invoked on a "Hello" class object to create an instance (or object) of the "Hello" class. The client program first initializes the "Hello" class object. Then, when the new "HelloCreate" method is invoked, the call includes an argument containing text that "sets" the "msg" attribute when an object of the "Hello" class is created. Thus, the "Hello" object is initialized with a "msg" of the client's choice.
- **Example 5 — Using metaclasses as counters**
  Does everything from Example 4, and extends it with an attribute ("numberObjs") for the "HelloCreate" method of the metaclass. The "numberObjs" attribute increments a counter each time a call to the "HelloCreate" method creates a new object of the "Hello" class. The client program also adds a print statement to report how many objects of the "Hello" class are created.
- **Example 6 — Using multiple inheritance**
  Does everything from Example 5, and extends it to provide the "Hello" class with multiple inheritance (from the "Disk" and "Printer" classes). The "Hello" interface defines an enum and an "output" attribute that takes its value from the enum (either "screen", "printer", or "disk"). The client program "sets" the form of "output" before invoking the "sayHello" method to send a "msg" (defined as in Examples 4 and 5).

1.2 Basic Steps for Implementing SOM Classes

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

1. Define the interface to objects of the new class (that is, the interface declaration), by creating a .idl file.
2. Run the SOM Compiler on the .idl file (by issuing the **sc** command) to produce binding files. One of these files is a template implementation file (a .c file for C programmers, or a .C (on AIX) or .cpp (on OS/2) file for C++ programmers). The SOM Compiler also produces a header file to be included in the implementation file (a .ih file for C programmers, or a .xih file for C++ programmers) and a header file to be included in client programs that use the class (a .h file for C clients, or a .xh file for C++ clients).

   To specify whether the SOM Compiler should produce C or C++ bindings, set the value of the SMEMIT environment variable or use the "–s" option of the **sc** command, as described in Section 3.4. The default is for the SOM Compiler to produce C bindings.
3. Customize the implementation, by adding code to the template implementation file.
4. Create a client program that uses the class.
5. Compile and link the client code with the class implementation, using a C or C++ compiler.
6. Execute the client program.

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

Example 1 — Implementing a Simple Class with One Method

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

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

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

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

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

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

   Note: For simplicity, this IDL fragment does not include a **releaseorder** modifier; consequently, the SOM Compiler will issue a warning for the method “sayHello”. For directions on using the **releaseorder** modifier to remove this warning, see the topic “Modifier statements” in Chapter 4, “Implementing Classes in SOM”)

2) Run the SOM Compiler to produce binding files and an implementation template (that is, issue the **sc** command):

```
> sc ”–sc;h;ih” hello.idl      (for C bindings)

> sc ”–sxc;xh;xih” hello.idl   (for C++ bindings)
```

   When set to generate C binding files, the SOM Compiler generates the following implementation template file, named “hello.c”, which contains *stub procedures* for each new method (procedures whose bodies are largely vacuous, to be filled in by the implementor). Unimportant details have been removed.

```
#include <hello.ih>

/*
 *  This method outputs the string ”Hello, World!”.
 */

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

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

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

The remaining lines of the template above are not pertinent at this point. (For those interested, they are discussed in section 4.7 of Chapter 4, "Implementing SOM Classes.") The file is now ready for customization with the C code needed to implement method "sayHello".

When set to generate C++ binding files, the SOM Compiler generates an implementation template file, "hello.C" (on AIX) or "hello.cpp (on OS/2), similar to the one above. (Chapter 4 discusses the implementation template in more detail.)

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

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

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

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

    printf("Hello, World!\n");
}
```

4) Create a client program that uses the class.

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

   A C programmer would write the following program in "main.c", using the bindings defined in the "hello.h" header file:

```
#include <hello.h>

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

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

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

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

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

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

_<*methodName*> **(**<*objectName*>, <*environment_argument*>, <*other_method_arguments*>**)**

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

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

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

```
#include <hello.xh>

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

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

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

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

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

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

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

Under AIX, for C programmers:

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

Under AIX, for C++ programmers:

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

Under OS/2, for C programmers:

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

Under OS/2, for C++ programmers:

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

6) Execute the client program.

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

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

File extensions for SOM files

- **IDL source file:**

.idl	for all users

- **Implementation template file:**

.c	for C, all systems
.C	for C++, on AIX
.cpp	for C++, on OS/2

- **Header file for implementation file:**

.ih	for C
.xih	for C++

- **Header file for program file:**

.h	for C
.xh	for C++

Example 2 — Adding an Attribute to the Hello class

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

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

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

```
attribute string msg;
```

   is equivalent to defining the two methods:

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

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

```
#include <somobj.idl>

interface Hello : SOMObject
{
  void  sayHello();

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

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

3) Customize the implementation.

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

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

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

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

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

4) Update the client program.

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

For C programmers:

```
#include <hello.h>

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

    obj = HelloNew();

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

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

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

For C++ programmers:

```
#include <hello.xh>

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

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

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

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

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

6) Execute the client program:

```
> hello
Hello World Again
```

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

Example 3 — Overriding an Inherited Method

As defined in Example 2, the "Hello" class has the limitation that it relies on the client program to initialize the "msg" attribute of an object before the "sayHello" method is invoked on that object. This example extends the class so that the "sayHello" method behaves reasonably even without client initialization of the "msg" attribute. Specifically, this example initializes the "msg" attribute as the object is created.

By default, the SOM method **somInit** is called once on every object as it is created, in order to initialize the object. Thus, to automatically initialize the "msg" attribute when an instance of the "Hello" class is created, the "Hello" class *overrides* (redefines) the SOM method **somInit**, which the "Hello" class inherits from its designated parent class, **SOMObject**.

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

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

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

```
#include <somobj.idl>

interface Hello : SOMObject
{
  void sayHello();

  attribute string msg;

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

};
```

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

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

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

```
#include <hello.ih>

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

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

SOM_Scope void   SOMLINK somInit(Hello somSelf)
{
    HelloData *somThis = HelloGetData(somSelf);
    HelloMethodDebug(”hello”,“somInit”);

    Hello_parent_SOMObject_somInit(somSelf);
}
```

Note that the SOM Compiler added code allowing the “Hello” class to redefine **somInit**. The SOM Compiler provides a default implementation for overriding the **somInit** method. This default implementation simply calls the “parent method” [the procedure that the parent class of “Hello” (**SOMObject**) uses to implement the **somInit** method]. This parent method call is accomplished by the invocation **`Hello_parent_SOMObject_somInit`**, which initializes any instance variables and attributes introduced by Hello’s parent class, **SOMObject**.

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

3) Customize the implementation.

Within the new **somInit** method procedure, after invocation of the parent method, code is added to initialize the “msg” attribute as each instance of the “Hello” class is created:

```
SOM_Scope void   SOMLINK somInit(Hello somSelf)
{
    HelloData *somThis = HelloGetData(somSelf);
    HelloMethodDebug(”Hello”,”somInit”);

    Hello_parent_SOMObject_somInit(somSelf);
    __set_msg(somSelf, somGetGlobalEnvironment(), ”Hello, World!”);
    /*  for C++, use:
     * somSelf->_set_msg(somGetGlobalEnvironment(),”Hello, World!”);
     */
}
```

4) Update the client program.

The client program can now be restored to the original version, which doesn't initialize the "msg" attribute. Then, the client program can be modified to change the "msg" that will print and to invoke the "sayHello" method a second time.

For C programmers:

```
#include <hello.h>

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

    obj = HelloNew();

    _sayHello(obj, ev);  /* Print the default message: */
    __set_msg(obj, ev, "Hello World Again"); /* Set new message: */

    _sayHello(obj, ev);

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

For C++ programmers:

```
#include <hello.xh>

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

    obj = new Hello;

    obj->sayHello(ev);
    obj->_set_msg(ev, "Hello World Again");
    obj->sayHello(ev);

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

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

6) Execute the client program, which now outputs both messages:

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

Example 4 — Using Metaclasses

The previous example showed how to override **somInit** to set a default value for the "msg" attribute. It would also be convenient if a user could initialize "msg" to a particular value as each "Hello" object is created. This is one use of *metaclasses*.

Recall that a metaclass is the class that a class object (such as "Hello") is an instance of. A metaclass defines the methods that a *class* uses to perform class-related operations. This includes such things as creating instances of the class, returning information about the class's methods or parent classes, and other "supervisory" operations that the *class itself* performs — as opposed to methods that *instances* of the class will perform (instance methods are defined within the class's .idl file or are inherited from its parent classes).

The default metaclass for every class is **SOMClass**. For example, **SOMClass** provides the **somNew** method that a class uses to create its instances. [The "HelloNew" procedure (used previously in the main program) invokes the **somNew** method on the "Hello" class object to create instances of the "Hello" class.] The current example, however, employs a different technique for creating objects — one that permits the user to specify an initial value for "msg".

A metaclass must be used here in order to modify a class-related operation (that is, to change the way in which the "Hello" class creates its instances, so that the client program can specify an initial value for the newly-created object's "msg" attribute). "Hello" obtains this class method from its metaclass "M_Hello", where the method is defined.

In summary, this example illustrates defining a new method of the "M_Hello" metaclass that, when performed by the "Hello" class object, creates an instance of the "Hello" class and sets the value of that instance's "msg" attribute to a specified text string. [The new method is declared in the "M_Hello" interface (step 1) and is implemented in the implementation file (step 2)]. This allows the value of the "msg" attribute to be initialized by the client as it creates each instance of the "Hello" class.

The client program then performs three main activities:

- Creates a "Hello" class object, which can perform the new class method defined by the metaclass.
- Creates instances of the "Hello" class, specifying the initial value for each instance's "msg" attribute (done by calling the class method defined by the metaclass).
- Prints the value of each instance's "msg" attribute (by invoking the "sayHello" method on each object).

The following example describes each step in more detail.

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

The interface for the metaclass of "Hello" must be declared, just as for the "Hello" class. By convention, the metaclass of "Hello" is called "M_Hello." The interface for "M_Hello" consists of a user-defined method, "HelloCreate", that creates an instance of the "Hello" class and initializes the "msg" attribute to the specified value.

```
#include <somobj.idl>  //# Get the parent (base) class definition
#include <somcls.idl>  //# Get the parent metaclass class definition

interface Hello;                // Forward declaration of Hello since
                                // HelloCreate returns class "Hello"
interface M_Hello : SOMClass    // The metaclass for Hello.
{
    Hello HelloCreate(in string message);
    //This method creates an instance of the Hello class and
    // uses the value of "message" to initialize it.
};
```

```
interface Hello : SOMObject
{
    void sayHello();

    attribute string msg;

#ifdef __SOMIDL__

     implementation {

        //# Class Modifiers
        metaclass = M_Hello; //# Identify Hello's metaclass
    };
#endif    //# __SOMIDL__
};
```

In the “implementation” statement for the “Hello” interface, we have removed the earlier statement `somInit:override`, since “msg” will be initialized differently. Now, the “implementation” statement for “Hello” designates the metaclass of “Hello” by using the “metaclass” modifier with a value of “M_Hello”. (If not explicitly specified, the metaclass of a class is derived from its parents.)

As shown in this example, the “interface” statements for a class and its metaclass may appear in the same .idl file. IDL also allows the interfaces to arbitrary sets of classes to be defined in the same .idl file; however, if the classes are not a class-metaclass pair, then the “interface” statements should be contained in a **module** statement. Chapter 4, “Implementing SOM Classes” contains a section entitled “SOM Interface Definition Language” that describes the use of modules in IDL.

2) Re-run the SOM Compiler on the updated .idl file, as before. The SOM Compiler extends the existing implementation file to include stub procedures for methods added for the class's metaclass, as specified by M_Hello's interface declaration. Below is the material that the SOM Compiler adds to the C language implementation file; C++ implementation files are similarly revised:

```
#include <hello.ih>

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

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

/*
 *  This method creates an instance of the Hello class and
 *  uses the value of "message" to initialize it.
 */
SOM_Scope Hello SOMLINK HelloCreate(M_Hello somSelf,
                                Environment *ev, string message)
{
    /* M_HelloData *somThis = M_HelloGetData(somSelf) ; */
    M_HelloMethodDebug("M_Hello","HelloCreate") ;

    /* Return statement to be customized: */
    return;
}
```

3) Customize the implementation.

Expand the “HelloCreate” method procedure as follows:

```
SOM_Scope Hello SOMLINK HelloCreate(M_Hello somSelf,
                                Environment *ev, string message)
{
    /* M_HelloData *somThis = M_HelloGetData(somSelf) ; */
    Hello obj = _somNew(somSelf) ;
        /* for C++, use:   somSelf->somNew()  */
    M_HelloMethodDebug("M_Hello","HelloCreate") ;

    __set_msg(obj, ev, message) ;
        /* for C++, use:  obj->_set_msg(ev, message);  */
    return (obj) ;
}
```

“HelloCreate” invokes the method **somNew** (provided by **SOMClass**) to create an instance of the “Hello” class. It then sets the value of the “msg” attribute by invoking the “_set_msg” method (similarly to Example 2 in the *client* program).

4) Update the client program.

The client program can now call the “HelloCreate” method to create an instance of the “Hello” class and set the “msg” attribute at the same time. This implementation is illustrated with the following C client program (a corresponding C++ example is also given following the explanatory paragraphs for step 4):

```
#include <hello.h>

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

    /*
    * Create the Hello Class and save a pointer to it.
    * The class of a Hello class object is M_Hello.
    */
    M_Hello helloClsObj = HelloNewClass(Hello_MajorVersion,
                                        Hello_MinorVersion) ;

    /* Now create three instances of the Hello class */
    a = _HelloCreate(helloClsObj, ev, "Hello from A") ;
    b = _HelloCreate(helloClsObj, ev, "Hello from B") ;
    c = _HelloCreate(helloClsObj, ev, "Hello from C") ;

    /* Now call sayHello on each object */
    _sayHello(a, ev) ;
    _sayHello(b, ev) ;
    _sayHello(c, ev) ;

    _somFree(a) ;
    _somFree(b) ;
    _somFree(c) ;
    return (0);
}
```

The client program first creates the “Hello” class object (`helloClsObj`), by calling the “HelloNewClass” procedure. The “HelloNewClass” procedure is defined by the SOM Compiler in the “hello.h” or “hello.xh” header file for the “Hello” class. (The SOM Compiler defines a <className>**NewClass** procedure for each class, in the header file it generates for the class.) In addition to creating the “Hello” class object, this procedure also creates the class objects for the parent classes and metaclass of “Hello”.

Previous examples did not explicitly call the “HelloNewClass” procedure, because it is called by the “HelloNew” macro. In this example, however, the client program must explicitly create the “Hello” class object, so that it can invoke the new “HelloCreate” method on it.

Note: The “Hello_MajorVersion” and “Hello_MinorVersion” arguments to “HelloNewClass” are defined in the header file for the “Hello” class by the SOM Compiler; they refer to the version numbers of the “Hello” class at the time the header file was produced. SOM ensures that the class object it creates is consistent with the specified version numbers.

After creating the “Hello” class object, the client program then creates three new instances of the “Hello” class and invokes the “sayHello” method on each one.

The following code illustrates the revised client program in C++.

```
#include <hello.xh>
#include <stdio.h>

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

    /*
     * Create the Hello class object, and save a pointer to it.
     * The class of a Hello class object is M_Hello.
     */
    M_Hello *helloClsObj = HelloNewClass(Hello_MajorVersion,
                                         Hello_MinorVersion);

    /* Create three different instances of the Hello class object. */
    a = helloClsObj->HelloCreate(ev, "Hello from A");
    b = helloClsObj->HelloCreate(ev, "Hello from B");
    c = helloClsObj->HelloCreate(ev, "Hello from C");

    /* Now call sayHello on each object. */
    a->sayHello(ev);
    b->sayHello(ev);
    c->sayHello(ev);

    a->somFree();
    b->somFree();
    c->somFree();
    return (0);
}
```

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

6) Execute the client program:

```
> hello
Hello from A
Hello from B
Hello from C
```

Example 5 — Using Metaclasses as Counters

The previous example introduced metaclasses as a way to provide *constructors* (methods for creating new instances of a class). Metaclasses are also useful for providing methods that maintain information about all the objects of a particular class. One simple use of metaclasses is to maintain a count of all of the objects that a class creates. This example does everything that Example 4 did, and also counts the number of objects created.

1) Modify the interface declaration in “hello.idl” so that it provides an attribute for the “Hello” class object to count the number of objects it creates.

```
#include <somobj.idl>
#include <somcls.idl>

interface Hello;               // Forward declaration of Hello since
                               // HelloCreate returns class "Hello"
interface M_Hello : SOMClass   // The meta class for Hello.
{
    Hello HelloCreate(in string message);
    //This method creates an instance of the Hello class and
    // uses the value of "message" to initialize it.

    readonly attribute long numberObjs;
};

interface Hello : SOMObject
{
    void sayHello();

    attribute string msg;

#ifdef __SOMIDL__

     implementation {

        //# Class Modifiers
        metaclass = M_Hello; //# Identify Hello's metaclass
    };
#endif    //# __SOMIDL__
};
```

The “readonly” keyword means that only a “_get_numberObjs” method will be defined, and not a “_set_numberObjs”. (A “_set_numberObjs” method is not needed because <u>clients</u> should not be permitted to change the value of the “numberObjs” attribute.) The implementation of the class can access the attribute by accessing its corresponding *instance variable*. Internal instance variables are automatically defined to store an attribute’s data. (An exception is attributes given the “nodata” SOM IDL attribute modifier, described in Chapter 4, “Implementing SOM Classes.)

2) Re-run the SOM Compiler on the updated .idl file, as before.
(In this case, the implementation file does not require modification.)

3) Customize the implementation.

Given the “readonly” attribute for “numberObjs”, modify the “HelloCreate” method to increment “numberObjs” every time a client invokes “HelloCreate”. Notice that the assignment statement for **somThis** below has been uncommented as well. This is necessary because the implementation of the “HelloCreate” method accesses the “numberObjs” attribute via the corresponding “numberObjs” instance variable. The assignment to **somThis** must be made before a method implementation can access instance variables of the class.

```
SOM_Scope Hello  SOMLINK HelloCreate(M_Hello somSelf,
                         Environment *ev, string message)
{
    M_HelloData *somThis = M_HelloGetData(somSelf) ;
    Hello obj = _somNew(somSelf) ;
        /* for C++, use:   somSelf->somNew()  */
    M_HelloMethodDebug("M_Hello","HelloCreate") ;

    __set_msg(obj, ev, message) ;
        /* for C++, use obj->_set_msg(ev, message) ;   */
    (somThis->numberObjs)++ ;
    return (obj) ;
}
```

The syntax `somThis->numberObjs` is used to set the value of the “numberObjs” attribute in the absence of a “set” method. In general, the implementation file for a class can access any instance variable of the class via the **somThis–>***variableName* macro, or by the shorthand form, *_variableName*. (In C++, the shorthand form is only available if VARIABLE_MACROS is defined (via a #define directive) in the implementation file prior to including the .xih file.)

4) Update the client program.

The client program can now report the number of “Hello” objects created during its lifetime, as in the following C program (the Example 4 code with a print statement added):

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

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

    /*
    * Create the Hello Class and save a pointer to it.
    * The class of a Hello class object is M_Hello.
    */
    M_Hello helloClsObj = HelloNewClass(Hello_MajorVersion,
                                        Hello_MinorVersion) ;

    /* Now create an instance of the Hello class */
    a = _HelloCreate(helloClsObj, ev, "Hello from A") ;
            /* for C++, use:
               a = helloClsObj->HelloCreate(ev, "Hello from A");  */
    b = _HelloCreate(helloClsObj, ev, "Hello from B") ;
    c = _HelloCreate(helloClsObj, ev, "Hello from C") ;

    /* Now call sayHello on each object */
    _sayHello(a, ev) ;   /* for C++, use:  a->sayHello(ev);  */
    _sayHello(b, ev) ;
    _sayHello(c, ev) ;

    printf("# objs created: %ld\n",
                             __get_numberObjs(helloClsObj, ev) );

          /*  for C++, use:  helloClsObj->_get_numberObjs(ev);   */
```

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

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

6) Execute the client program:

```
> hello
Hello from A
Hello from B
Hello from C
# objs created: 3
```

Attributes vs instance variables

As an alternative to defining “numberObjs” as an attribute, it could be defined as an instance variable, with a “get_numberObjs” method also defined for retrieving its value, as shown below:

```
interface M_Hello
//  The metaclass for Hello.
{
  Hello HelloCreate(in string message) ;
  // This method creates an instance of the Hello class and
  // uses the value of "message" to initialize it.

    long get_numberObjs() ;

#ifdef __SOMIDL__
implementation
{
    long numberObjs;
};
#endif
};
```

The disadvantage to using an instance variable is that the “get_numberObjs” method must be defined in the implementation file by the class implementor. For attributes, by contrast, default implementations of the “get” and “set” methods are generated automatically by the SOM Compiler in the .ih and .xih header files.

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

Regardless of whether you let the SOM Compiler generate your implementations or not, if access to instance data is required, either from a subclass or a client program, then this access should be facilitated by using an attribute. Otherwise, instance data can be defined in the “implementation” statement as above (using the same syntax with which variables are declared in C or C++), with appropriate methods defined to access it.

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

```
interface Date
{
    string get_date() ;

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

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

somThis–>variableName

or

_variableName

For example, the implementation for “get_date” would

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

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

Example 6 — Using Multiple Inheritance

The “Hello” class is useful for writing messages to the screen. So that clients can also write messages to printers and disk files, this example references two additional classes: “Printer” and “Disk”. The “Printer” class will manage messages to a printer, and the “Disk” class will manage messages sent to files. These classes can be defined as follows (this example also retains all of the functionality of Example 5):

```
#include <somobj.idl>

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

#include <somobj.idl>

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

};
```

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

1) Modify the interface declaration in “hello.idl”.

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

interface Hello;

interface M_Hello : SOMClass
// The metaclass for Hello.
{
    Hello HelloCreate(in string message);
    // This method creates an instance of the Hello class and
    // uses the value of ”message” to initialize it.

    readonly attribute long numberObjs;
};

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

    attribute string msg;

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

    attribute outputTypes output;
    // The current form of output
```

```
#ifdef __SOMIDL__

  implementation {

      metaclass = M_Hello;
      override: somInit; //# Override somInit to call the somInit
                         //# method of each parent as it is created.
  };
#endif //#  __SOMIDL__
};
```

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

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

Note: SOM IDL allows the use of structures, unions (though with a syntax different from C or C++), enumerations, constants, and typedefs, both inside and outside the body of an interface statement. Declarations that appear inside an interface body will be emitted in the header file (that is, in “hello.h” or “hello.xh”). Declarations that appear outside of an interface body do not appear in the header file (unless required by a special #pragma directive; see the SOM Compiler options in Chapter 4).

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

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

3) Customize the implementation, as in the following “hello.c” file:

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

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

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

Next, the implementation for “HelloCreate” is modified so that the “msg” output will go to the screen by default. Code in the client program can reassign output to the printer or to disk, as needed. In addition, we *must* override the **somInit** method for “Hello”, to ensure that all of the parent **somInit** methods are called.

```
SOM_Scope Hello  SOMLINK HelloCreate(M_Hello somSelf,
                                  Environment *ev, char *message)
{
    /* M_HelloData *somThis = M_HelloGetData(somSelf) ; */
    Hello obj = _somNew(somSelf) ;
        /* for C++, use:   somSelf->somNew()   */
    M_HelloMethodDebug("M_Hello","HelloCreate") ;

    __set_msg(obj, ev, message) ;
        /* for C++, use:  obj->_set_msg(ev, message) ;  */
    (somThis->numberObjs)++ ;
    __set_output(obj, ev, Hello_screen) ;
         /* for C++, use: obj->_set_output(ev, Hello_screen);  */
    return (obj) ;
}

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

    Hello_parent_Disk_somInit(somSelf);
    Hello_parent_Printer_somInit(somSelf);
}
```

4) Update the client program, as illustrated next:

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

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

    /*
    * Create the Hello Class and save a pointer to it.
    * The class of a Hello class object is M_Hello.
    */
    M_Hello helloClsObj = HelloNewClass(Hello_MajorVersion,
                                        Hello_MinorVersion) ;

    /* Now create an instance of the Hello class object*/
    a = _HelloCreate(helloClsObj, ev, "Hello from A") ;
             /* for C++, use:
                a = helloClsObj->HelloCreate(ev, "Hello from A"); */
```

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

    printf("# objs created: %ld\n",
                            __get_numberObjs(helloClsObj, ev) );

         /*  for C++, use:  helloClsObj->_get_numberObjs(ev) ; */

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

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

6) Execute the client program:

```
Hello from A
Hello from A              – goes to a Printer
Hello from A              – goes to Disk
# objs created: 1
```

This tutorial has described the features of SOM IDL that will be most useful to C and C++ programmers. SOM IDL also provides features such as full type checking, constructs for declaring private methods, and constructs for defining methods that receive and return pointers to structures. Chapter 4, “Implementing SOM Classes,” gives a complete description of the SOM IDL syntax. That chapter also describes how to use the SOM Compiler and provides more complete information on implementing SOM class libraries.

Chapter 2. Implementing Classes in SOM

This chapter begins with a more in-depth discussion of SOM concepts and the SOM run-time environment than was appropriate in Chapter 1, “Tutorial for Implementing SOM Classes.” Subsequent sections then describe the full syntax of the SOM Interface Definition Language (SOM IDL), followed by a description of the SOM Compiler command and options, plus additional information about completing an implementation template file, updating a template file, compiling and linking, packaging classes in libraries, and other useful topics for class implementors.

2.1 The SOM Run-Time Environment

As discussed in the Overview document, the SOMobjects Developer Toolkit provides

- The **SOM Compiler**, used when creating SOM class libraries, and
- The **SOM run-time library**, for using SOM classes at execution time.

The SOM run-time library provides a set of *functions* used primarily for creating objects and invoking methods on them. The *data structures* and *objects* that are created, maintained, and used by the functions in the SOM run-time library constitute the **SOM run-time environment**.

A distinguishing characteristic of the SOM run-time environment is that SOM *classes* are represented by run-time *objects*; these objects are called *class objects*. By contrast, other object-oriented languages such as C++ treat classes strictly as compile-time structures that have no properties at run time. In SOM, however, each class has a corresponding run-time object. This has three advantages: First, application programs can access information about a class at run time, including its relationships with other classes, the methods it supports, the size of its instances, and so on. Second, because much of the information about a class is established at run time rather than at compile time, application programs needn't be recompiled when this information changes. Finally, because class objects can be instances of user-defined classes in SOM, users can adapt the techniques for subclassing and inheritance in order to build object-oriented solutions to problems that are otherwise not easily addressed within an OOP context.

Run-time environment initialization

When the SOM run-time environment is initialized, four primitive SOM objects are automatically created. Three of these are class objects (**SOMObject**, **SOMClass**, and **SOMClassMgr**), and one is an instance of **SOMClassMgr**, called the **SOMClassMgrObject**. Once loaded, application programs can invoke methods on these class objects to perform tasks such as creating other objects, printing the contents of an object, freeing objects, and the like. These four primitive objects are discussed below.

In addition to creating the four primitive SOM objects, initialization of the SOM run-time environment also involves initializing global variables to hold data structures that maintain the state of the environment. Other functions in the SOM run-time library rely on these global variables.

For application programs written in C or C++ that use the language-specific bindings provided by SOM, the SOM run-time environment is automatically initialized the first time any object is created. Programmers using other languages must initialize the run-time environment explicitly by calling the **somEnvironmentNew** function (provided by the SOM run-time library) before using any other SOM functions or methods.

SOMObject class object

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

SOMClass class object

Because SOM classes are run-time objects, and since all run-time objects are instances of some class, it follows that a SOM class object must also be an instance of some class. The <u>class of a class is called a *metaclass*</u>. Hence, the instances of an ordinary class are individuals (nonclasses), while the instances of a metaclass are class objects.

In the same way that the class of an object defines the "instance methods" that the object can perform, the metaclass of a class defines the "class methods" that the class itself can perform. <u>*Class methods*</u> (sometimes called *factory methods* or *constructors*) are <u>performed by class objects</u>. Class methods perform tasks such as creating new instances of a class, maintaining a count of the number of instances of the class, and other operations of a "supervisory" nature. Also, class methods facilitate inheritance of instance methods from parent classes.

The distinction between instance methods and class methods, as well as that between objects, classes, and metaclasses, is illustrated in Figure 1. For information on the distinction between parent classes and metaclasses, see the section "Parent class vs. metaclass," below.

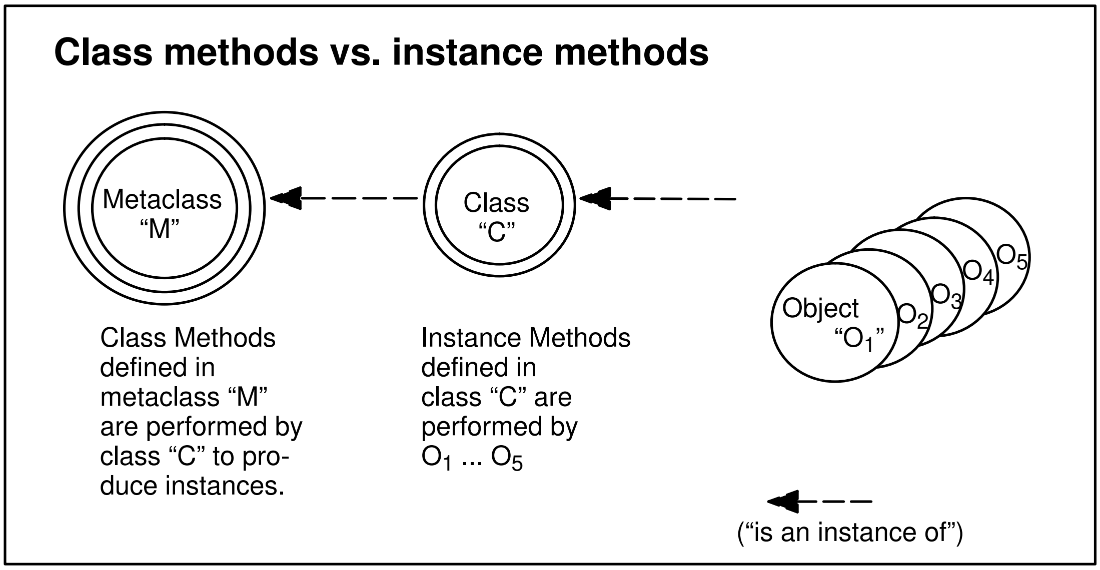


Figure 1. Class methods vs. instance methods.

<u>**SOMClass** is the root class for all SOM *metaclasses*</u>. That is, all SOM metaclasses must be subclasses of **SOMClass** or of some metaclass derived from **SOMClass**. **SOMClass** defines the essential <u>behavior common to all SOM *class objects*</u>. In particular, **SOMClass** provides:

- Two class methods, **somNew** and **somRenew**, for creating new class instances,
- Class methods that dynamically obtain or update information about a class and its methods at run time, as well as
- A method, **somInitMIClass,** for implementing multiple inheritance from parent classes, a method, **somOverrideSMethod,** for overriding inherited methods, and two methods, **somAddStaticMethod** and **somAddDynamicMethod**, for introducing new methods.

SOMClass is a subclass (or child) of **SOMObject**. Hence, SOM *class objects* can also perform the same set of basic *instance methods* common to all SOM objects. This is what allows SOM classes to be real objects in the SOM run-time environment. **SOMClass** also has the unique distinction of being its own metaclass (that is, **SOMClass** defines its own class methods).

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

SOMClassMgr class object and SOMClassMgrObject

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

- Maintain a registry (a run-time directory) of all SOM classes that exist within the current process, and to
- Assist in the dynamic loading and unloading of class libraries.

For C/C++ application programs using the SOM C/C++ language bindings, the **SOMClassMgrObject** automatically loads the appropriate library file and constructs a run-time object for the class the first time an instance of a class is created. For programmers using other languages, **SOMClassMgr** provides a method, **somFindClass**, for directing the **SOMClassMgrObject** to load the library file for a class and create its class object.

Relationships among the four primitive SOM run-time objects are illustrated in Figure 2. Again, the primitive classes supplied with SOM are **SOMObject, SOMClass,** and **SOMClassMgr.** During SOM initialization, the latter class generates an instance called **SOMClassMgrObject.** The left-hand side of Figure 2 shows parent-class relationships between the built-in SOM classes, and the right-hand side shows instance/class relationships. That is, on the left **SOMObject** is the parent class of **SOMClass** and **SOMClassMgr**. On the right **SOMClass** is the metaclass of itself, of **SOMObject,** and of **SOMClassMgr**, which are all class objects at run time. **SOMClassMgr** is the class of **SOMClassMgrObject**.

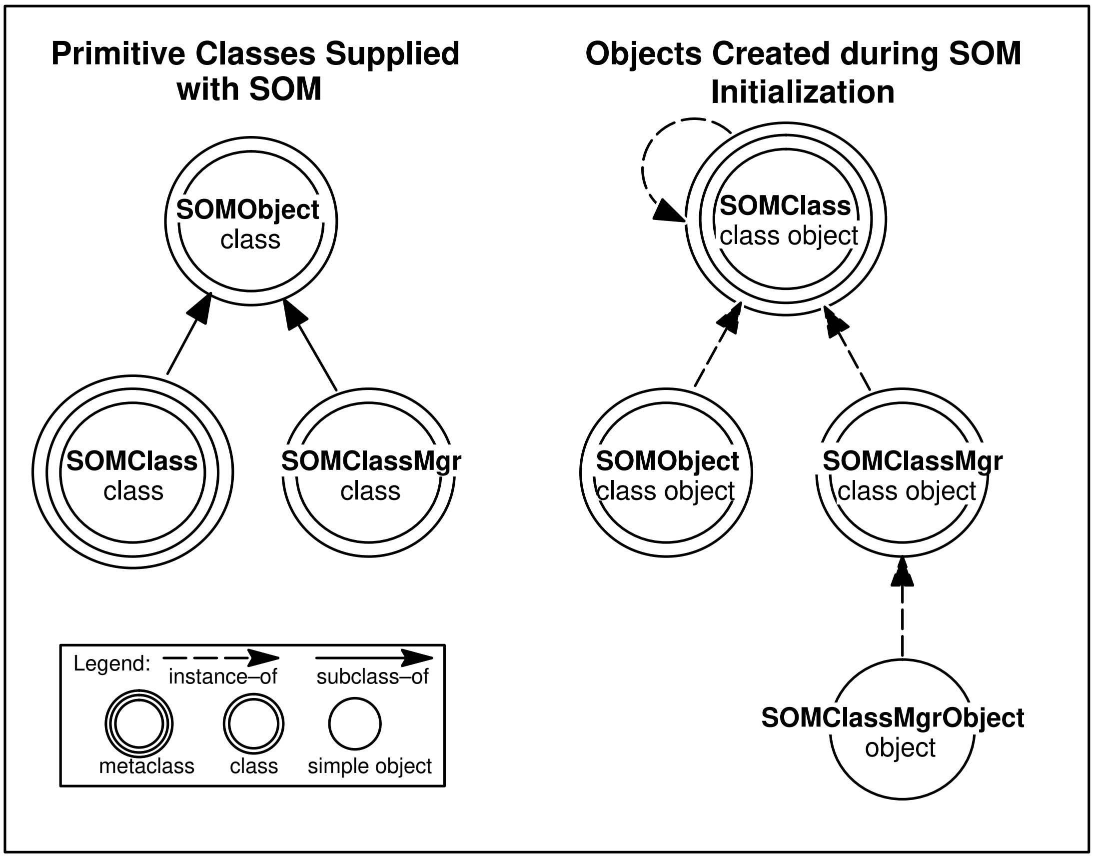


Figure 2. The SOM run-time environment provides four primitive objects, three of them class objects.

Parent class vs. metaclass

There is a distinct difference between the notions of "parent" (or base) class and "metaclass." Both notions are related to the fact that a class defines the methods and variables of its instances, which are therefore called *instance methods* and *instance variables*.

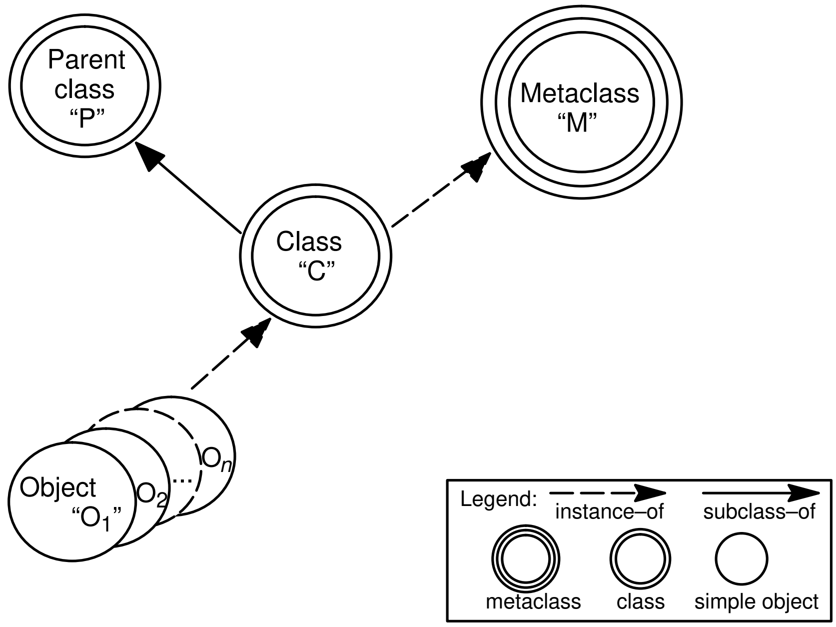
A *parent* of a given class is a class from which the given class is *derived* by *subclassing*. (Thus, the given class is called a *child* or a *subclass* of the parent.) A parent class is a class from which instance methods and instance variables are *inherited*. For example, the parent of class "Dog" might be class "Animal". Hence, the instance methods and variables introduced by "Animal" (such as methods for breathing and eating, or a variable for storing an animal's weight) would also apply to instances of "Dog", because "Dog" *inherits* these from "Animal", its parent class. As a result, any given dog instance would be able to breath and eat, and would have a weight.

A *metaclass* is a class whose instances are class objects, and whose instance methods and instance variables (as described above) are therefore the methods and variables of class objects. For this reason, a *metaclass* is said to define *class methods* — the methods that a class object performs. For example, the metaclass of "Animal" might be "AnimalMClass", which defines the methods that can be invoked on class "Animal" (such as, to create Animal instances — objects that are not classes, like an individual pig or cat or elephant or dog).

Note: It is important to distinguish the methods of a class object (that is, the methods that can be invoked on the class object, which are defined by its metaclass) from the methods that the class defines for its instances.

To summarize: the *parent* of a class provides inherited methods that the class's *instances* can perform; the *metaclass* of a class provides class methods that the *class itself* can perform. The distinctions between parent class and metaclass are summarized in Figure 3.

Characteristics of parent class vs. metaclass



Any class "C" has both a *metaclass* and one or more *parent* class(es).

- The *parent* class(es) of "C" provide the inherited *instance methods* that individual instances (objects "O_i") of class "C" can perform. Instance methods that an instance "O_i" performs might include (a) initializing itself, (b) performing computations using its instance variables, (c) printing its instance variables, or (d) returning its size. New instance methods are defined by "C" itself, in addition to those inherited from C's parent classes.
- The *metaclass* "M" defines the *class methods* that class "C" can perform. For example, class methods defined by metaclass "M" include those that allow "C" to (a) inherit its parents's instance methods and instance variables, (b) tell its own name, (c) create new instances, and (d) tell how many instance methods it supports. These methods are inherited from SOMClass. Additional methods supported by "M" might allow "C" to count how many instances it creates.
- Each class "C" has one or more parent classes and exactly one metaclass. (The single exception is SOMObject, which has no parent class.) Parent class(es) must be explicitly identified in the IDL declaration of a class. (SOMObject is given as a parent if no subsequently-derived class applies.) If a metaclass is not explicitly listed, the SOM run time will determine an applicable metaclass.
- An instance of a metaclass is always another *class object*. For example, class "C" is an instance of metaclass "M". SOMClass is the SOM-provided metaclass from which all subsequent metaclasses are derived.

Figure 3. A class has both parent classes and a metaclass.

A metaclass has its own inheritance hierarchy (through its parent classes) that is independent of its instances' inheritance hierarchies. In Figure 4, a sequence of classes is defined (or derived), stemming from **SOMObject**. The child class (or subclass) at the end of this line ("C_2") inherits instance methods from all of its ancestor classes (here, **SOMObject** and "C_1"). An instance created by "C2" can perform any of these instance methods. In an analogous manner, a line of metaclasses is defined, stemming from **SOMClass.** Just as a new class is derived from an existing class (such as **SOMObject**), a new metaclass is derived from an existing metaclass (such as **SOMClass**). In this example, both **SOMObject** and class "C1" are instances of the **SOMClass** metaclass, whereas class "C2" is an instance of metaclass "M2", which inherits from **SOMClass**.

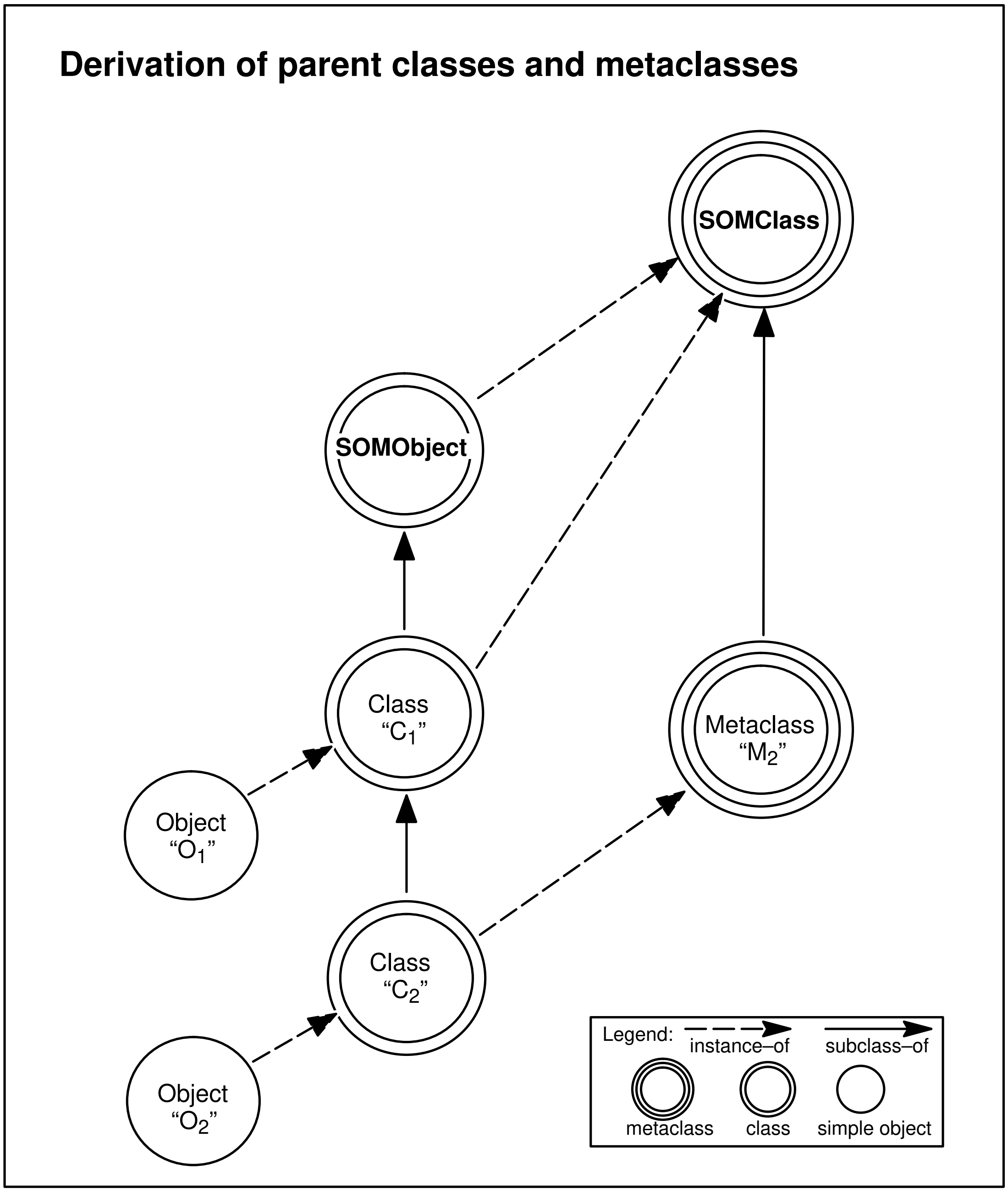


Figure 4. Parent classes and metaclasses each have their own independent inheritance hierarchies.

SOM-derived metaclasses

As previously discussed, a class object can perform any of the class methods that its metaclass defines. New metaclasses are typically created to modify existing class methods or introduce new class method(s). Example 4 of the Tutorial in Chapter 1 illustrates this.

Three factors are essential for effective use of metaclasses in SOM:

- First, every class in SOM is an object that is implemented by a metaclass.
- Second, programmers can define and name new metaclasses, and can use these metaclasses when defining new SOM classes.
- Finally, and most importantly, metaclasses cannot interfere with the fundamental guarantee required of every OOP system: specifically, any code that executes without method-resolution error on instances of a given class will also execute without method-resolution errors on instances of any subclass of this class.

Surprisingly, SOM is currently the only OOP system that can make this final guarantee while also allowing programmers to explicitly define and use named metaclasses. This is possible because SOM automatically determines an appropriate metaclass that supports this guarantee, automatically deriving new metaclasses by subclassing at run time when this is necessary.

To better understand this concept, consider the situation in Figure 5. Here, class “A” is an instance of metaclass “AMeta”. Assume that “AMeta” supports a method “bar” and that “A” supports a method “foo” that uses the expression “_bar(_somGetClass(somSelf)).” That is, method “foo” invokes “bar” on the class of the object on which “foo” is invoked. For example, when method “foo” is invoked on an instance of class “A” (say, object “O_1”), this in turn invokes “bar” on class “A” itself.

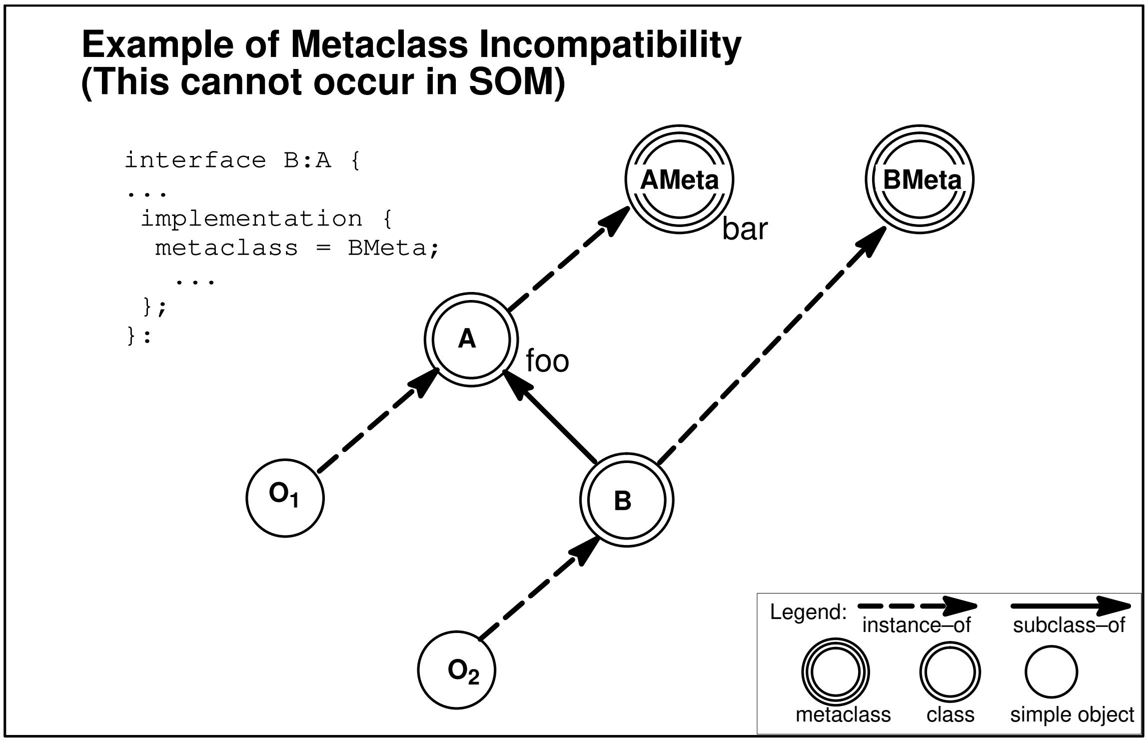


Figure 5. Example of Metaclass Incompatibility.

Now consider what happens when “A” is subclassed by “B”, a class that has the explicit metaclass “BMeta” declared in its SOM IDL source file, as shown by the code in Figure 5. If the class hierarchy were formed as in Figure 5, then an invocation of “foo” on “O_2” would fail, because metaclass “BMeta” does not support the “bar” method introduced by “AMeta”.

There is only one way that “BMeta” can support this specific method — by inheriting it from “AMeta” (“BMeta” could introduce another method named “bar”, but this would be a *different* method from the one introduced by “AMeta”). Therefore, in this example, because “BMeta” is not a subclass of “AMeta”, “BMeta” cannot be allowed to be the metaclass of “B”. That is, “BMeta” is not compatible with the requirements placed on “B” by the fundamental principle of OOP referred to above. This situation is referred to as *metaclass incompatibility*.

SOM does not allow hierarchies with metaclass incompatibilities. Instead, SOM automatically builds *derived metaclasses* when this is necessary. The resulting class hierarchy in this example is depicted in Figure 6, where SOM has automatically built the metaclass “DerivedMetaclass”. This ensures that the invocation of method “foo” on instances of class “B” will not fail, and also ensures that the desired class methods provided by “BMeta” will be available on class “B”.

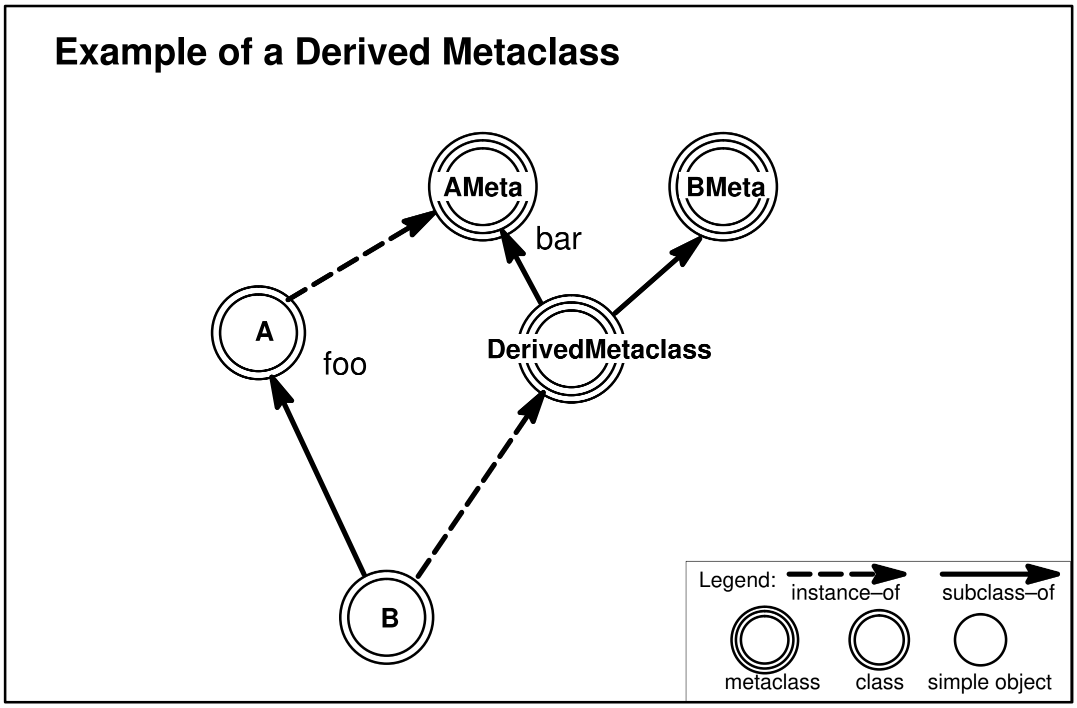


Figure 6. Example of a Derived Metaclass.

There are three important aspects of SOM’s approach to derived metaclasses:

- First, the creation of SOM-derived metaclasses is integrated with programmer–specified metaclasses. If a programmer–specified metaclass already supports all the class methods and variables needed by a new class, then the programmer–specified metaclass will be used as is.
- Second, if SOM must derive a different metaclass than the one explicitly indicated by the programmer (in order to support all the necessary class methods and variables), then the SOM-derived metaclass inherits from the explicitly indicated metaclass first. As a result, the method procedures defined by the specified metaclass take precedence over other possibilities (see the following section on inheritance and the discussion of resolution of ambiguity in the case of multiple inheritance).
- Finally, the class methods defined by the derived metaclass invoke the appropriate initialization methods of its parents to ensure that the class variables of its instances are correctly initialized.

As further explanation for the automatic derivation of metaclasses, consider the following multiple-inheritance example. In Figure 7, class “C” does not have an explicit metaclass declaration in its SOM IDL, yet its parents do. As a result, class “C” requires a derived metaclass. (If you still have trouble following the reasoning behind derived metaclasses, ask yourself the following question: What class should “C” be an instance of? After a bit of reflection, you will conclude that, if SOM did not build the derived metaclass, you would have to do so yourself.)

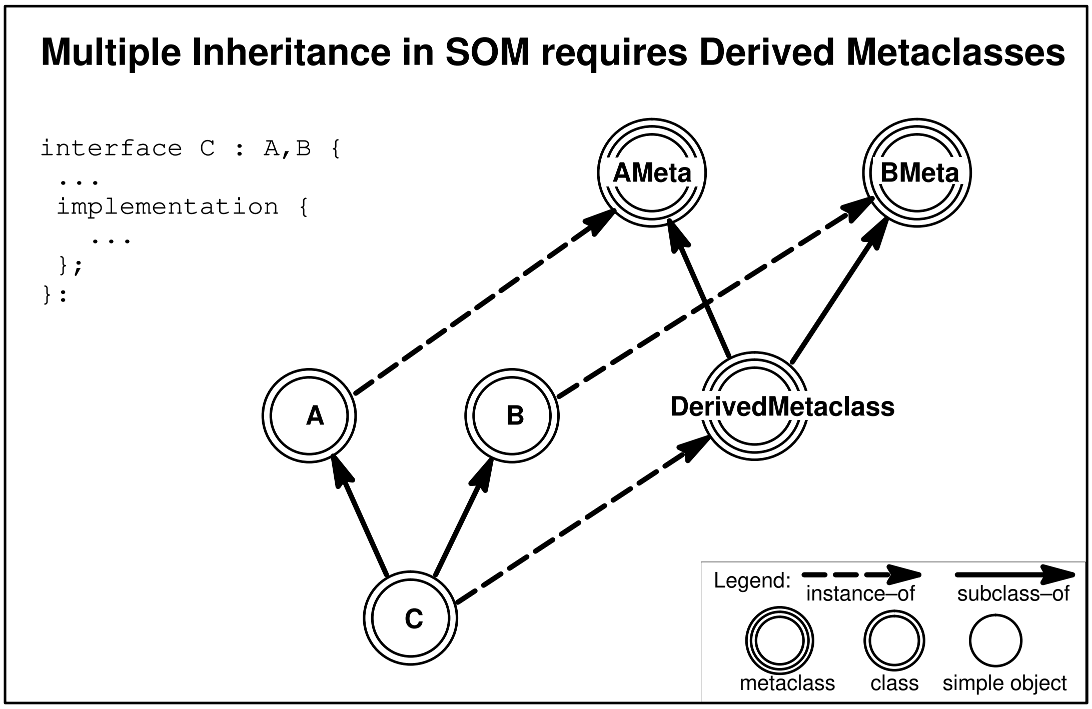


Figure 7. Multiple Inheritance requires Derived Metaclasses.

In summary, SOM allows and encourages the definition and explicit use of named metaclasses. With named metaclasses, programmers can not only affect the behavior of class instances by choosing the parents of classes, but they can also affect the behavior of the classes themselves by choosing their metaclasses. Because the behavior of classes in SOM includes the implementation of inheritance itself, metaclasses in SOM provide an extremely flexible and powerful capability allowing classes to package solutions to problems that are otherwise very difficult to address within an OOP context.

At the same time, SOM is unique in that it relieves programmers of the responsibility for avoiding metaclass incompatibility when defining a new class. At first glance, this might seem to be merely a useful (though very important) convenience. But, in fact, it is absolutely essential, because SOM is predicated on binary compatibility with respect to changes in class implementations.

A programmer might, at one point in time, know the metaclasses of all ancestor classes of a new subclass, and, as a result, be able to explicitly derive an appropriate metaclass for the new class. Nevertheless, SOM must guarantee that this new class will still execute and perform correctly when any of its ancestor class’s implementations are changed (which could even include specifying different metaclasses). Derived metaclasses allow SOM to make this guarantee. A SOM programmer need never worry about the problem of metaclass incompatibility; SOM does this for the programmer. Instead, explicit metaclasses can simply be used to “add in” whatever behavior is desired for a new class. SOM automatically handles anything else that is needed. Chapter 10 provides useful examples of such metaclasses. A SOM programmer should find numerous applications for the techniques that are illustrated there.

2.2 Inheritance

One of the defining aspects of an object model is its characterization of inheritance. This section describes SOM's model for inheritance.

A class in SOM defines an implementation for objects that support a specific interface:

- The *interface* defines the methods supported by objects of the class, and is specified using SOM IDL.
- The *implementation* defines what instance variables implement an object's state and what procedures implement its methods.

New classes are derived (by subclassing) from previously existing classes through inheritance, specialization, and addition. Subclasses 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). Subclasses also inherit implementation (the procedures that implement the methods) from their parent classes *unless* the methods are *overridden* (redefined or specialized). In addition, a subclass may introduce new instance methods and instance variables that will be inherited by other classes derived from it.

SOM also supports *multiple inheritance*. That is, a class may be derived from (and may inherit interface and implementation from) multiple parent classes. Note: Multiple inheritance is available only to SOM classes whose interfaces are specified in IDL, and not to SOM classes whose interfaces are specified in SOM's earlier interface definition language, OIDL. See Appendix B for information on how to automatically convert existing OIDL files to IDL.

It is possible under multiple inheritance to encounter potential conflicts or ambiguities with respect to inheritance. All multiple inheritance models must face these issues, and resolve the ambiguities in some way. For example, when multiple inheritance is allowed, it is possible that a class will inherit the same method or instance variable from different parents (because each of these parents has some common ancestor). In this situation, a SOM subclass inherits only one implementation of the method or instance variable. (The implementation of an instance variable within an object is just the location where it is stored. The implementation of a method is a procedure pointer, stored within a method table.) The following illustration addresses the question of which method implementation would be inherited.

Consider the situation in Figure 8. Class "W" defines a method "foo", implemented by procedure "proc1". Class "W" has two subclasses, "X" and "Y". Subclass "Y" overrides the implementation of "foo" with procedure "proc2". Subclass "X" does not override "foo". In addition, classes "X" and "Y" share a common subclass, "Z". That is, the IDL interface statement for class "Z" lists its parents as "X" and "Y" in that order.

The question is thus: which implementation of method "foo" does class "Z" inherit — procedure "proc1" defined by class "W", or procedure "proc2" defined by class "Y"? The procedure for performing inheritance that is defined by SOMClass resolves this ambiguity by using the *left path precedence* rule: when the same method is inherited from multiple ancestors, the procedure used to support the method is the one used by the *leftmost ancestor* from which the method is inherited. (The ordering of parent classes is determined by the order in which the class implementor lists the parents in the IDL specification for the class.)

In Figure 8, then, class "Z" inherits the implementation of method "foo" defined by class "W" (procedure "proc1"), rather than the implementation defined by class "Y" (procedure "proc2"), because "X" is the leftmost ancestor of "Z" from which the method "foo" is inherited. This rule may be interpreted as giving priority to classes whose instance interfaces are mentioned first in IDL interface definitions.

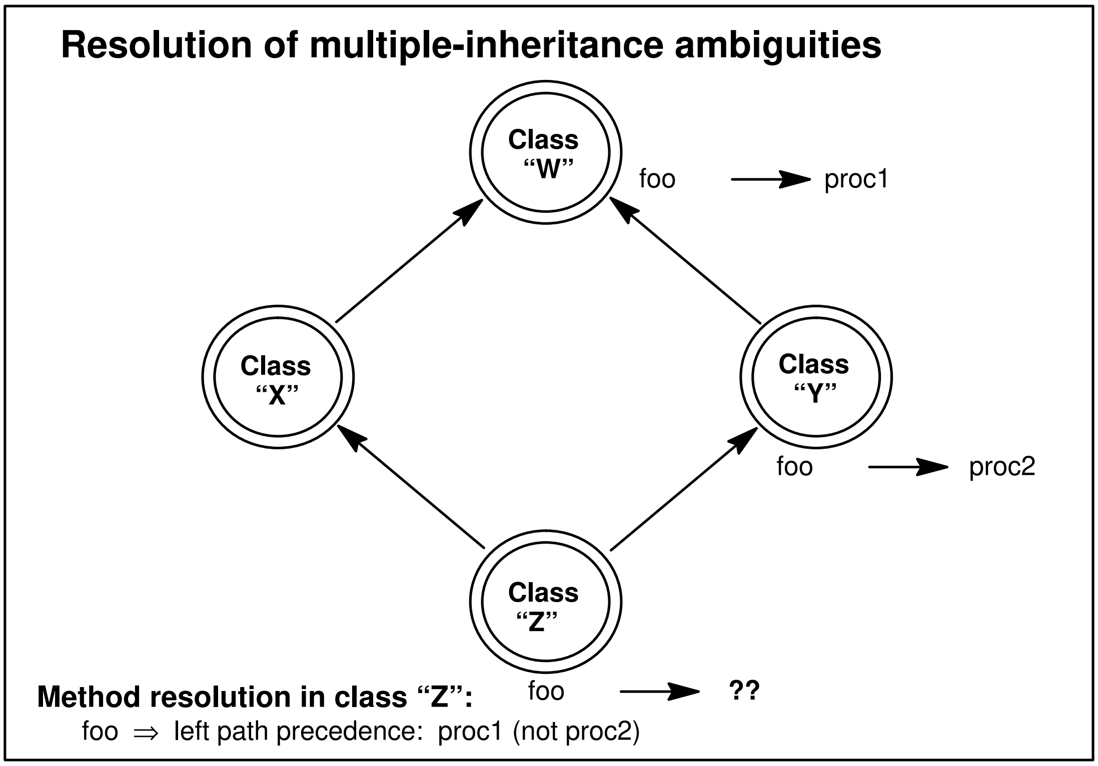


Figure 8. SOMClass uses the *left path precedence* rule to resolve some multiple inheritance ambiguities.

If a class implementor decides that the inherited implementation is not appropriate, and that procedure "proc2" is desired, then there are two alternatives in SOM: (1) An explicit metaclass can be introduced for Z that changes this default, and loads the new class's instance method table entry in whatever way is desired by the class designer (this can be viewed as modifying the default semantics of inheritance provided by SOMClass). Or, (2) the inherited method can be overridden. For example, class "Z" could override the implementation of "foo" that it inherits from "W" (procedure "proc1") with a new method procedure that uses a parent method call to invoke the implementation of "foo" defined by "Y" (procedure "proc2").

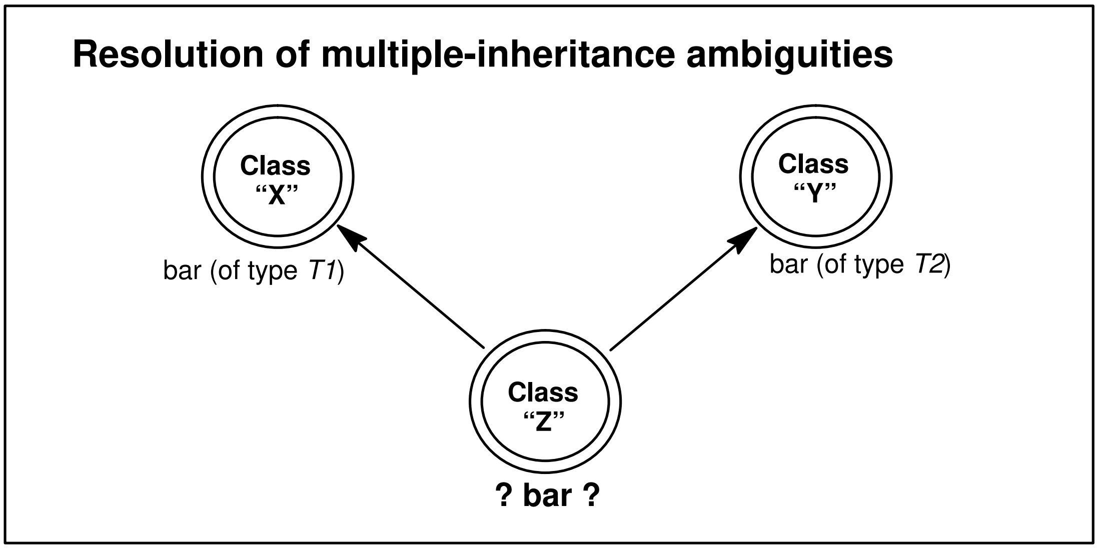


Figure 9. Some multiple inheritance ambiguities are illegal in IDL.

Another conflict that may arise with the use of multiple inheritance is when two ancestors of a class define different methods (in general, with different signatures) of the same name. For example, consider Figure 9. Class "X" defines a method "bar" with type *T1*, and class "Y" defines a method "bar" with type *T2*. Class "Z" is derived from both "X" and "Y", and "Z" does not override method "bar".

This example illustrates a method name that is "overloaded" — that is, used to name two entirely different methods (note that overloading is completely unrelated to overriding). This is not necessarily a difficult problem to handle. Indeed, the run-time SOM API allows the construction of a class that supports the two different "bar" methods illustrated in FIgure 9. (They are implemented using two different method-table entries, each of which is associated with its introducing class.)

<u>However, the interface to instances of such classes can*not* be defined using IDL</u>. IDL specifically forbids the definition of interfaces in which method names are overloaded. Furthermore, within SOM itself, the use of such classes can lead to anomalous behavior unless care is taken to avoid the use of name-lookup method resolution (discussed in the following section), since, in this case, a method name alone does not identify a unique method. For this reason, (statically declared) multiple-inheritance classes in SOM are currently restricted to those whose instance interfaces can be defined using IDL. Thus, the above example cannot be constructed with the aid of the SOM Compiler.

2.3 Method Resolution

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

- Class “Dog” introduces a method “bark”, and
- A subclass of “Dog”, called “BigDog”, overrides “bark”, and
- A client program creates an instance of either “Dog” or “BigDog” (depending on some run-time criteria) and invokes method “bark” on that instance.

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

SOM allows class implementors and client programs considerable flexibility in deciding how SOM performs method resolution. In particular, SOM supports three mechanisms for method resolution, described in order of increased flexibility and increased computational cost: offset resolution, name-lookup resolution, and dispatch-function resolution. Furthermore, the default behavior exhibited by each of these methods can be influenced (in various ways) by class implementors. For instance, the behavior of offset resolution can be affected by defining new metaclasses whose classes load their instance method tables differently; the behavior of dispatch-function resolution is especially designed to allow tailoring by overriding inherited instance methods.

Offset resolution

When using SOM’s C and C++ language bindings, offset resolution is the default way of resolving methods, because it is the fastest (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 the programmer), and
- The method to be invoked must be part of the introducing class’s static (IDL) interface definition.

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

An object’s method table is a table of pointers to the procedures that implement the methods that the object supports. This table is constructed by the object’s class and is shared among the class instances. The method table built by class (for its instances) is referred to as the class’s *instance method table.* This is useful terminology, since, in SOM, a class is itself an object with a method table (created by its metaclass) used to support method calls on the class.

Usually, offset method resolution is sufficient; however, in some cases, the more flexible name-lookup resolution is required.

Name-lookup resolution

Name-lookup resolution is similar to the method resolution techniques employed by Objective-C and Smalltalk. It is 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 run time, or
- The method is added to the class interface at run time, or
- The name of the class introducing the method isn't known until run time.

For example, a client program may use two classes that define two different methods of the same name, and it might not be known until run time which of the two methods should be invoked (because, for example, it will not be known until run time which class's instance the method will be applied to).

Name-lookup resolution is always performed by the class of an object, in response to a method call. (Offset resolution, by contrast, requires no method calls.) To perform name-lookup method resolution, the class of which the intended receiver is an instance obtains a method procedure pointer for the desired method that is appropriate for its instances. In general, this will require a name–based search through various data structures maintained by ancestor classes.

Offset and name-lookup resolution achieve the same net effect (that is, they select the same method procedure); they just achieve it differently (via different mechanisms for locating the method's method token). Offset resolution is faster, because it does not require searching for the method token, but name-lookup resolution is more flexible.

When defining (in SOM IDL) the interface to a class of objects, the class implementor can decide, for each method, whether the SOM Compiler will generate usage bindings that support name–lookup resolution for invoking the method. Regardless of whether this is done, however, application programs using the class can have SOM use either technique, on a per-method-call basis. Chapter 3, "Using SOM Classes in Client Programs," describes how client programs invoke methods."

Dispatch-function resolution

Dispatch-function resolution is the slowest, but most flexible, of the three method-resolution techniques. Dispatch functions permit method resolution to be <u>based on arbitrary rules associated with the class</u> of which the receiving object is an instance. Thus, a class implementor has complete freedom in determining how methods invoked on its instances are resolved.

With both offset and name-lookup resolution, the net effect is the same — the method procedure that is ultimately selected is the one supported by the class of which the receiver is an instance. For example, if the receiver is an instance of class "Dog", then Dog's method procedure will be selected; but if the receiver is an instance of class "BigDog", then BigDog's method procedure will be selected.

By contrast, dispatch-function resolution allows a class of instances to be defined such that the method procedure is selected using some other criteria. For example, the method procedure could be selected on the basis of the arguments to the method call, rather than on the receiver. The SOMobjects *Users Guide* (chapter 5) describes how SOM'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 *SOMobjects Developer Toolkit: Programmers Reference Manual*.

2.4 Interface vs Implementation

The remainder of this chapter describes how to define and implement 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, as follows:

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) 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* [.idl] *file*) and
- Method procedures written in the implementor's language of choice (the *implementation file*).

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

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

- An *implementation template* that serves as a guide for how the implementation file for the class should look. The class implementor fills in this template file with language-specific code to implement the methods that are available on the class' instances.
- *Header files* to be included (a) in the class's implementation file and (b) in client programs that use the class.

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

The subsequent sections of this chapter provide more thorough coverage of SOM IDL, the SOM Compiler, and implementing classes.

2.5 SOM Interface Definition Language

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

Constants (words to be used literally, such as keywords) appear in **bold**.
User-supplied elements appear in *italics*.

{ }	Groups related items together as a single item.
[ ]	Encloses an optional item.
*	Indicates zero or more repetitions of the preceding item.
+	Indicates one or more repetitions of the preceding item.
|	Separates alternatives.
_	Within a set of alternatives, an underscore indicates the default, if defined.

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

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

- IDL uses the ISO Latin-1 (8859.1) character set (as per the CORBA standard).
- White space is ignored except as token delimiters.
- C and C++ comment styles are supported.
- IDL supports standard C/C++ preprocessing, including macro substitution, conditional compilation, and source file inclusion.
- Identifiers (user-defined names for methods, attributes, instance variables, and so 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++.

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

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

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

Table 1. KEYWORDS FOR SOM IDL

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

Include directives

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

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

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

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

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

Type and constant declarations

IDL specifications may include type declarations and constant declarations as in C and C++, with the restrictions/extensions described below. [Note: For any reader not familiar with C, a recommended reference is *The C Programming Language* (2nd edition, 1988, Prentice Hall) by Brian W. Kernighan and Dennis M. Ritchie. See pages 36–40 for a discussion of type and constant declarations.]

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

Integral types

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

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

Floating point types

IDL supports the **float** and **double** floating-point types. The **float** type represents the IEEE single-precision floating-point numbers; **double** represents the IEEE double-precision floating-point numbers.

Character type

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

Boolean type

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

Octet type

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

Any type

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

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

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

Constructed types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

That specification would generate the following three types:

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

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

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

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

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

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

Union type

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

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

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

case–label+   type–spec   declarator ;

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

case *const–expr***:**
default:

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

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

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

is mapped onto the following C struct:

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

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

```
Foo v;

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

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

Template types (sequences and strings)

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

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

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

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

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

use:

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

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

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

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

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

results in the following C **struct**:

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

and an instance of this type is declared as follows:

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

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

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

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

_IDL_SEQUENCE_*type*

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

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

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

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

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

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

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

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

```
typedef char *foo;
```

A variable of this type is then declared as follows:

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

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

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

Arrays

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

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

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

Pointers

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

type **declarator*

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

Object types

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

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

```
Foo myobj;
```

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

```
Foo *myobj;
```

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

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

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

- It more closely resembles the bindings for C++, thus making it easier to change to the C++ bindings at a later date; and
- It is required for compatibility with existing SOM OIDL code.

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

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

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

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

Exception declarations

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

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

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

type–spec declarators ;

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

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

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

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

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

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

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

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

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

would map onto:

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

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

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

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

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

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

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

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

Table 2. STANDARD EXCEPTIONS DEFINED BY CORBA

```
module StExcep {
    #define ex_body { unsigned long minor; completion_status completed; }

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

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

};
```

Interface declarations

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

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

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

Note: In general, an "**interface** class-name" header must precede any subsequent implementation that references "class-name." For more discussion of multiple **interface** statements, refer to the later topic "Defining multiple interfaces within a single .idl file."

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

Constant, type, and exception declarations

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

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

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

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

```
MyParentClass::myType
```

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

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

Attribute declarations

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

Attributes are declared as follows:

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

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

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

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

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

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

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

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

Method (operation) declarations

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

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

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

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

Oneway keyword

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

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

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

Parameter list

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

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

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

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

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

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

Classes derived from **SOMObject** can declare methods that take a 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.

For all IDL types except **arrays**, if a parameter of a method is **out** or **inout**, then C/C++ clients must pass the address of a variable of that type (or the value of a pointer to that variable) rather than the variable itself. (For example, to invoke method "meth1" above, a pointer to a variable of type **float** must be passed in place of parameter "f".) For **arrays**, C/C++ clients must pass the address of the first element of the **array.**

If the return type of a method is a **struct, union, sequence**, or **any** type, then for C/C++ clients, the method returns the value of the C/C++ struct representing the IDL **struct, union, sequence**, or **any**. If the return type is **string,** then the method returns a pointer to the first character of the **string**. If the return type is **array**, then the method returns a pointer to the first element of the **array**.

The pointers implicit in the parameter types and return types for IDL method declarations are made explicit in SOM's C and C++ bindings. Thus, the stub procedure that the SOM Compiler generates for method “meth1”, above, has the following signature:

```
SOM_Scope short  SOMLINK meth1(char c, float *f)
```

For C and C++ clients, if a method has an **out** parameter of type unbounded **string** or unbounded **sequence**, then the storage for the **string** or for the “_value” member of the struct that represents the **sequence** must be allocated by the method. It is then the responsibility of the client program to free the storage when it is no longer needed. Similarly, if the return type of a method is unbounded **string**, unbounded **sequence**, or **array**, then storage will be allocated automatically, and it will be the responsibility of the client program to subsequently free it.

2.6 The SOM Compiler

The SOM Compiler translates the IDL definition of a SOM class into a set of “binding files” appropriate for the language that will implement the class’s methods and the language(s) that will use the class. These bindings make it more convenient for programmers to implement and use SOM classes. The SOM Compiler currently produces binding files for the C and C++ languages.

Important Note: C and C++ bindings can<u>not</u> both be generated during the same execution of the SOM compiler.

Generating binding files

The SOM Compiler operates in two phases:

- A <u>precompile phase</u>, in which a precompiler analyzes an OIDL or IDL class definition, and
- An <u>emission phase</u>, in which one or more <u>emitter programs</u> produce <u>binding files</u>.

Each binding file is generated by a separate emitter program. Setting the SMEMIT environment variable determines which emitters will be used, as described below. Note: In the discussion below, the <<u>filestem</u>> is determined by default from the <u>name of the source .idl file</u> with the “.idl”extension removed. Otherwise, a “filestem” modifier can be defined in the .idl file to specify another file name (see “Modifier statements” above).

Note: If changes to definitions in the .idl file later become necessary, the SOM Compiler should be rerun to update the current implementation template file, provided that the **c** or **xc** emitter is specified (either with the –s option or the SMEMIT environment variable, as described below). For more information on generating updates, see “Running incremental updates of the implementation template file” later in this chapter.

<u>The emitters for the C language</u> produce the following binding files:

*<filestem>***.c** — (produced by the **c** emitter)

This is a <u>template for a C source program</u> that implements a class’s methods. This will become the primary source file for the class. (The other binding files can be generated from the **.idl** file as needed.) This template implementation file contains “stub” procedures for each method introduced or overridden by the class. (The stub procedures are empty of code except for required initialization and debugging statements.) After the class implementor has supplied the code for the method procedures, running the **c** emitter again will update the implementation file to reflect changes made to the class definition (in the **.idl** file). These updates include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. Existing code within method procedures is not disturbed, however.

The **.c** file contains an **#include** directive for the **.ih** file, described below.

The contents of the C source template is controlled by the Emitter Framework file <SOMBASE>/include/ctm.efw. This file can be customized to change the template produced. For detailed information on changing the template file see the *Emitter Framework Guide and Reference*.

<*filestem*>**.h** — (produced by the **h** emitter)
This is the header file to be included by C client programs (programs that use the class). It contains the C usage bindings for the class, including macros for accessing the class's methods and a macro for creating new instances of the class. This header file includes the header files for the class's parent classes and its metaclass, as well as the header file that defines SOM's generic C bindings, **som.h.**

<*filestem*>**.ih** — (produced by the **ih** emitter)
This is the header file to be included in the implementation file (the file that implements the class's methods — the .c file). It contains the implementation bindings for the class, including:

- a **struct** defining the class's instance variables,
- macros for accessing instance variables,
- macros for invoking parent methods the class overrides,
- the <*className*>**GetData** macro used by the method procedures in the <*filestem*>**.c** file (see section 2.7),
- a <*className*>**NewClass** procedure for constructing the class object at run time, and
- any IDL types and constants defined in the IDL interface.

The emitters for the C++ language produce the following binding files:

<*filestem*>**.C** (for AIX) or <*filestem*>**.cpp** (for OS/2)
— (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 file for the class. (The other binding files can be generated from the **.idl** file as needed.) This template implementation file contains "stub" procedures for each method introduced or overridden by the class. (The stub procedures are empty of code except for required initialization and debugging statements.) After the class implementor has supplied the code for the method procedures, running the **xc** emitter again will update this file to reflect changes made to the class definition (in the **.idl** file). These updates include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. Existing code within method procedures is not disturbed, however.

The C++ implementation file contains an **#include** directive for the **.xih** file, described below.

The contents of the C++ source template is controlled by the Emitter Framework file <SOMBASE>/include/ctm.efw. This file can be customized to change the template produced. For detailed information on changing the template file see the *Emitter Framework Guide and Reference*.

<*filestem*>**.xh** — (produced by the **xh** emitter)
This is the 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 in-

cludes the header files for the class's parent classes and its metaclass, as well as the header file that defines SOM's generic C++ bindings, **som.xh**.

<filestem>.**xih** — (produced by the **xih** emitter)
This is the header file to be included in the implementation file (the file that implements the class's methods). It contains the implementation bindings for the class, including:

- a **struct** defining the class's instance variables,
- macros for accessing instance variables,
- macros for invoking parent methods the class overrides,
- the *<className>***GetData** macro (see section 2.7),
- a *<className>***NewClass** procedure for constructing the class object at run time, and
- any IDL types and constants defined in the IDL interface.

Other files the SOM Compiler generates:

<filestem>.**pdl** — (produced by the **pdl** emitter)
This file is the same as the .idl file from which it is produced except that all items within the .idl file that are marked as "private" are removed. (An item is marked as private by surrounding it with "#ifdef __PRIVATE__" and "#endif" directives.) Thus, the **pdl** (Public Definition Language) emitter can be used to generate a "public" version of a .idl file.

<filestem>.**def** (for OS/2) — (produced by the **def** emitter)
This file is used by the linker to package a class as a library. To combine several classes into a single library, you must merge the **exports** statements from each of their **.def** files into a single **.def** file for the entire library. When packaging multiple classes in a single library, you must also write a simple C procedure named **SOMInitModule** and add it to the export list. This procedure should call the routine *<className>***NewClass** for each class packaged in the library. The **SOMInitModule** procedure is called by the SOM Class Manager when the library is dynamically loaded.

<filestem>.**exp** (for AIX) — (produced by the **exp** emitter)
This file is used by the linker to package a class as a library. To combine several classes into a single library, you must merge the **exports** statements from each of their **.exp** files into a single **.exp** file for the entire library. When packaging multiple classes in a single library, you must also write a simple C procedure named **SOMInitModule** and add it to the export list. This procedure should call the routine *<className>***NewClass** for each class packaged in the library. The **SOMInitModule** procedure is called by the SOM Class Manager when the library is dynamically loaded.

The **Interface Repository** (produced by the **ir** emitter)
See Chapter 7 for a discussion on the Interface Repository.

Note: The C/C++ bindings generated by the SOM Compiler have the following limitation: If two classes named "ClassName" and "ClassNameC" are defined, the bindings for these two classes will clash. That is, if a client program uses the C/C++ bindings (includes the .h/.xh header file) for both classes, a name conflict will occur. Thus, class implementors should keep this limitation in mind when naming their classes.

SOM users can extend the SOM Compiler to generate additional files by writing their own emitters. To assist users in extending the SOM Compiler, SOM provides an Emitter Framework — a collection of classes and methods useful for writing object-oriented emitters that the SOM Compiler can invoke. For more information, see the *Emitter Framework Guide and Reference*.

Note re: porting SOM classes: The header files (binding files) that the SOM Compiler generates will only work on the platform (operating system) on which they were generated. Thus, when porting SOM classes from the platform where they were developed to another platform, the header files must be regenerated from the .idl file by the SOM Compiler on that target platform.

Environment variables affecting the SOM Compiler

To execute the SOM Compiler on one or more files that contain IDL specifications for one or more classes, use the command:

sc [–*options*] *files*

where "files" specifies one or more .idl files.

Available "–options" for the command are detailed in the next topic. The operation of the SOM Compiler (whether it produces C binding files or C++ binding files, for example) is also controlled by a set of environment variables that can be set before the **sc** command is issued. The applicable environment variables are as follows:

SMEMIT — Determines which output files the SOM Compiler produces. Its value consists of a list of items separated by semicolons for OS/2, or by semicolons or colons for AIX. Each item designates an emitter to execute. For example, the statement:

```
SET SMEMIT=c;h;ih              (for OS/2)
export SMEMIT="c;h;ih"         (for AIX)
```

directs the SOM Compiler to produce the C binding files "hello.c", "hello.h", and"hello.ih" from the "hello.idl" input specification. By comparison,

```
SET SMEMIT=xc;xh;xih           (for OS/2)
export SMEMIT="xc;xh;xih"      (for AIX)
```

directs the SOM Compiler to produce C++ binding files "hello.C" (for AIX) or "hello.cpp" (for OS/2), "hello.xh", and "hello.xih" from the "hello.idl" input specification.

By default, all output files are placed in the same directory as the input file. If the SMEMIT environment variable is not set, then a default value of "h;ih" is assumed.

SMINCLUDE — Specifies where the SOM Compiler should look for .idl files #included by the .idl file being compiled. Its value should be one or more directory names separated by a semicolon when using OS/2, or separated by a semicolon or colon when using AIX. Directory names can be specified with absolute or relative pathnames. For example:

```
SET SMINCLUDE=.;..\MYSCDIR;C:\TOOLKT20\C\INCLUDE;
                                          (for OS/2)

export SMINCLUDE=.:myscdir:/u/som/include
                                          (for AIX)
```

The default value of the SMINCLUDE environment variable is the “include” subdirectory of the directory into which SOM has been installed.

SMTMP — Specifies the directory that the SOM Compiler should use to hold intermediate output files. This directory should *not* coincide with the directory of the input or output files. For AIX, the default setting of SMTMP is /tmp; for OS/2, the default setting of SMTMP is the root directory of the current drive.

OS/2 example:

```
SET SMTMP=%TMP%
```

tells the SOM Compiler to use the same directory for temporary files as given by the setting of the TMP environment variable (the default location for temporary system files). Or,

```
SET SMTMP=..\MYSCDIR\GARBAGE
```

tells the SOM Compiler to place the temporary files in the GARBAGE directory.

AIX example:

```
export SMTMP=$TMP
export SMTMP=../myscdir/garbage
```

SMKNOWNEXTS — Specifies additional emitters to which the SOM Compiler should add a header. For example, if you were to write a new emitter for Pascal, called “emitpas”, then by default the SOM Compiler would not add any header comments to it. However, by setting SMKNOWNEXTS=pas, as shown:

```
set SMKNOWNEXTS=pas          (for OS/2)
export SMKNOWNEXTS=pas       (for AIX)
```

the SOM Compiler will add a header to files generated with the “emitpas” emitter. The “header” added is a SOM Compiler-generated message plus any comments, such as copyright statements, that appear at the head of your .idl input file. For details on writing your own emitter, see the *Emitter Framework Guide and Reference*.

SOMIR — Specifies the name (or list of names) of the Interface Repository file. The **ir** emitter, if run, creates the Interface Repository, or checks it for consistency if it already exists. If the **–u** option is specified when invoking the SOM Compiler, the **ir** emitter also updates an existing Interface Repository.

Running the SOM Compiler

The syntax of the **sc** command for running the SOM Compiler takes the form:

sc [*–options*] *files*

The “files” specified in the **sc** command denote one or more files containing the IDL class definitions to be compiled. If no extension is specified, .idl is assumed. By default, the <filestem> of the .idl file determines the filestem of each emitted file. Otherwise, a “filestem” modifier can be defined in the .idl file to specify another name (see “Modifier statements” discussed earlier).

Selected “–options” can be specified individually, as a string of option characters, or as a combination of both. Any option that takes an argument either must be specified individually or must appear as the final option in a string of option characters. Available options and their purposes are as follows:

–C *n*
: Sets the maximum allowable size for a simple comment in the .idl file (default: 32767). This is only needed for very large single comments.

–D *name*[*=def*]
: Defines *name* as in a #define directive. The default *def* is 1. This option is the same as the –D option for the C compiler. Note: This option can be used to define **__PRIVATE__** so that the SOM Compiler will also compile any methods and attributes that have been defined as private using the directive **#ifdef __PRIVATE__**; however, the –p option does the same thing more easily.

–E *variable=value*
: Sets an environment variable. (See the previous topic for a discussion of the available environment variables: SMEMIT, SMINCLUDE, SMTMP, and SMNOIR.)

–I *dir*
: When looking for #included files, looks first in dir, then in the standard directories (same as the C compiler **–I** option).

–S *n*
: Sets the total allowable amount of unique string space used in the IDL specification for names and passthru lines (default: 32767). This is only needed for very large .idl files.

–U *name*
: Removes any initial definition (via a #define preprocessor directive) of symbol *name*.

–V
: Displays version information about the SOM Compiler.

–c
: Turns off comment processing. This allows comments to appear anywhere within an IDL specification (rather than in restricted places), and it causes comments not to be transferred to the output files that the SOM Compiler produces.

–d *directory*
: Specifies a directory where all output files should be placed. If the –d option is not used, all output files are placed in the same directory as the input file.

–h or **–?**
: Produces a listing of this option list. (This option is typically used in an **sc** command that does not include a .idl file name).

–i *filename*
: Specifies the name of the class definition file. Use this option to override the built-in assumption that the input file will have a .idl extension. Any *filename* supplied with the –i option is used exactly as it is specified.

–m *name*[**=***value*]

Adds a global modifier. The currently supported global modifiers are as follows:

addprefixes

Adds 'functionprefixes' to the method procedure prototypes during an incremental update of the implementation template file. This option applies only when rerunning the **c** or **xc** emitter on an IDL file that previously did *not* specify a functionprefix. A class implementor who later decides to use prefixes should add a line in the 'implementation' section of the .idl file containing the specification:

functionprefix = *prefix*

(as described earlier in the topic "Modifier statements") and then rerun the c or xc emitter using the **–maddprefixes** option. The method procedure prototypes in the implementation file will then be updated so that each method name includes the assigned prefix. (This option does not support changes to existing prefix names, nor does it apply for OIDL files.)

addstar This option causes all interface references to have a '*****' added to them for the C bindings. See the earlier section entitled "Class types" for further details.

comment=*comment string*

where *comment string* can be either of the designations: "/*" or "//". This option indicates that comments marked in the designated manner in the .idl file are to be completely ignored by the SOM Compiler and will *not* be included in the output files. <u>Note</u>: Comments on lines beginning with "//#" are always ignored by the SOM Compiler.

corba This option directs the SOM Compiler to compile the input definition according to strict CORBA-defined IDL syntax. This means, for example, that comments may appear anywhere and that pointers are not allowed. When the **–mcorba** option is used, parts of a .idl file surrounded by **#ifdef __SOMIDL__** and **#endif** directives are ignored. This option can be used to determine whether all nonstandard constructs (those specific to SOM IDL) are properly protected by **#ifdef __SOMIDL__** and **#endif** directives.

csc This option forces the OIDL compiler to be run. This is required only if you want to compile an OIDL file that does not have an extension of .csc or .sc.

emitappend

This option causes emitted files to be appended at the end of existing files of the same name.

noheader

This option ensures that the SOM Compiler does not add a header to the beginning of an emitted file.

noint This option directs the SOM Compiler not to warn about the portability problems of using `int`'s in the source.

nolock This option causes the Interface Repository Emitter **emitir** (see Chapter 7, "Interface Repository Framework") to leave the IR unlocked when updates are made to it. This can improve performance on networked file systems. By not locking the IR, however, there is the risk of multiple processes attempting to write to the same IR, with unpredictable results. This option should only be used when you know that only one process is updating an IR at once.

nopp This option directs the SOM Compiler not to run the SOM preprocessor on the .idl input file.

notc This option directs the SOM Compiler not to create TypeCode information when emitting files. This is required only when the .idl files contain some undeclared types. This option is typically used when compiling converted .csc files that have not had typing information added.

nouseshort
This option directs the SOM Compiler not to generate short forms for type names in the .h and .xh public header files. This can be useful to save disk space.

pp=*preprocessor*
This option directs the SOM Compiler to use the specified preprocessor as the SOM preprocessor, rather than the default "somcpp". Any standard C/C++ preprocessor can be used as a preprocessor for IDL specifications.

tcconsts This option directs the SOM Compiler to generate TypeCode constants in the .h and .xh public header files. Please refer to the Interface Repository (described in Chapter 7) for more details.

Note: All command-line **–m** modifier options can be specified in the environment by changing them to UPPERCASE and prepending "SM" to them. For example, if you want to always set the options "–mnotc" and "–maddstar", set corresponding environment variables as follows:

On OS/2:

```
set SMNOTC=1
set SMADDSTAR=1
```

On AIX:

```
export SMNOTC=1
export SMADDSTAR=1
```

–p Causes the "private" sections of the IDL file to be included in the compilation (that is, sections preceded by #ifdef __PRIVATE__ that contain private methods and attributes). Note: The –p option is equivalent to the earlier option –D__PRIVATE__.

–r Checks that all names specified in the release order statement are valid method names (default: FALSE).

–s *string* Substitutes *string* in place of the contents of the **SMEMIT** environment variable for the duration of the current **sc** command. This determines which emitters will be run and, hence, which output files will be produced. (If a list of values is given, on OS/2 *only* the list must be enclosed in double quotes.)

The –s option is a convenient way to override the SMEMIT environment variable. In OS/2 for example, the command:

```
> SC –s"h;c" EXAMPLE
```

is equivalent to the following sequence of commands:

```
> SET OLDSMEMIT=%SMEMIT%
> SET SMEMIT=H;C
> SC EXAMPLE
> SET SMEMIT=%OLDSMEMIT%
```

Similarly, in AIX the command:

```
> sc –sh";"c example
```

is equivalent to the following sequence of commands:

```
> export OLDSMEMIT=$SMEMIT
> export SMEMIT=h";"c
> sc example
> export SMEMIT=$OLDSMEMIT
```

–u Updates the Interface Repository (default: no update). With this option, the Interface Repository will be updated even if the **ir** emitter is not explicitly requested in the SMEMIT environment variable or the **–s** option.

–v Uses verbose mode to display informational messages (default: FALSE). This option is primarily intended for debugging purposes and for writers of emitters.

–w Suppresses warning messages (default: FALSE).

The following sample commands illustrate various options for the **sc** command:

`sc –sc hello.idl`	Generates file "hello.c".
`sc –hV`	Generates a help message and displays the version of the SOM Compiler currently available.
`sc –vsh";"ih hello.idl`	Generates "hello.h" and "hello.ih" with informational messages.
`sc –sxc –doutdir hello.idl`	Generates "hello.xc" in directory "outdir".

2.7 Implementing SOM 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 file. To assist users in implementing classes, the SOM Compiler produces a template implementation file — a type-correct guide for how the implementation of a class should look. The class implementor then modifies this template to implement the class's methods.

The SOM Compiler can also update the implementation file to reflect later changes made to a class's interface definition file (the .idl file). These *incremental updates* include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. These updates to the implementation file, however, do *not* disturb existing code in the method procedures. These updates are discussed further in "Running incremental updates of the implementation template file" later in this section.

For C programmers, the SOM Compiler generates a <*filestem*>**.c** file; and for C++ programmers, the SOM Compiler generates a <*filestem*>**.C** file (for AIX) or a <*filestem*>**.cpp** file (for OS/2). To specify whether the SOM Compiler should generate a C or C++ implementation template, set the value of the SMEMIT environment variable, or use the **–s** option when running the SOM Compiler. (See the previous section.)

Note: As this chapter describes, C++ can be used to implement a SOM class by using C++ to define the instance variables introduced by the class and to define the procedures that implement the overridden and introduced methods of the class. This does *not* mean that the C++ class defined by the C++ usage bindings for a SOM class (described in Chapter 3) can be subclassed in C++ to create new C++ or SOM classes.†

The implementation template

Consider the following IDL description of the "Hello" class:

```
#include <somobj.idl>

interface Hello : SOMObject
{
    void sayHello();
    // This method outputs the string "Hello, World!".
};
```

From this IDL description, the SOM Compiler generates the following C implementation template, hello.c (a C++ implementation template, hello.C or hello.cpp, is identical except that the *#include*d file is <hello.xih> rather than <hello.ih>):

† The reason why the C++ implementation of a SOM class involves the definition of C++ procedures (not C++ methods) to support SOM methods is that there is no language-neutral way to call a C++ method. Only C++ code can call C++ methods, and this calling code must be generated by the same compiler that generates the method code. In contrast, the method procedures that implement SOM methods must be callable from any language, without knowledge on the part of the object client as to which language is used to implement the resolved method procedure.

```
#define Hello_Class_Source
#include <hello.ih>

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

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

The first line defines the "Hello_Class_Source" symbol, which is used in the SOM-generated implementation header files for C to determine when to define various functions, such as "HelloNewClass." For interfaces defined within a module, the directive "#define <className>_Class_Source" is replaced by the directive "#define SOM_Module_<moduleName>_Source".

The second line (`#include <hello.ih>` for C, or `#include <hello.xih>` for C++) includes the SOM-generated implementation header file. This file defines a **struct** holding the class's instance variables, macros for accessing instance variables, macros for invoking parent methods, and so forth.

Stub procedures for methods

For each method introduced or overridden by the class, the implementation template includes a *stub procedure* — a procedure that is empty except for an *initialization* statement, a *debugging* statement, and possibly a *return* statement. The stub procedure for a method is preceded by any comments that follow the method's declaration in the IDL specification.

For method "sayHello" above, the SOM Compiler generates the following prototype of the stub procedure:

```
SOM_Scope void SOMLINK sayHello(Hello somSelf, Environment *ev)
```

The "SOM_Scope" symbol is defined in the implementation header file as either "extern" or "static," as appropriate. The term "void" signifies the return type of method "sayHello". The "SOMLINK" symbol is defined by SOM; it represents the keyword needed to link to the C or C++ compiler, and its value is system-specific. Using the "SOMLINK" symbol allows the code to work with a variety of compilers without modification.

Following the "SOMLINK" symbol is the name of the procedure that implements the method. If no **functionprefix** modifier has been specified for the class, then the procedure name is the same as the method name. If a **functionprefix** modifier is in effect, then the procedure name is generated by prepending the specified prefix to the method name. For example, if the class definition contained the following statement:

```
functionprefix = xx_;
```

then the prototype of the stub procedure for method "sayHello" would be:

```
SOM_Scope void SOMLINK xx_sayHello(Hello somSelf, Environment *ev)
```

The **functionprefix** cannot be

```
<classname>_
```

since this is used in method invocation macros defined by the C usage bindings.

Following the procedure name is the formal parameter list for the method procedure. Because each SOM method always receives at least one argument (a pointer to the SOM object that responds to the method), the first parameter name in the prototype of each stub procedure is called **somSelf**. (The macros defined in the implementation header file rely on this convention.) The **somSelf** parameter is a pointer to an object that is an instance of the class being implemented (here, class “Hello”) or an instance of a class derived from it.

Unless the IDL specification of the class included the **callstyle=oidl** modifier, then the formal parameter list will include one or two additional parameters before the parameters declared in the IDL specification: an **(Environment *ev)** input/output parameter, which permits the return of exception information, and, if the IDL specification of the method includes a context specification, a **(Context *ctx)** input parameter. These parameters are prescribed by the CORBA standard. For more information on using the **Environment** and **Context** parameters, see the section entitled “Exceptions and error handling” in Chapter 3, “Using SOM Classes in Client Programs,” and the book *The Common Object Request Broker: Architecture and Specification*, published by Object Management Group and X/Open.

The first statement in the stub procedure for method “sayHello” is the statement:

```
/* HelloData *somThis = HelloGetData(somSelf); */
```

This statement is enclosed in comments only when the class does *not* introduce any instance variables. The purpose of this statement, for classes that do introduce instance variables, is to initialize a local variable (**somThis**) that points to a *structure* representing the instance variables introduced by the class. The **somThis** pointer is used by the macros defined in the “Hello” implementation header file to access those instance variables. (These macros are described below.) In this example, the “Hello” class introduces no instance variables, so the statement is commented out. If instance variables are later added to a class that initially had none, then the comment characters can be removed if access to the variable is required.

The “HelloData” type and the “HelloGetData” macro used to initialize the **somThis** pointer are defined in the implementation header file. Within a method procedure, class implementors can use the **somThis** pointer to access instance data, or they can use the convenience macros defined for accessing each instance variable, as described below.

To implement a method so that it can modify a local copy of an object’s instance data without affecting the object’s real instance data, declare a variable of type <*className*>**Data** (for example, “HelloData”) and assign to it the structure that **somThis** points to; then make the **somThis** pointer point to the copy. For example:

```
HelloData myCopy = *somThis;
somThis = &myCopy;
```

Next in the stub procedure for method “sayHello” is the statement:

```
HelloMethodDebug(”Hello”,"sayHello”);
```

This statement facilitates debugging. The “HelloMethodDebug” macro is defined in the implementation header file. It takes two arguments, a class name and a method name. If debugging is turned on (that is, if global variable **SOM_TraceLevel** is set to one in the calling program), the macro produces a message each time the method procedure is entered. (See the next Chapter 3, “Using SOM Classes in Client Programs,” for information on debugging with SOM.) 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(c,m)** following the #include directive for the implementation header file. (This can yield a slight performance improvement.) For example, to permanently disable debugging for the “Hello” class, insert the following lines in the hello.c implementation file following the line “#include hello.ih” (or “#include hello.xih,” for classes implemented in C++):

```
#undef HelloMethodDebug
#define HelloMethodDebug(c,m) SOM_NoTrace(c,m)
```

The way in which the stub procedure ends is determined by whether the method is a new or an overriding method.

- For non-overriding (new) methods, the stub procedure ends with a return statement (unless the return type of the method is **void**). The class implementor should customize this return statement.
- For overriding methods, the stub procedure ends by making a "parent method call" for each of the class's parent classes. If the method has a return type that is not **void**, the last of these parent method calls is returned as the result of the method procedure. The class implementor can customize this return statement if needed (for example, if some other value is to be returned, or if the parent method calls should be made before the method procedure's own processing). See the next section for a discussion of parent method calls.

If a **classinit** modifier was specified to designate a user-defined procedure that will initialize the "Hello" class object, as in the statement:

```
classinit = HInit;
```

then the implementation template file would include the following stub procedure for "HInit", in addition to the stub procedures for Hello's methods:

```
void  SOMLINK HInit(SOMClass *cls)
{

}
```

This stub procedure is then filled in by the class implementor. If the class definition specifies a **functionprefix** modifier, the **classinit** procedure name is generated by prepending the specified prefix to the specified **classinit** name, as with other stub procedures.

Extending the implementation template

To implement a method, add code to the body of the stub procedure. In addition to standard C or C++ code, class implementors can also use any of the functions, methods, and macros SOM provides for manipulating classes and objects. Chapter 3, "Using SOM Classes in Client Programs," discusses these functions, methods, and macros.

In addition to the functions, methods, and macros SOM provides for both class clients and class implementors, SOM provides two facilities especially for class implementors. They are for (1) accessing instance variables of the object responding to the method and (2) making parent method calls, as follows.

Accessing internal instance variables

To access internal instance variables, class implementors can use either of the following forms:

_variableName

somThis–>*variableName*

To access internal instance variables "a", "b", and "c", for example, the class implementor could use either _a, _b, and _c, or somThis–>a, somThis–>b, and somThis–>c. These expressions can appear on either side of an assignment statement. The **somThis** pointer must be properly initialized in advance using the <*className*>**GetData** procedure, as shown above.

Instance variables can be accessed only within the implementation file of the class that introduces the instance variable, and not within the implementation of subclasses or within client programs. (To allow access to instance data from a subclass or from client programs, use an *attribute* rather than an instance variable to represent the instance data.) For C++ programmers, the *_variableName* form is available only if the macro **VARIABLE_MACROS** is defined (that is, **#define VARIABLE_MACROS**) in the implementation file prior to including the .xih file for the class.

Making parent method calls

In addition to macros for accessing instance variables, the implementation header file that the SOM Compiler generates also contains definitions of macros for making "parent method calls." When a class overrides a method defined by one or more of its parent classes, often the new implementation simply needs to augment the functionality of the existing implementation(s). Rather than completely re-implementing the method, the overriding method procedure can conveniently invoke the procedure that one or more of the parent classes uses to implement that method, then perform additional computation (redefinition) as needed. The parent method call can occur anywhere within the overriding method. (See Example 3 of the SOM IDL tutorial.)

The SOM-generated implementation header file defines the following macros for making parent-method calls from within an overriding method:

<*className*>**_parent_**<*parentClassName*>_<*methodName*>
(for each parent class of the class overriding the method), and

<*className*>**_parents_**<*methodName*>.

For example, given class "Hello" with parents "File" and "Printer" and overriding method **somInit** (the SOM method that initializes each object), the SOM Compiler defines the following macros in the implementation header file for "Hello":

```
Hello_parent_Printer_somInit
Hello_parent_File_somInit
Hello_parents_somInit
```

Each macro takes the same number and type of arguments as <*methodName*>. The <*className*>**_parent_**<*parentClassName*>_<*methodName*> macro invokes the implementation of <*methodName*> inherited from <*parentClassName*>. Hence, using the macro "Hello_parent_File_somInit" invokes File's implementation of **somInit**.

The <*className*>**_parents_**<*methodName*> macro invokes the parent method for *each* parent of the child class that supports <*methodName*>. That is, "Hello_parents_somInit" would invoke File's implementation of **somInit**, followed by Printer's implementation of **somInit**. The <*className*>**_parents_**<*methodName*> macro is redefined in the binding file each time the class interface is modified, so that if a parent class is added or removed from the class definition, or if <*methodName*> is added to one of the existing parents, the macro <*className*>**_parents_**<*methodName*> will be redefined appropriately.

Compiling and linking

After filling in the method stub procedures, the implementation template file can be compiled and linked with a client program as follows. (In these examples, the environment variable SOMBASE represents the directory in which SOM has been installed.)

For AIX: When the client program (main.c) and the implementation file (hello.c) are written in C:

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

When the client program and the implementation file are written in C++:

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

For OS/2: When the client program (main.c) and the implementation file (hello.c) are in C:

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

When the client program and the implementation file are written in C++:

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

If the class definition (in the .idl file) changes, run the SOM Compiler again. This will generate new header files, and it will update the implementation file to include any:

- New comments,
- Stub procedures for any new methods, and
- Revised method procedure prototypes for methods whose signatures have been changed in the .idl file.

After rerunning the SOM Compiler, add to the implementation file the code for any newly added method procedures, and recompile the implementation file with the client program.

2.8 Creating a SOM Class Library

One of the principal advantages of SOM is that it makes "black box" (or binary) reusability possible. Consequently, SOM classes are frequently packaged and distributed as class libraries. A *class library* holds the actual implementation of one or more classes and can be dynamically loaded and unloaded as needed by applications. Importantly, class libraries can also be replaced independently of the applications that use them and, provided that the class implementor observes simple SOM guidelines for preserving binary compatibility, can evolve and expand over time.

Since class libraries are not programs, users cannot execute them directly. To enable users to make direct use of your classes, you must also provide one or more programs that create the classes and objects that the user will need. This section describes how to package your classes in a SOM class library and what you must do to make the contents of the library accessible to other programs.

On AIX, class libraries are actually produced as AIX shared libraries, whereas on OS/2 they appear as dynamically-linked libraries (or DLLs). The term "DLL" is sometimes used to refer to either an AIX or OS/2 class library, and (by convention only) the file suffix ".dll" is used for SOM class libraries on both platforms.

A program can use a class library containing a given class or classes in one of two ways:

1. If the programmer employs the SOM bindings to instantiate the class and invoke its methods, the resulting client program contains static references to the class. The operating system will automatically resolve those references when the program is loaded, by also loading the appropriate class library.
2. If the programmer uses only the dynamic SOM mechanisms for finding the class and invoking its methods (for example, by invoking **somFindClass, somFindMethod, somLookupMethod, somDispatch, somResolveByName**, and so forth), the resulting client program does not contain any static references to the class library. Thus, SOM will load the class library dynamically during execution of the program. Note: For SOM to be able to load the class library, the **dllname** modifier must be set in the .idl file. (See the topic "Modifier statements" earlier in this chapter.)

Because the provider of a class library cannot predict which of these ways a class will be used, SOM class libraries must be built such that either usage is possible. The first case above requires the class library to export the entry points needed by the SOM bindings, whereas the second case requires the library to provide an initialization function to create the classes it contains. The following topics discuss each case.

Building export files

The SOM Compiler provides an "exp" emitter for AIX and a "def" emitter for OS/2 to produce the necessary exported symbols for each class. For example, to generate the necessary exports for a class "A", issue the **sc** command with one of the following **–s** options. (For a discussion of the **sc** command and options, see "Running the SOM Compiler" earlier in this chapter.)

For AIX, this command generates an "a.exp" file:

```
sc –sexp a.idl
```

For OS/2, this command generates an "a.def" file:

```
sc –sdef a.idl
```

Typically, a class library contains multiple classes. To produce an appropriate export file for each class that the library will contain, you can create a new export file for the library itself by combining the exports from each “exp” or “def” file into a single file. Following are examples of a combined export “exp” file for AIX and a combined “def” file for OS/2. Each example illustrates a class library composed of three classes, “A”, “B”, and “C”.

AIX “exp” file:

```
#! abc.dll
ACClassData
AClassData
ANewClass
BCClassData
BClassData
BNewClass
CCClassData
CClassData
CNewClass
```

OS/2 “def” file:

```
LIBRARY abc INITINSTANCE
DESCRIPTION 'abc example class library'
PROTMODE
DATA MULTIPLE NONSHARED LOADONCALL
EXPORTS
   ACClassData
   AClassData
   ANewClass
   BCClassData
   BClassData
   BNewClass
   CCClassData
   CClassData
   CNewClass
```

Other symbols in addition to those generated by the “def” or “exp” emitter can be included if needed, but this is not required by SOM. One feature of SOM is that a class library needs no more than three exports per class (by contrast, many OOP systems require externals for every method as well). One required export is the name of a procedure to create the class (<*className*>**NewClass**), and the others are two external data structures that are referenced by the SOM bindings. Strictly speaking, only the <*className*>**ClassData** structure is required. The other, <*className*>**CClassData**, is used only by SOM 1.0 bindings, and is retained for backward compatibility. If you know your class will never be accessed by programs built with SOM 1.0, there is no need to export the <*className*>**CClassData** symbol.

Specifying the initialization function

An initialization function for the class library must be provided to support dynamic loading of the library by the SOM Class Manager. The SOM Class Manager expects that, whenever it loads a class library, the initialization function will create and register class objects for all of the classes contained in the library. These classes are then managed as a group (called an *affinity group*). One class in the affinity group has a privileged position — namely, the class that was specifically requested when the library was loaded. If that class (that is, the class that caused loading to occur) is subsequently unregistered, the SOM Class Manager will automatically unregister all of the other classes in the affinity group as well, and will unload the class library. Similarly, if the SOM Class Manager is explicitly asked to unload the class library, it will also automatically unregister and free all of the classes in the affinity group.

It is the responsibility of the class-library creator to supply the initialization function. The interface to the initialization function is given by the following C/C++ prototype:

#ifdef __IBMC__
#pragma linkage (SOMInitModule, system)
#endif

SOMEXTERN void SOMLINK SOMInitModule (**long** *majorVersion*,
long *minorVersion*,
string *className*);

The parameters provided to this function are the *className* and the major/minor version numbers of the class that was requested when the library was loaded (that is, the class that caused loading). The initialization function is free to use or to disregard this information; nevertheless, if it fails to create a class object with the required name, the SOM Class Manager considers the load to have failed. As a rule of thumb, however, if the initialization function invokes a <*className*>**NewClass** procedure for each class in the class library, this condition will always be met. Consequently, the parameters supplied to the initialization function are not needed in most cases.

Here is a typical class-library initialization function, written in C, for a library with three classes ("A", "B", and "C"):

```
#include "a.h"
#include "b.h"
#include "c.h"
#ifdef __IBMC__
  #pragma linkage (SOMInitModule, system)
#endif

SOMEXTERN void  SOMLINK SOMInitModule (long majorVersion,
                          long minorVersion, string className)
{
    SOM_IgnoreWarning (majorVersion);  /* This function makes  */
    SOM_IgnoreWarning (minorVersion);  /* no use of the passed */
    SOM_IgnoreWarning (className);     /* arguments.   */
    ANewClass (A_MajorVersion, A_MinorVersion);
    BNewClass (B_MajorVersion, B_MinorVersion);
    CNewClass (C_MajorVersion, C_MinorVersion);
}
```

The source code for the initialization function can be added to one of the implementation files for the classes in the library, or you can put it in a separate file and compile it independently.

Creating the import library

Finally, for each of your class libraries, you should create an import library that can be used by client programs (or by other class libraries that use your classes) to resolve the references to your classes.

Here is an example illustrating all of the steps required to create a class library ("abc.dll") that contains the three classes "A", "B", and "C".

1. Compile all of the implementation files for the classes that will be included in the library. Include the initialization function also.

   For AIX written in **C**:

   ```
   xlc -I. -I$SOMBASE/include -c a.c
   xlc -I. -I$SOMBASE/include -c b.c
   xlc -I. -I$SOMBASE/include -c c.c
   xlc -I. -I$SOMBASE/include -c initfunc.c
   ```

For AIX written in **C++**:

```
xlC –I. –I$SOMBASE/include –c a.C
xlC –I. –I$SOMBASE/include –c b.C
xlC –I. –I$SOMBASE/include –c c.C
xlC –I. –I$SOMBASE/include –c initfunc.C
```

For OS/2 written in **C**:

```
icc –I. –I%SOMBASE%\include –c a.c
icc –I. –I%SOMBASE%\include –c b.c
icc –I. –I%SOMBASE%\include –c c.c
icc –I. –I%SOMBASE%\include –c initfunc.c
```

For OS/2 written in **C++**:

```
icc –I. –I%SOMBASE%\include –c a.cpp
icc –I. –I%SOMBASE%\include –c b.cpp
icc –I. –I%SOMBASE%\include –c c.cpp
icc –I. –I%SOMBASE%\include –c initfunc.cpp
```

2. Produce an export file for each class.

   For AIX:

```
sc –sexp a.idl b.idl c.idl
```

   For OS/2:

```
sc –sdef a.idl b.idl c.idl
```

3. Manually combine the exported symbols into a single file.

   For AIX, create a file "abc.exp" from "a.exp", "b.exp", and "c.exp". Do *not* include the initialization function (**SOMInitModule**) in the export list.

   For OS/2, create a file "abc.def" from "a.def", "b.def", and "c.def". Include the initialization function (**SOMInitModule**) in the export list, unless your initialization function does not need arguments and you explicitly invoke it yourself from an OS/2 DLL initialization routine.

4. Using the object files and the export file, produce a binary class library.

   For AIX:

```
ld –o abc.dll –bE:abc.exp –e SOMInitModule –H512 –T512 \
   a.o b.o c.o initfunc.o –lc –L$SOMBASE/lib –lsomtk
```

   The **–o** option assigns a name to the class library ("abc.dll"). The **–bE:** option designates the file with the appropriate export list. The **–e** option designates **SOMInitModule** as the initialization function. The **–H** and **–T** options must be supplied as shown; they specify the necessary alignment information for the text and data portions of your code. The **–l** options name the specific libraries needed by your classes. If your classes make use of classes in other class libraries, include a **–l** option for each of these also. The **ld** command looks for a library named "lib<*x*>.a", where <*x*> is the name provided with each **–l** option. The **–L** option specifies the directory where the "somtk" library resides.

   For OS/2:

```
set LIB=%SOMBASE%\lib;%LIB%
link386 /noi /packd /packc /align:16 /exepack \
    a.obj b.obj c.obj initfunc.obj, abc.dll,,os2386 somtk, \
    abc.def
```

   If your classes make use of classes in other class libraries, include the names of their import libraries immediately after "somtk" (before the next comma).

5. Create an import library that corresponds to the class library, so that programs and other class libraries can use (import) your classes.

   For AIX:

   ```
   ar ruv libabc.a abc.exp
   ```

   The first filename ("libabc.a") specifies the name to give to the import library. It should be of the form "lib<*x*>.a", where <*x*> represents your class library. The second filename ("abc.exp") specifies the exported symbols to include in the import library.

   Caution: Although AIX shared libraries can be placed directly into an archive file ("lib<*x*>.a"), this is not recommended! A SOM class library should have a corresponding import library constructed directly from the combined export file.

   For OS/2:

   ```
   implib /noi abc.lib abc.def
   ```

   The first filename ("abc.lib") specifies the name for the import library and should always have a suffix of ".lib". The second filename ("abc.def") specifies the exported symbols to include in the import library. If you are using an export file that contains the symbol **SOMInitModule**, delete it first; **SOMInitModule** should not appear in your import library.

Chapter 3. Using SOM Classes in Client Programs

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

Some of the macros and functions described here are supplied as part of SOM's C and C++ *usage bindings*. These bindings are functions and macros defined in header files to be included in client programs. The usage bindings make it more convenient for C and C++ programmers to create and use instances of SOM classes.

To use the C or C++ bindings for a class, a client program must include a header file for the class (using the **#include** preprocessor directive). For a C language client program, the file <*classFileStem*>.h must be included. For a C++ language client program, the file <*classFileStem*>.xh must be included. The SOM Compiler generates these header files from an IDL interface definition. The header files contain definitions of the macros and functions that make up the C or C++ bindings for the class.

3.1 An Example Client Program

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

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

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

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

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

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

    return(0);
}
```

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

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

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

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

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

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

    return(0);
}
```

These client programs both produce the output:

```
Hello World Again
```

3.2 Using SOM Classes — the Basics

Declaring object variables

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

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

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

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

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

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

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

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

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

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

Note: If a C programmer also prefers to use explicit pointers to *<interfaceName>* types, then the SOM Compiler option **–maddstar** can be used when the C binding files are generated, and the explicit "*" will then be required in declarations of object variables. (This option is required for compatibility with existing SOM OIDL code. For information on using the **–maddstar** option, see "Running the SOM Compiler" in Chapter 2, "Implementing SOM Classes.")

Users of other programming languages must also define object variables to be pointers to the data structure used to represent SOM objects. The way this is done is programming-language dependent. The header file "somitype.h" defines the structure of SOM 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, SOM provides the *<className>***New** and the *<className>***Renew** macros for creating instances of a class.

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

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

Using *<className>* New

The *<className>***New** macro allocates enough space for a new instance of *<className>*, creates a new, initialized class instance, and returns a pointer to it. The *<className>***New** macro automatically creates the class object for *<className>*, as well as its ancestor classes and metaclass, if needed.

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

```
_somFree(obj);
```

This is important because storage for an object created using the *<className>***New** macro is allocated by the class of the object. Because only the class of an object created this way can know how to reclaim the object's storage, the **somFree** method is designed to call the class object for storage deallocation.

Using *<className>* Renew

The *<className>***Renew** macro is only used when the space for the object has been allocated previously. (Perhaps the space holds an old object that is not needed anymore.) This macro converts the given space into a new, initialized instance of *<className>* and returns a pointer to it. The argument of *<className>***Renew** must point to a block of storage large enough to hold an instance of class *<className>*. The SOM method **somGetInstanceSize** can be invoked on the class to determine the amount of memory required.

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

In addition, use of the<*className*>**Renew** macro requires that the class object has already been created (otherwise an error will result). The C and C++ usage bindings for a SOM class provide static linkage to a <*className*>**NewClass** procedure that can be used to create the class object.

For example, the following C code uses the function **HelloNewClass** to create the "Hello" class object. The arguments to this function are defined by the usage bindings, and indicate the version of the class implementation that is assumed by the bindings. (For more detail on creation of classes, see the later section, "Creating a class object.") Once the class object has been created, the example invokes the method **somGetInstanceSize** on this class to determine the size of a "Hello" object, uses **SOMMalloc** to allocate storage, and then uses the **HelloRenew** macro to create ten instances of the "Hello" class:

```
#include <hello.h>
main()
{
SOMClass helloCls; /*  A pointer for the Hello class object */
Hello objA[10];    /*  an array of Hello instances */
unsigned char *buffer;
int i;
int size;

/* create the Hello class object:  */
helloCls = HelloNewClass(Hello_MajorVersion, Hello_MinorVersion);

/* get the amount of space needed for a Hello instance:
 * (somGetInstanceSize is a method provided by SOM.) */
size = _somGetInstanceSize(helloCls);
size = ((size+3)/4)*4;  /* round up to doubleword multiple */

/* allocate the total space needed for ten instances: */
buffer = SOMMalloc(10*size);

/* convert the space into ten separate Hello instances: */
for (i=0; i<10; i++)
   objA[i] = HelloRenew(buffer+i*size);
...
...
/* Uninitialize the objects and free them */
for (i=0; i<10; i++)
  _somUninit(objA[i]);
SOMFree(buffer);
}
```

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

- For example, if an object was originally initialized using the <*className*>**New** macro, then, as discussed previously, the client should use the **somFree** method on it.
- On the other hand, if the program uses the **SOMMalloc** function to allocate memory, as illustrated in the example above, then the **SOMFree** function must be called to free the objects' storage (because **SOMFree** is the dual to **SOMMalloc**). Before doing so, however, the objects in the region to be freed should be deinitialized by invoking the **somUninit** method on them. This allows each object to free any memory that it may have allocated

without the programmer's knowledge. (The **somFree** method also calls the **somUninit** method.)

<u>For C++ programmers with usage bindings</u>, SOM recognizes the **new** operator, just as in standard C++, using the form:

new <*className*>

For example:

```
obj = new Hello;
```

The **new** operator allocates enough space for an instance of <*className*>, creates a new, initialized class instance, and returns a pointer to it. The **new** operator automatically creates the class object for <*className*>, as well as its ancestor classes and metaclass, if needed.

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

```
delete obj;
```

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

Invoking methods on objects

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

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

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

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

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

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

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

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

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

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

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

If the IDL specification of the method includes a *context* specification, then the method has an additional (implicit) **context** parameter. Thus, when invoking the method, <u>this argument must</u>

follow immediately after the **Environment** pointer argument. (None of the SOM-supplied methods require **context** arguments.) The **Environment** and **context** method parameters are prescribed by the CORBA standard.

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

If you use a C expression to compute the first argument to a method call (the receiver), you must use an expression without side effects, because the first argument is evaluated twice by the **_**<*methodName*> macro expansion. In particular, a **somNew** method call or a macro call of <*className*>**New** can *not* be used as the first argument to a C method call, because doing so would create two new class instances rather than one.

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

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

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

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

where “obj” has type “Bar” and “x” and “y” are the arguments to method “foo”. A programmer will usually know what type an object is expected to have. If this is not known, but the different methods have the same signature, the method can be invoked using name-lookup resolution, as described in the next section.

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

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

can be invoked by specifying a variable number of arguments, as in:

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

This is the short form of the invocation macro. Thus, the variable-arguments form illustrated above is only available in the absence of ambiguity. Alternatively, the long-form macro (which is always available) requires a **va_list**, as suggested by the method definition. Thus, to use a **va_list** argument, the method must be invoked as <*className*>_<*methodName*>, where <*className*> is the name of a class that supports the method.

For example, assuming that class “Vector” supports the “setMany” method above, the following C code constructs a variable of type **va_list** containing the arguments to “setMany” and invokes the “setMany” method with this variable:

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

Note: In most cases, a C programmer may use either the short or the long form of a method invocation macro interchangeably. However, *only* the long form complies with the CORBA standard for C usage bindings. If you wish to write code that can be easily ported to other vendor platforms that support the CORBA standard, use the long form exclusively. The long form is always available for every method that a class supports. The short form is provided both as a programming convenience and for source code compatibility with release 1 of SOM.

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

obj–><*methodName*> (*args*)

where *args* are the arguments to the method.

For instance, the following example invokes method “foo” on “obj”:

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

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

The **Environment** pointer is an implicit parameter; in the IDL definition of a method, the **Environment** pointer is *not* explicitly listed as a parameter to the method. For example, if a method is defined in IDL with two explicit parameters, as in:

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

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

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

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

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

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

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

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

can be invoked either by specifying a variable number of arguments, as in:

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

or by specifying a **va_list** in place of the variable number of arguments. To invoke a method using a **va_list** argument, the method must be invoked as <*className*>_<*methodName*>, where <*className*> is the name of the class that introduces the method. For example, assuming that class “Vector” defines the “setMany” method above, the following C++ code constructs a variable of type **va_list** containing the arguments to “setMany” and invokes the “setMany” method with this variable as an argument:

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

Using name-lookup method resolution

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

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

Name-lookup resolution is also used to invoke *dynamic methods* (that is, methods that have been added to a class’s interface at run time rather than being specified in the class’s IDL specification). For more information on name-lookup method resolution, see the topic “Method Resolution” in Chapter 2, “Implementing SOM Classes.”

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

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

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

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

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

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

- Use the **somResolveByName** procedure (described in the following section), or any of the methods **somLookupMethod, somFindMethod** or **somFindMethodOk** to obtain a pointer to the procedure that implements the desired method.
- Then, invoke the desired method by calling that procedure, passing the method's intended receiver, the **Environment** pointer (if needed), the **context** argument (if needed), and the remainder of the method's arguments, if any.

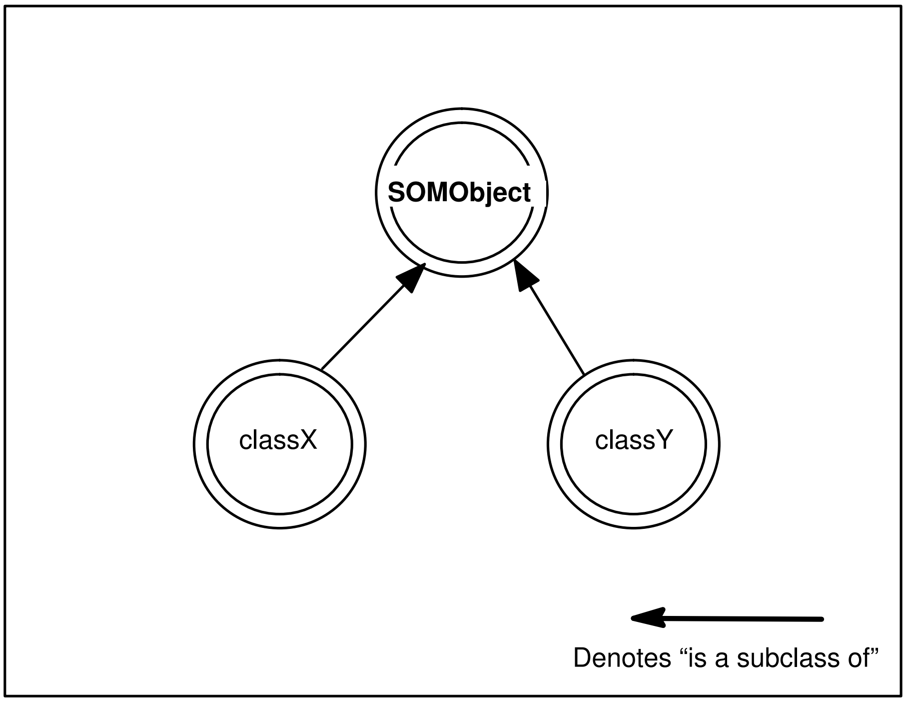
A name-lookup example

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

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

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

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



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

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

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

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

On the other hand, If the classes were designed in conjunction with each other, and the class designer felt that programmers might want to write generic code appropriate to either class of object, the **namelookup** modifier might have been used. This is a possibility in SOM release 2, even with multiple inheritance, but it is much more likely that the class designer would use multiple inheritance to introduce the reduce method in a sepatate class, and then use this other class as a parent for both classX and classY (thereby allowing the use of offset resolution).

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

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

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

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

Obtaining a method's procedure pointer

Method resolution is the process of obtaining a pointer to the procedure that implements a particular method for a particular object at run time. The method is then invoked subsequently by calling that procedure, passing the method's intended receiver, the **Environment** pointer (if needed), the **context** argument (if needed), and the method's other arguments, if any. C and C++ programmers may wish to obtain a pointer to a method's procedure for efficient repeated invocations.

Obtaining a pointer to a method's procedure is achieved in one of two ways, depending on whether the method is to be resolved using **offset** resolution or **name-lookup** resolution. Obtaining a method's procedure pointer via offset resolution is faster, but it requires that the name of the class that introduces the method and the name of the method be known at compile time. It also requires that the method be defined as part of that class's interface in the IDL specification of the class. (See the topic "Method Resolution" in Chapter 2, "Implementing SOM Classes," for more information on offset and name-lookup method resolution.)

Offset resolution

To obtain a pointer to a procedure using **offset** resolution, the C/C++ usage bindings provide the **SOM_Resolve** and **SOM_ResolveNoCheck** macros. The usage bindings themselves use the first of these, **SOM_Resolve**, for offset-resolution method calls. The difference in the two macros is that the **SOM_Resolve** macro performs consistency checking on its arguments, but the macro **SOM_ResolveNoCheck**, which is faster, does not. Both macros require the same arguments:

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

where the arguments are as follows:

receiver — The object to which the method will apply. It should be specified as an expression without side effects.

className — The name of the class that introduces the method.

methodName — The name of the desired method.

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

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

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

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

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

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

Name-lookup method resolution

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

Method name or signature not known at compile time

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

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

Dispatch-function method resolution

If the *signature* of the method is *not* known at compile time (and hence the method's argument list cannot be constructed until run time), then the method can be invoked at run time by (a) placing the arguments in a variable of type **va_list** at run time and (b) either using the

somGetMethodData method followed by use of the **somApply** function, or by invoking the **somDispatch** or **somClassDispatch** method. Using **somApply** is more efficient, since this is what the **somDispatch** method does, but it requires two steps instead of one. In either case, the result invokes a "stub" procedure called an *apply stub*, whose purpose is to remove the method arguments from the **va_list**, and then pass them to the appropriate method procedure in the way expected by that procedure. For more information on these methods and for examples of their use, see the **somApply** function, and the **somGetMethodData, somDispatch,** and **somClassDispatch** methods in the *SOMobjects Programmers Reference Manual*.

Accessing Attributes

In addition to methods, SOM objects can also have attributes. An *attribute* is an IDL shorthand for declaring methods, and does not necessarily indicate the presence of any particular instance data in an object of that type. Attribute methods are called "get" and "set" methods. For example, if a class "Hello" declares an attribute called "msg", then object variables of type "Hello" will support the methods **_get_msg** and **_set_msg** to access or set the value of the "msg" attribute. (Attributes that are declared as "readonly" have no "set" method, however.)

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

<u>For C</u>:

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

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

<u>For C++</u>:

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

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

Using class objects

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

Getting the class of an object

To get the class that an object is an instance of, SOM provides a method called **somGetClass**. The **somGetClass** method takes an object as its only argument and returns a pointer to the class object of which it is an instance. For example, the following statements store in "myClass" the class object of which "obj" is an instance.

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

Getting the class of an object is useful for obtaining information about the object; in some cases, such information cannot be obtained directly from the object, but only from its class. The section below entitled “Getting information about a class” describes the methods that can be invoked on a class object after it is obtained using **somGetClass**.

The **somGetClass** method can be overridden by a class to provide enhanced or alternative semantics for its objects. Because it is usually important to respect the intended semantics of a class of objects, the **somGetClass** method should normally be used to access the class of an object.

In a few special cases, it is not possible to make a method call on an object in order to determine its class. For such situations, SOM provides the **SOM_GetClass** macro. In general, the **somGetClass** method and the **SOM_GetClass** macro may have different behavior (if **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 (see the example under “Creating instances of a class” early in this chapter). As another example, a class object must be explicitly created when a program does not use the SOM bindings for a class. Without SOM bindings for a class, its instances must be created using **somNew** or **somRenew**, and these methods require that the class object be created in advance.

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

- When using the C/C++ language bindings for the class, and
- When the name of the class is known at compile time.

Using <*className*>NewClass

The <*className*>**NewClass** procedure initializes the SOM run-time environment, if necessary, creates the class object (unless it already exists), creates class objects for the ancestor classes and metaclass of the class, if necessary, and returns a pointer to the newly created class object. After its creation, the class object can be referenced in client code using the macro

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

or the expression

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

The <*className*>**NewClass** procedure takes two arguments, the major version number and minor version number of the class. These numbers are checked against the version numbers built into the class library to determine if the class is compatible with the client’s expectations. The class is compatible if it has the same major version number and the same or a higher minor version number. If the class is not compatible, an error is raised. Major version numbers usually only change when a significant enhancement or incompatible change is made to a class. Minor version numbers change when minor enhancements or fixes are made. Downward compatibility is usually maintained across changes in the minor version number. Zero can be used in place of version numbers to bypass version number checking.

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

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

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

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

Using somFindClass or somFindClsInFile

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

- First, initialize the SOM run-time environment by calling the **somEnvironmentNew** function (unless it is known that the SOM run-time environment has already been initialized).
- Then, use the **somFindClass** or **somFindClsInFile** method to create the class object. (The class must already be defined in a dynamically linked library, or DLL.)

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

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

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

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

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

major version number — The expected major version number of the class.

minor version number — The expected minor version number of the class.

The version numbers are checked against the version numbers built into the class library to determine if the class is compatible with the client's expectations.

The **somFindClass** method dynamically loads the DLL containing the class's implementation, if needed, creates the class object (unless it already exists) by invoking its <*className*>**NewClass** procedure, and returns a pointer to it. If the class could not be created,

somFindClass returns NULL. For example, the following C code fragment creates the class "Hello" and stores a pointer to it in "myClass":

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

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

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

Referring to class objects

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

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

_<*className*>

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

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

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

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

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

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

stores in “myClass” a pointer to the class object for class “Hello” (or NULL, if the class cannot be created).

Compiling and linking

This section describes how to compile and link C and C++ client programs. Compiling and linking a client program with a SOM class is done in one of two ways, depending on how the class is packaged.

If the class is not packaged as a library (that is, the client program has the implementation source code for the class, as in the examples given in the SOM IDL tutorial), then the client program can be compiled together with the class implementation file as follows. (This assumes that the client program and the class are both implemented in the same language, C or C++. If this is not the case, then each module must be compiled separately to produce an object file and the resulting object files linked together to form an executable.) In the examples below, the environment variable **SOMBASE** refers to the directory in which SOM has been installed. For client program “main” and class “Hello”:

Under AIX, for **C** programmers:

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

Under AIX, for **C++** programmers:

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

Under OS/2, for **C** programmers:

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

Under OS/2, for **C++** programmers:

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

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

Under AIX:

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

Under OS/2:

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

Both of these examples assume that the header files and the import library for the “Hello” class reside in the “include” and “lib” directories where SOM has been installed. If this is not the case, additional path information should be supplied for these files.

3.3 Language-neutral Methods and Functions

This section describes methods, functions, and macros that client programs can use regardless of the programming language in which they are written. In other words, these macros and functions 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.]

somPrintSelf — (method) Writes a brief description of an object, including its class and location in memory. The receiver of the method is the object to be printed.

somPrintf — (function) SOM's version of the C "printf" function. It generates character stream output via **SOMOutCharRoutine**. It has the same interface as the C "printf" function.

somVprintf — (function) Represents the "vprint" form of **somPrintf**. Its arguments are a formatting string and a **va_list** holding the remaining arguments.

somPrefixLevel — (function) Generates (via **somPrintf**) spaces to prefix a line at the indicated level. The return type is **void**. The argument is an integer specifying the level. The number of spaces generated is (2 * level).

somLPrintf — (function) Combines **somPrefixLevel** and **somPrintf.** The first argument is the level of the description (as for **somPrefixLevel**) and the remaining arguments are as for **somPrintf** (or for the C "printf" function).

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

Getting information about a class

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

somCheckVersion — Checks a class for compatibility with the specified major and minor version numbers. The receiver of the method is the SOM class about which information is needed. Additional arguments are values of the major and minor version numbers. The method returns TRUE if the class is compatible, or FALSE otherwise.

somClassFromId — Finds the class object of an existing class when given its **somId**, but without loading the class. The receiver of the method is the class manager (pointed to by the global variable **SOMClassMgrObject**). The additional argument is the class's **somId**. The method returns a pointer to the class (or NULL if the class does not exist).

somDescendedFrom — Tests whether one class is derived from another. The receiver of the method is the class to be tested, and the potential ancestor class is the argument. The method returns TRUE if the relationships exists, or FALSE otherwise.

somFindClass — Finds or creates the class object for a class, given the class's **somId** and its major and minor version numbers. The receiver of the method is the class manager (pointed to by the global variable **SOMClassMgrObject**). Additional arguments are the class's **somId** and the major and minor version numbers. The method returns a pointer to the class object, or NULL if the class could not be created.

somFindClsInFile — Finds or creates the class object for a class. This method is similar to **somFindClass**, except the user also provides the name of a file to be used for dynamic loading, if needed. The receiver of the method is the class manager (pointed to by the global variable **SOMClassMgrObject**). Additional arguments are the class's **somId**, the major and minor version numbers, and the file name. The method returns a pointer to the class object, or NULL if the class could not be created.

somGetInstancePartSize — Obtains the size of the instance variables introduced by a class. The receiver of the method is the class object. The method returns the amount of space, in bytes, needed for the instance variables.

somGetInstanceSize — Obtains the total size requirements for an instance of a class. The receiver of the method is the class object. The method returns the amount of space, in bytes, required for the instance variables introduced by the class itself and by all of its ancestor classes.

somGetName — Obtains the name of a class. The receiver of the method is the class object. The method returns the class name.

somGetNumMethods — Obtains the number of methods available for a class. The receiver of the method is the class object. The method returns the total number of currently available methods (static or otherwise, including inherited methods).

somGetNumStaticMethods — Obtains the number of static methods available for a class. (A static method is one declared in the class's interface specification [.idl] file.) The receiver of the method is the class object. The method returns the total number of available static methods, including inherited ones.

somGetParent — Obtains a pointer to a class's parent (base) class. The receiver of the method is the class object. The method returns a pointer to the class's parent class object (unless the receiver is **SOMObject**, for which it returns NULL).

somGetParents — Obtains a list of the parent (base) classes of a specified class. The receiver of the method is the class object. The method returns a pointer to a linked list of the parent (base) classes (unless the receiver is **SOMObject**, for which it returns NULL).

somGetVersionNumbers — Obtains the major and minor version numbers of a class. The return type is void, and the two arguments are pointers to locations in memory where the method can store the major and minor version numbers (of type **long**).

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

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

Getting information about an object

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

Methods

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

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

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

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

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

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

Functions

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

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

Chapter 4. Introduction to Distributed SOM (DSOM)

4.1 Introduction

What is Distributed SOM?

Whereas the power of SOM technology comes from the fact that SOM insulates the client of an object from the object's *implementation*, the power of DSOM lies in the fact that DSOM insulates the client of an object from the object's *location*.

Distributed SOM (or DSOM) provides a framework that allows application programs to access objects across address spaces. That is, application programs can access objects in other processes, even on different machines. Both the location and implementation of an object are hidden from a client, and the client accesses the object (via method calls) in the same manner regardless of its location.

DSOM currently supports two types of distribution: distribution among processes on the same machine — referred to as *Workstation DSOM* — and distribution among a network of machines — referred to as *Workgroup DSOM*. DSOM runs on the AIX (Release 3.2 and above) and OS/2 (Release 2.0 and above) operating systems, and Workgroup DSOM supports distribution across local area networks comprised of both OS/2 and AIX systems. Future releases of DSOM may support large, enterprise-wide networks.

DSOM can be viewed in two ways:

- As an extension to the System Object Model, that allows a program to invoke methods on SOM objects in other processes, and
- As an Object Request Broker (ORB), i.e., a standardized "transport" for distributed object interaction. In this respect, DSOM complies with the Common Object Request Broker Architecture (CORBA) specification, published by the Object Management Group (OMG) and x/Open™.

This chapter describes DSOM from both perspectives.

DSOM features

Here is a quick summary of some of DSOM's more important features:

- Uses the standard SOM Compiler, Interface Repository, language bindings, and class libraries. DSOM provides a growth path for non-distributed SOM applications.
- Allows an application program to access a mix of local and remote objects. The fact that an object is remote is transparent to the program.
- Provides run-time services for creating, destroying, identifying, locating, and dispatching methods on remote objects. These services can be overridden or augmented to suit the application.
- Uses existing interprocess communication (IPC) facilities for Workstation communication, and common local area network (LAN) transport facilities for Workgroup communications. Support for TCP/IP, Netware IPX/SPX, and Netbios is provided. DSOM communications is extensible in that an application can provide its own transport (see Appendix E of this User's Guide).
- Provides support for writing multi-threaded (on OS/2) and event-driven programs.

- Provides a default object server program, which can be easily used to create SOM objects and make those objects accessible to one or more client programs. If the default server program is used, SOM class libraries are loaded upon demand, so no server programming or compiling is necessary.
- Complies with the CORBA 1.1 specification, which is important for application portability.

When to use DSOM

DSOM should be used for those applications that require *sharing* of objects among multiple programs. The object actually exists in only one process (this process is known as the object's *server*); the other processes (known as *clients*) access the object via remote method invocations, made transparently by DSOM.

DSOM should also be used for applications that require objects to be *isolated* from the main program. This is usually done in cases where reliability is a concern — either to protect the object from failures in other parts of the application, or (less often), to protect the application from an object.

Note that some distributed applications may have special performance, reliability, or cooperative processing requirements, to which the SOM Replication framework is better suited. The Replication framework is oriented toward "groupware" (multi-party cooperative processing) applications, and has facilities for fault tolerance and recovery. The Replication framework is distinct from DSOM in that it maintains a complete replica of an object in each participant's address space, while DSOM establishes remote connections to shared objects. For more information, see Chapter 6, "Introduction to the Replication Framework".

4.2 A Simple DSOM Example

A sample "Stack" application is presented in this section as a tutorial introduction to DSOM. It demonstrates that, for simple examples like a "Stack", after very little work the class can be used to implement distributed objects that are accessed remotely. The example first presents the "Stack" application components and the steps that the implementor must perform before the application can be run, and then describes the run-time activity that results from executing the application. This run-time scenario introduces several of the key architectural components of the DSOM run-time environment.

The source code for this example is provided with the DSOM samples in the SOMobjects Developer Toolkit.

The "Stack" interface

The example starts with the assumption that the class implementor has successfully built a SOM class library DLL, called "stack.dll", in the manner described in Chapter 2. The DLL implements the following IDL interface.

```
interface Stack: SOMObject {
        const long stackSize = 100;
        boolean full();
        boolean empty();
        long top();
        void pop();
        void push(in long element);
        #ifdef __SOMIDL__
        implementation {
           releaseorder: full, empty, top, pop, push;
           dllname="stack.dll";          // dll name
           somInit: override;            // method modifier
           long stackimpl[stackSize];    // instance data
        };
        #endif
};
```

This DLL could have been built without the knowledge that it would ever be accessed remotely (that is, built following the procedures in Chapter 2, "Implementing Classes in SOM"). Note, however, that some DLLs may require changes in the way their classes pass arguments and manage memory, in order to be used by remote clients.

Client program using a local stack

A simple client program *written to use a local "Stack" object* is displayed below. This C program is shown so that the differences between a local and remote client program can be highlighted.

```
#include <stack.h>

int main(int argc, char *argv[]) {
        Stack stk;
        Environment e;

        SOM_InitEnvironment(&e);
        stk = StackNew();

        _push(stk,&e,100);
        _push(stk,&e,200);
        _pop(stk,&e);
        if (!_empty(stk,&e)) somPrintf("Top: %d\n", _top(stk,&e));

        _somFree(stk);
        SOM_UninitEnvironment(&e);

        return(0);
}
```

Client program using a remote stack

The preceding program has been rewritten below to use DSOM to create and access a "Stack" object somewhere in the system. The exact location of the object does not matter to the application — it just wants a "Stack" object. Note that the stack operations of the two programs are identical. The main differences lie in stack creation and destruction, as highlighted below.

```
#include <somd.h>
#include <stack.h>

int main(int argc, char *argv[]) {
        Stack stk;
        Environment e;

        SOM_InitEnvironment(&e);
        SOMD_Init(&e);

        StackNewClass(Stack_MajorVersion, Stack_MinorVersion);
        stk = _somdNewObject(SOMD_ObjectMgr, &e, "Stack", "");

        /* Note that the stack is accessed as if it is local */
        _push(stk,&e,100);
        _push(stk,&e,200);
        _pop(stk,&e);
        if (!_empty(stk,&e)) somPrintf("Top: %d\n", _top(stk,&e));

        _somdDestroyObject(SOMD_ObjectMgr, &e, stk);
        SOMD_Uninit(&e);
        SOM_UninitEnvironment(&e);

        return(0);
}
```

Let's step through the differences.

First, every DSOM program must include the file <somd.h> (or, when using C ++, <somd.xh>). This file defines constants, global variables, and run-time interfaces used by DSOM. Usually, this file is sufficient to establish all necessary DSOM definitions.

Next, DSOM requires its own initialization call.

```
        SOMD_Init(&e);
```

The call to **SOMD_Init** initializes the DSOM run-time environment. **SOMD_Init** must be called before any DSOM run-time calls are made. A side-effect of calling **SOMD_Init** is that a run-time object, called the *DSOM Object Manager,* is created and a pointer to it is stored in the global variable, **SOMD_ObjectMgr**, for programming convenience. The DSOM Object Manager provides basic run-time support for *clients* to find, create, destroy, and identify objects. The Object Manager is discussed in detail in the section entitled "Basic Client Programming."

Next, the local stack creation statement,

```
        stk = StackNew();
```

was replaced by

```
        StackNewClass(Stack_MajorVersion, Stack_MinorVersion);
        stk = _somdNewObject(SOMD_ObjectMgr, &e, "Stack", "");
```

The call to "StackNewClass" registers the "Stack" class with the SOM Class Manager. This call is necessary to define the "Stack" class and its interfaces to the SOM run time. In fact, in the original program "StackNewClass" is implicitly called by the "StackNew" procedure.

The call to **somdNewObject** asks the DSOM Object Manager (**SOMD_ObjectMgr**) to create a "Stack" object, wherever it can find an implementation of "Stack". (There are other methods with which one can request *specific* servers.) If no object could be created, NULL is returned and an exception is raised. Otherwise, the object returned is a "Stack" proxy.

A *proxy* is an object that is a local representative for a remote *target object*. A proxy inherits the target object's interface, so it responds to the same methods. Operations invoked on the proxy are not executed locally, but are forwarded to the "real" target object for execution. The client program always has a proxy for each remote target object on which it operates.

From this point on, the client program treats the "Stack" proxy exactly as it would treat a local "Stack". The "Stack" proxy takes responsibility for forwarding requests to and yielding results from the remote "Stack". For example,

```
_push(stk,&e,100);
```

causes a message representing the method call to be sent to the server process containing the remote object. The DSOM run time in the server process decodes the message and invokes the method on the target object. The result (in this case, just an indication of completion) is then returned to the client process in a message. The DSOM run time in the client process decodes the result message and returns any result data to the caller.

At the end of the original client program, the local "Stack" was destroyed by the statement,

```
_somFree(stk);
```

whereas in the client program above, the "Stack" proxy and the remote "Stack" are destroyed by the statement,

```
_somdDestroyObject(SOMD_ObjectMgr, &e, stk);
```

If the client only wants to release its use of the remote object (freeing the proxy) without destroying the remote object, it can call the **somdReleaseObject** method instead of **somdDestroyObject**.

Finally, the client must shut down DSOM, so that any operating system resources acquired by DSOM for communications or process management can be returned:

```
SOMD_Uninit(&e);
```

This call must be made at the end of every DSOM program.

Using specific servers

In DSOM, the process that manages a target object is called the object's *server.* Servers are implemented as programs that use SOM classes. Server implementations are registered with DSOM in an *Implementation Repository* — the Implementation Repository is a database queried by clients in order to find desired servers, and queried by DSOM in order to activate those servers upon demand.

The example above placed no constraints on the DSOM Object Manager as to where the remote "Stack" object should be created. The **somdNewObject** call creates a remote object of a specified class in an arbitrary server that implements that class. However, the DSOM Object Manager provides methods for finding specific servers.

For example, the client program above can be modified slightly to find a specific server named "StackServer", which has already been registered in DSOM's Implementation Repository. (Note that the programmer knew or discovered that the "StackServer" server implementation supports the "Stack" class.) The highlighted lines below show the changes that were made:

```
#include <somd.h>
#include <stack.h>

int main(int argc, char *argv[]) {
        Stack stk;
        Environment e;
        SOMDServer server;

        SOM_InitEnvironment(&e);
        SOMD_Init(&e);

        StackNewClass(Stack_MajorVersion, Stack_MinorVersion);
        server =
          _somdFindServerByName(SOMD_ObjectMgr, &e, "StackServer");
        stk = _somdCreateObj(server, &e, "Stack", "");

        _push(stk,&e,100);
        _push(stk,&e,200);
        _pop(stk,&e);
        if (!_empty(stk,&e)) somPrintf("Top: %d\n", _top(stk,&e));

        _somdDeleteObj(server, &e, stk);
        _somdReleaseObject(SOMD_ObjectMgr, &e, stk);
        _somdReleaseObject(SOMD_ObjectMgr, &e, server);
        SOMD_Uninit(&e);
        SOM_UninitEnvironment(&e);

        return(0);
}
```

This version of the program replaces the **somdNewObject** operation with calls to **somdFindServerByName** and **somdCreateObj**. The **somdFindServerByName** method consults the Implementation Repository to find the DSOM server implementation whose name is "StackServer", and creates a *server proxy,* which provides a connection to that server. Every DSOM server process has a *server object* that defines methods to assist in the creation and management of objects in that server. Server objects must be instances of **SOMDServer** or one of its subclasses. The **somdFindServerByName** returns a proxy to the **SOMDServer** object in the named server.

Once the client has the server proxy, it can create and destroy objects in that server. The **somdCreateObj** call creates an object of the class "Stack" in the server named "StackServer".

To free the remote "Stack" object, the example shows a **somdDeleteObj** request on the stack object's server. Next, **somdReleaseObject** requests are made on the DSOM Object Manager, to free the stack proxy and the server proxy in the client. (Note that these three calls are equivalent to the **somdDestroyObject** call in the previous example.)

A note on finding existing objects

The two examples above show how a remote, transient object can be created by a client, for its exclusive use. It is also likely that clients will want to find and use objects that are already in existence. In that case, the calls to **somdNewObject** or **somdCreateObj** would be replaced with other "lookup" calls on some directory object that would take an object name or identifier and return a proxy to the remote object.

Such a directory object could be implemented by the application as a persistent SOM object, using DSOM to share it among processes.

"Stack" server implementation

A server consists of three parts. First, a "main" program, when run, provides an address space for the objects it manages, and one or more process "threads" that can execute method calls.

(AIX currently does not have multi-thread support, while OS/2 does.) Second, a server object, derived from the **SOMDServer** class, provides methods used to manage objects in the server process. Third, one or more class libraries provide object implementations. Usually these libraries are constructed as *dynamically linked libraries* (DLLs), so they can be loaded and linked by a server program dynamically.

In this simple example, we can use the default DSOM server program, which is already compiled and linked. The default server behaves as a simple server, in that it simply receives requests and executes them, continuously. The default server creates its server object from the class, **SOMDServer**. The default server will load any class libraries it needs upon demand.

The "Stack" class library, "stack.dll", can be used without modification in the distributed application. This is possible because the "Stack" class is "well formed" — *there are no methods that implicitly assume the client and the object are in the same address space*.

Thus, by using the default server and the existing class library, a simple "Stack" server can be provided without any additional programming!

An application may require more functionality in the server program or the server object than the default implementations provide. A discussion on how to implement server programs and server objects is found in the SOMobjects *Users Guide.*

Compiling the application

DSOM programs and class libraries are compiled and linked like any other SOM program or library. The header file "somd.h" (or for C++, "somd.xh") should be included in any source program that uses DSOM services. DSOM run-time calls can be resolved by linking with the SOMobjects Toolkit library: "libsomtk.a" on AIX and "somtk.lib" on OS/2. (The DSOM DLL, "somd.dll", will be loaded at run time.)

For more information, see "Compiling and linking" in Chapter 3, "Using SOM Classes in Client Programs," and the same topic in Chapter 2, "Implementing Classes in SOM."

Installing the implementation

Before the application can be run, certain environment variables must be set and the stack class and server implementations must be registered in the SOM Interface Repository and DSOM Implementation Repository. Registration with the Interface and Implementation Repositoriesmust be performed on *every* machine that has a copy of these repositories. (If the IR and Implementation Repository are in a shared file system, however, this need only be done once.)

Setting environment variables

Several environment variables are used by SOM and DSOM. These variables need to be set before registering the DSOM application in the Interface and Implementation Repositories.

For this example, the following environment variables could be set as shown. A full description of the environment variables and how to set them is given in the DSOM chapter of the SOMobjects *Users Guide.*

On AIX (in the syntax of the default shell, /bin/ksh):

```
export HOSTNAME=machine3
export SOMIR=$SOMBASE/etc/som.ir:/u/shared/my.ir
export SOMDDIR=/u/shared/somddir
export SOMSOCKETS=TCPIPSockets
export LIBPATH=$LIBPATH:$SOMBASE/lib:/u/shared/lib
export MALLOCTYPE=3.1
```

On OS/2:

```
set USER=pat
set HOSTNAME=machine3
set SOMDDIR=c:\somddir
set SOMSOCKETS=TCPIPSockets

rem *** The following variables are set in CONFIG.SYS by
rem *** the install program on OS/2, assuming "c:\som" is the
rem *** value of %SOMBASE% supplied by the user.
set SOMIR=c:\som\etc\som.ir;c:\som.ir
set LIBPATH=.;c:\som\lib;<previous LIBPATH>
```

USER identifies the user of a DSOM client application. DSOM sends the USER ID with every remote method call, in case the remote object wishes to perform any access-control checking. This is discussed later in the section "Basic Server Programming." (Note that USER is usually set automatically by AIX when a user logs in.)

HOSTNAME identifies the name of each machine running DSOM.

SOMIR gives a list of files that together constitute the Interface Repository. The IR is used by DSOM to guide the construction and interpretation of request messages. For DSOM, it is preferable to use *full* pathnames in the list of IR files, since the IR will be shared by several programs that may not all reside in the same directory.

SOMDDIR gives the name of a directory used to store DSOM configuration files, including the Implementation Repository.

SOMSOCKETS identifies the name of a class that implements communications services for DSOM and other SOM frameworks. (The **TCPIPSockets** class comes with the SOMobjects Toolkit; Netbios, Netware, and TCP/IP support are packaged with Workgroup DSOM.)

LIBPATH gives a list of directories where DLLs can be found.

MALLOCTYPE is defined by AIX; it forces the standard C library routines to use the same memory allocation algorithm as in AIX 3.1. (DSOM occasionally experiences problems using the AIX 3.2 algorithm.)

Registering the class in the Interface Repository

Before an object can be accessed remotely by DSOM, it is necessary to register the class's interface and implementation in the Interface Repository (IR). DSOM uses the interface information when transforming local method calls on proxies into request messages transmitted to remote objects.

DSOM servers also consult the IR to find the name of the DLL for a dynamically loaded class. The DLL name for the "Stack" class must be specified using the **dllname="stack.dll"** modifier in the **implementation** statement of the "Stack" IDL. The Interface Repository is described in detail in Chapter 7, "The Interface Repository Framework," of the SOMobjects *Users Guide.*

The IDL specification of "Stack" is compiled into the Interface Repository using the following command:

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

When a class has not been compiled into the Interface Repository, DSOM will generate a run-time error when an attempt is made to invoke a method from that class. The error indicates that the method's descriptor was not found in the IR.

Registering the server in the Implementation Repository

It is necessary to register a description of a server's implementation in the Implementation Repository. DSOM uses this information to assist clients in finding servers, and in activating server processes upon demand.

For this example, where the default server is used, we need only to identify the server's name, and the class that the server implements. This is accomplished using the **regimpl** utility discussed in the DSOM chapter of the SOMobjects *Users Guide*. The following commands define a default server, named "StackServer", which supports the Stack class:

```
regimpl –A –i StackServer
regimpl –a –i StackServer –c Stack
```

Running the application

Starting the DSOM daemon

Before running a DSOM application, the DSOM daemon, **somdd,** must be started on each (server) machine. (Client-only machines do not require an active DSOM daemon.) The daemon can be started manually from the command line, or it could be started automatically from a start-up script run at boot time. It may be run in the background with the commands **somdd&** on AIX and **start somdd** on OS/2. (The **somdd** program takes no parameters.)

The **somdd** daemon is responsible for establishing a "binding" (i.e., a connection) between a client process and a server. It will activate the desired server automatically, if necessary.

Running the client

Once the DSOM daemon is running on every machine, the application may be started. This is accomplished by running the client program. If the StackServer is not running, it will be started automatically by the DSOM daemon when the client attempts to invoke a method on one of its objects.

"Stack" example run–time scenario

The following scenario steps through the actions taken by the DSOM run time in response to each line of code in the second "Stack" client program presented above. The illustration following the scenario is an illustration of the processes, and the objects within them, that participate in these actions.

- Initialize an environment for error passing:

  ```
  SOM_InitEnvironment(&e);
  ```

- Initialize DSOM:

  ```
  SOMD_Init(&e);
  ```

  This causes the creation of the DSOM Object Manager (with **SOMDObjectMgr** interface). The global variable **SOMD_ObjectMgr** points to this object.

- Initialize "Stack" class object:

  ```
  StackNewClass(Stack_MajorVersion, Stack_MinorVersion);
  ```

- Find the "StackServer" implementation and assign its proxy to the variable `server`:

  ```
  server = _somdFindServerByName(SOMD_ObjectMgr, &e, ”StackServer”);
  ```

  This causes the creation of the server proxy object in the client process. Proxy objects are shown as shaded circles. Note that the "real" server object in the server process is not created at this time. In fact, the server process has not yet been started.

- Ask the server object to create a “Stack” and assign “Stack” proxy to variable `stk`:

```
stk = _somdCreateObj(server, &e, ”Stack”, ””);
```

  This causes **somdd**, the DSOM daemon (already running on the server machine), to activate the stack server process (by starting the “generic” server program). The stack server process, upon activation, creates the “real” **SOMDServer** object in the server process. The **SOMDServer** object works with the DSOM run time to create a local “Stack” object and return a “Stack” proxy to the client.

- Ask the “Stack” proxy to push 100 onto the remote stack:

```
_push(stk,&e,100);
```

  This causes a message representing the method call to be marshalled and sent to the server process. In the server process, DSOM demarshals the message and, with the help of the **SOMDServer**, locates the target “Stack” object upon which it invokes the method (**“**push”). The result (which is void in this case) is then passed back to the client process in a message.

- Invoke more “Stack” operations on the remote stack, via the proxy:

```
_push(stk,&e,200);
_pop(stk,&e);
if (!_empty(stk,&e)) t = _top(stk,&e);
```

- Explicitly destroy both the remote stack, the stack proxy, and the server proxy:

```
_somdDeleteObj(server, &e, stk);
_somdReleaseObject(SOMD_ObjectMgr, &e, stk);
_somdReleaseObject(SOMD_ObjectMgr, &e, server);
```

- Free the error-passing environment:

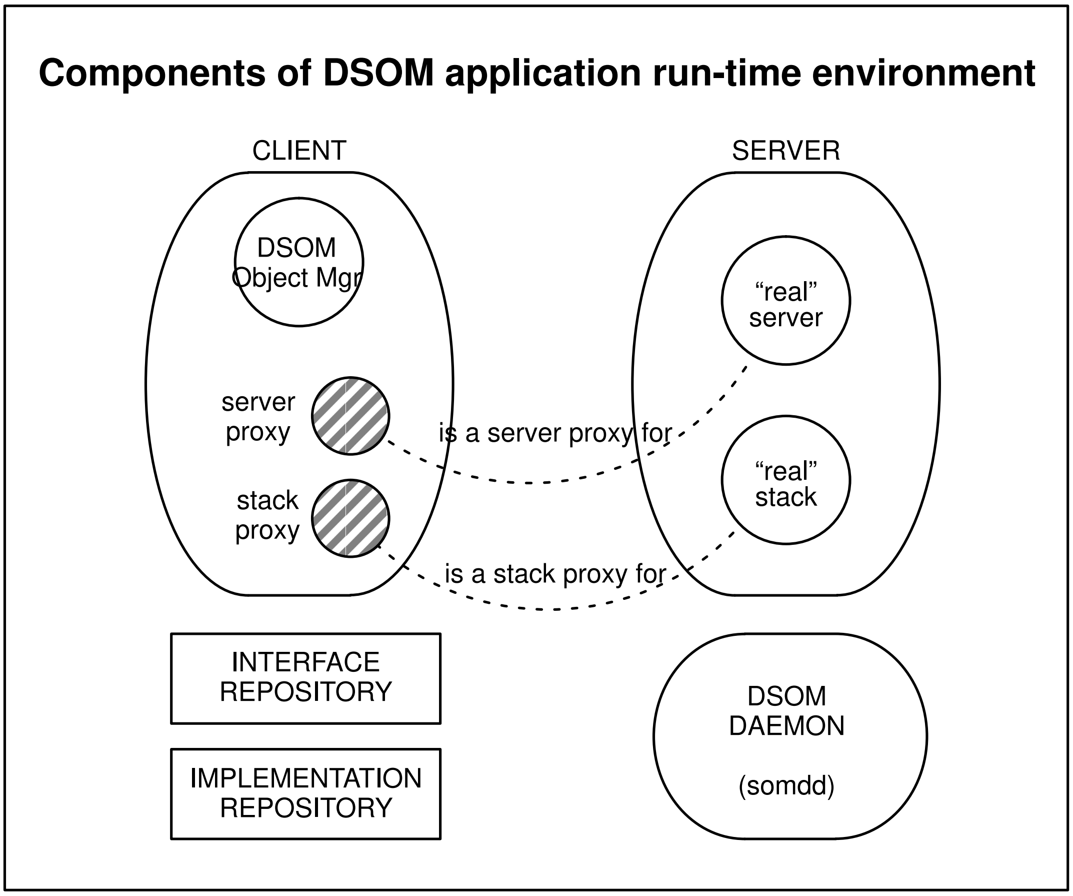
```
SOM_UninitEnvironment(&e);
```

This scenario has introduced the key processes in a DSOM application: client, server, and **somdd**. Also introduced are the key objects that comprise the DSOM run-time environment: the **SOMD_ObjectMgr** in the client process and the **SOMD_ServerObject** in the server process.

Summary

This example has introduced the key concepts of building, installing, and running a DSOM application. It has also introduced some (though not all) of the key components that comprise the DSOM application run-time environment, as pictured below.

The following sections, “Basic Client Programming”, “Basic Server Programming” , and “Implementing Classes” provide more detail on how to use, manage, and implement remote objects, respectively.



Chapter 5. Introduction to the Persistence Framework

5.1 Introduction

This chapter and the programs it describes demonstrate the use of the Persistence Framework of the SOMobjects Toolkit. It is assumed that the reader is proficient in the C programming language and has a working knowledge of Object-Oriented Programming (OOP) with SOM.

The SOMobjects Persistence Framework allows SOM objects to be saved in a persistent state and then later restored. The term “persistent” as used here means that an object’s state can be preserved beyond the termination of the process that creates it. Objects can be stored by themselves or grouped with other objects. Objects can be stored in default formats or in specially designed formats. Objects can be stored in the file system or in more specialized repositories. The examples in this chapter assume that objects are being stored in the file system by the default classes of the framework.

Functionally, the Persistence Framework replaces the file-system interface of a programming language. The Persistence Framework is not, however, designed to compete with a database management system. It does not provide recovery, transactions, or multi-user record locking. These should be handled at a higher level.

Several of the classes of the Persistence Framework can be subclassed to customize how and where objects are stored. This chapter deals primarily with the default framework classes. For a detailed discussion of subclassing the framework to change its default behavior, refer to Appendix D, “Subclassing the Persistence Framework.”

5.2 The Telephone-Directory Application

This chapter uses example programs to demonstrate use of the SOMobjects Persistence Framework. Each example is a variation on a telephone-directory application. The telephone-directory application creates a list of phone numbers, stores the list, recalls the list, and displays its contents. Examples use the Persistence Framework to store and restore the list.

The telephone-directory application includes SOM classes: A directory-entry class (“dirEntry”) and a phone-directory class (“phoneDir”). An object of class “phoneDir” encapsulates the phone directory so that individual entires can be added to or deleted from the phone directory.

5.3 Persistent Objects

A persistent object is one whose state can be preserved beyond the termination of the process that creates it. An object is potentially persistent if its class is derived (either directly or indirectly) from the **SOMPPersistentObject** class, either through single or multiple inheritance. If through single inheritance, the class of persistent objects (such as “phoneDir”) is usually derived directly from **SOMPPersistentObject**. If through multiple inheritance, typically a nonpersistent version of the class already exists, and then the persistent version of the class is derived from both the nonpersistent version and **SOMPPersistentObject.**

The following definition of the “phoneDir” class from the telephone directory application demonstrates the single inheritance case. Bold type is used to highlight the differences between a nonpersistent version of “phoneDir” and the persistent version of “phoneDir”.

Example 1: Single inheritance definition of persistent “phoneDir”

```
#include <po.idl>

interface dirEntry;

interface phoneDir : SOMPPersistentObject
{
   const unsigned long MAXDIRSIZE = 16;
   attribute sequence<dirEntry, MAXDIRSIZE> directory;

   long addEntry(in dirEntry entry);
// Adds a dirEntry object to the directory.

   void printDirInfo();
// Invokes lsEntry for each instance of direntry.

   dirEntry getEntry(in string entryID);
// getEntry returns a pointer to the list entry with name == entryID.
// If a matching entry is not found, it returns NULL.

   dirEntry dropEntry(in string entryID);
// dropEntry removes the entry with name == entryID from the directory
// array, and returns a pointer to the dropped entry.
// If a matching entry is not found, it returns NULL.
// The client application must use sompDeleteObject on the returned object
// to delete it from the IO Group.  The client must also free the dirEntry
// Object returned.

   #ifdef __SOMIDL__

   implementation
   {
      callstyle=oidl;

      releaseorder: addEntry, printDirInfo, getEntry, dropEntry,
                    _get_directory, _set_directory;

   // Class Modifiers
      filestem = phonedir;

      passthru C_h =  ”#include <direntry.h>”;

   // Attribute Modifiers
      directory: persistent;

   // Method Modifiers
      somInit: override;
      somUninit: override;
      sompIsDirty: override;
   };
   #endif /* __SOMIDL__ */

};
```

To make the object persistent we have derived it from **SOMPPersistentObject**. The **SOMPPersistentObject** class is defined in the po.idl file which is included at the beginning of the file.

Observe that Example 1 includes the following new statement, which registers the “directory” attribute as a persistent attribute:

```
directory: persistent;
```

In order to use the default format (class **SOMPAttrEncoderDecoder**) for writing persistent object data to a file, persistent object classes must fulfill the following design criteria:

- All persistent data must be one of these standard CORBA types:
  short, long, unsigned short, unsigned long, float, double, boolean, char, octet, string, sequence, structure, union, enum, array, union, or a persistent object.
- All persistent data must be declared as an **attribute**, and each of these attributes must be signified by the attribute modifier **persistent**. The **persistent** modifier for an attribute may appear in a derived class.
- Your persistent object interface definitions and those which they are derived from, if they contain persistent data, must be put in the SOM Interface Repository. This is done by compiling your .idl files with the –u flag, for example:

```
sc –sir –u myclass.idl
```

Finally, observe that the **SOMPPersistentObject** method **sompIsDirty** has been overridden. The **sompIsDirty** method is responsible for reporting whether the object is “dirty”. That is, that the object’s persistent data has been changed. By default, the **sompIsDirty** method always returns TRUE.

The implementation of the persistent “phoneDir” class is, for the most part, unchanged from the nonpersistent “phoneDir” class, except for the addition of the lines shown in bold in the following methods and the addition of the **sompIsDirty** implementation. Also take note of the additions to **somInit** and **somUninit**. These methods must pass through to their parent implementations for the Persistence Framework to initialize your object correctly.

Implementation of the persistent telephone directory

The following highlights the modifications to the “addEntry” method:

```
SOM_Scope long  SOMLINK addEntry(phoneDir somSelf, dirEntry entry)
{
    Environment *ev;
    long rc;

    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug(”phoneDir”,”addEntry”);

    ev = SOM_CreateLocalEnvironment();
    if (sequenceLength(_directory) < sequenceMaximum(_directory)) {
        sequenceElement(_directory, sequenceLength(_directory)) = entry;
        sequenceLength(_directory)++;
        _sompSetDirty (somSelf, ev);
        rc = 0L;
    } else
        rc = –1L;
    SOM_DestroyLocalEnvironment(ev);
    return(rc);
}
```

The modification of the “dropEntry” method is similar to the “addEntry” method:

```
SOM_Scope dirEntry  SOMLINK dropEntry(phoneDir somSelf, string entryID)
{
    int probe;
    char *probename;
    dirEntry dptr;
    Environment *ev;

    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","dropEntry");

    ev = SOM_CreateLocalEnvironment();
    for (probe = 0;probe < sequenceLength(_directory);probe++) {
        probename = dirEntry__get_name(sequenceElement(_directory,
                                       probe));
        if (strcmp(probename, entryID) == 0) {
            dptr = sequenceElement(_directory, probe);
            sequenceLength(_directory)--;
            for (; probe < sequenceLength(_directory); probe++) {
                sequenceElement(_directory, probe) =
                sequenceElement(_directory, (probe + 1));
            }
            _sompSetDirty (somSelf, ev);
            SOM_DestroyLocalEnvironment(ev);
            return dptr;
        } /* endif */
    } /* endfor */

    SOM_DestroyLocalEnvironment(ev);
    return (dirEntry) NULL;
}
```

Dirty Objects: Observe that the implementation of the persistent “phoneDir” class has the following line added to each method that modifies a persistent object:

```
    _sompSetDirty (somSelf, env);
```

This line invokes the **sompSetDirty** method. The Persistence Framework can be optimized to write only “dirty” objects (objects that have been modified since they were last written) whenever possible. Therefore, any method that updates the persistent data of an object should use the **sompSetDirty** method to mark the object as dirty. Clean objects may also be stored at the discretion of the Persistence Framework.

Recall that **sompIsDirty** by default always returns TRUE. In the implementation of the **sompIsDirty** method below, **sompIsDirty** returns the result of **sompGetDirty**. **sompGetDirty** reports TRUE if **sompSetDirty** has been invoked on the object.

```
SOM_Scope boolean  SOMLINK sompIsDirty(phoneDir somSelf,
Environment *ev)
{
    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","sompIsDirty");

    return (_sompGetDirty(somSelf, ev));
}
```

Below, the additions to **somInit** and **somUninit** are shown in bold. These additions must be made to your persistent object if your object overrides **somInit** and **somUninit**. These methods perform initialization and cleanup that must be completed for the framework to work correctly.

```
SOM_Scope void  SOMLINK somInit(phoneDir somSelf)
{
    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","somInit");

    phoneDir_parent_SOMPPersistentObject_somInit(somSelf);
    sequenceMaximum(_directory) = MAXDIRSIZE;
    sequenceLength(_directory) = 0;
    _directory._buffer =
        SOMMalloc(sizeof (dirEntry) * MAXDIRSIZE);
}

SOM_Scope void  SOMLINK somUninit(phoneDir somSelf)
{
    phoneDirData *somThis = phoneDirGetData(somSelf);
    phoneDirMethodDebug("phoneDir","somUninit");

    if (_directory._buffer) SOMFree(_directory._buffer);
    phoneDir_parent_SOMPPersistentObject_somUninit(somSelf);
}
```

Example 2: Definition of persistent “dirEntry”

Revising “dirEntry”: The following code shows the definition of the persistent “dirEntry” class, with differences from the nonpersistent version shown in bold:

```
#include <po.idl>

interface dirEntry : SOMPPersistentObject
{
   attribute string name;
   attribute string phone;

   void mkEntry(in string name, in string phone_no);
// Initialize new entry.

   void lsEntry();
// lsEntry displays the entry on a single line.

   #ifdef __SOMIDL__
   implementation
   {
      callstyle=oidl;
      // no Environment on methods

      releaseorder: mkEntry, lsEntry,
                    _get_name, _set_name,
                    _get_phone, _set_phone;

      // Class modifiers
      filestem = direntry;

      // Attribute modifiers
      name: persistent;
      phone: persistent;

      // Method modifiers
      somInit: override;
      somUninit: override;
      sompIsDirty: override;

   };
   #endif /* __SOMIDL__ */
};
```

These changes are analogous to the changes in the “phoneDir” class. The *implementation* of the persistent “dirEntry” class is the same as for the nonpersistent “dirEntry” and includes the addition of the **sompIsDirty** implementation. One might expect to initialize the persistent objects as dirty in their **somInit** methods; however, this is unnecessary because initializing an object with a persistent ID automatically sets its state to dirty. Persistent-object initialization is discussed in a later section.

The next version of the “phoneDir” class demonstrates the multiple inheritance case.

Chapter 6. Introduction to the Replication Framework

6.1 Introduction

The continuing integration of electronic computation and electronic communication has generated a new kind of application. Whether called “a multiparty application,” “groupware,” or “computer-supported cooperative work,” this kind of application includes a dimension well beyond that of traditional distributed computing. Rather than merely placing a user in contact with a remote database (as in transaction processing — a common experience when people use an automatic teller machine), this new kind of application places a person in direct contact with other people.

Currently, the most common multiparty applications consist of two-player games where the players are located in the same room using one computer. In the future, however, multiparty applications will commonly coordinate more than two people who are not necessarily located together. For example, there are a number of group editors in the computer science literature. In addition, decision support systems for meeting rooms are beginning to emerge.

An essential aspect of successful multiparty applications is that participants must really feel they are touching one another. To facilitate the implementation of an application that meets this requirement, the SOMobjects Toolkit provides the **Replication Framework**.

The Replication Framework enables an object to be replicated (copied) in the address spaces of several processes distributed about a network. This provides the fast response times necessary for cooperative interactive applications. Each *replica* of the object can be updated, and the framework guarantees that the updates are serialized. (That is, there is a specific order in which updates will be performed, despite the fact that the updates originate from different processes.)

6.2 Principles of the Replication Framework

Figure 1 depicts a simple but typical situation using the Replication Framework. Notice that there are multiple processes running on possibly different nodes of the network, and each process has a copy of a replicated object. Here, the object *Lassie* is an instance of the class “ReplicatedDog”, which is a subclass of “Dog”. “ReplicatedDog” obtains its ability to replicate objects by being a subclass of the SOM-provided class **SOMRReplicbl**.

The Replication Framework provides the following properties:

1. The number of replicas and their location in the network is hidden from the application program.
2. Updates (changes in value for instance variables) are communicated from the originating process to the other participants without the use of secondary storage.
3. The updates are serialized by the framework.
4. A participant is free at any time to join or leave the group of processes holding a replica.
5. The framework is tolerant of a single failure resulting from either a process crash or a network partition.

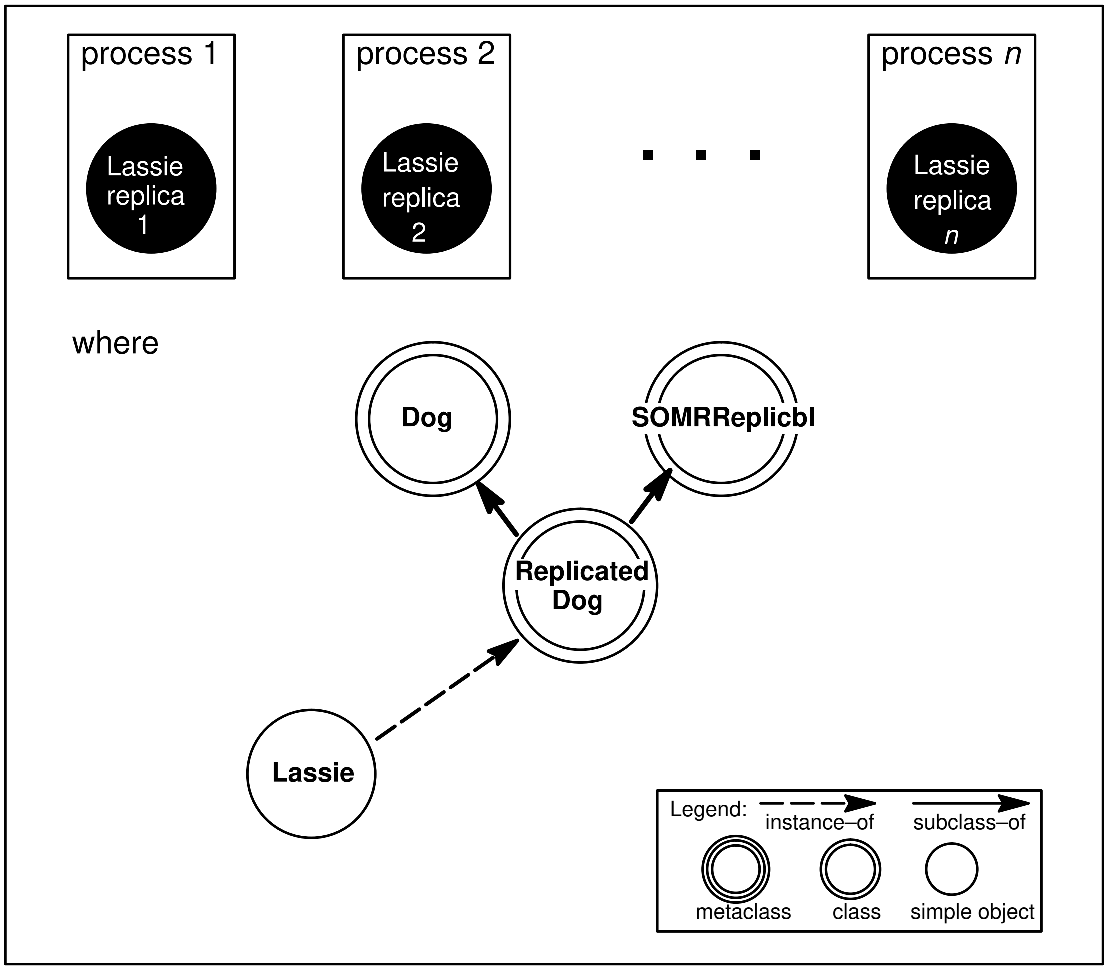


Figure 1. Multiple processes running replicas of a simple object.

Steps in using Replication

The Replication Framework can be exploited only if the applications are structured appropriately. The required structure is similar to the “Model–View–Controller” paradigm used by Smalltalk programmers. The Replication Framework proposes a View–Data paradigm. The Data object has whatever ‘state’ information the application desires to store in it. The View object has no state. It has methods to show a rendition of the state contained in the Data object. It may have some data that purely pertains to the image being displayed to the user. For example, in a visual presentation, the colors used for different regions may be in the View object, but the content information comes from the Data object.

Further, the View and Data must have a protocol between them such that whenever the Data object changes, a “signal” is sent to the View object to take note of the change and refresh the display (if necessary). This protocol can be extended to multiple Views on the same Data object, whereby an update to the Data object is automatically seen in all visual presentations. Effectively, the Views “observe” the Data.

The Replication Framework is concerned with Data objects only. It is the responsibility of the application developer to implement the “observation” protocol between the Views and Data. The Replication Framework requires the Data objects to be derived from a distinguished framework class **SOMRReplicbl.**

The recommended steps in building a replicated application are as follows:

- Structure the application to follow the View–Data paradigm, build the View and Data as **SOMObject**s, build a driver main program, and test the application as a stand-alone program.
- Convert the Data object (it is possible to have multiple replicated objects in a given program) to be a replicated object by making it a subclass of **SOMRReplicbl**. Do all the necessary method overrides to be a subclass of **SOMRReplicbl**. Again, test it as a stand-alone program.
- Enable replication by doing the appropriate replication initialization in the application's main program, and test it with multiple copies of the application running.

6.3 Components of the Framework

The Replication Framework consists of three components, the:

- **SOMRReplicbl** class,
- **SOMR** class, and
- **.scf** file.

The **SOMRReplicbl** class is used to make subclasses that can create replicated objects. In Figure 1, for example, the class "ReplicatedDog" (a subclass of "Dog") obtains its ability to replicate the object "Lassie" by also being a subclass of **SOMRReplicbl**. Techniques for making a "replicated" class (that is, a class that can create replicated objects) are discussed in the subsequent paragraphs.

The **SOMR** class provides a number of services for replicated objects. It is not necessary to know what they are; it is only necessary to create one instance of **SOMR** during the initialization of your program. [This is one of only two requirements imposed by the Replication Framework on the main program. The other one is the use of the Event Management Framework that is included with SOMobjects.]

All replicated objects have names; it is this name by which the replicas locate each other on the network. That is, all objects initialized for replication under the same name (given during object initialization) are replicas of each other. Note that replicas must be instances of the same class, and that the name of the object is a null-terminated string.

Like all distributed systems, the Replication Framework needs a way to establish communication among the various participating processes. This is accomplished with the *.scf* files. (*scf* stands for share control file.) By convention, the *.scf* file corresponding to a set of replicas is named by appending ".scf" to the object's name. Because the object's name is just a string, the name may represent a path in the file system. If the name is a path, all the directories in the path must exist.

For example, in Figure 1 the processes initially locate each other by reading the "Lassie.scf" file. The Replication Framework creates such files automatically in the distributed file system. (Note: This is not a contradiction to the earlier statement that updates are communicated without the use of secondary storage. The *.scf* file is only used by replicas to establish communication channels among them. Once they are established, the updates are propagated on these channels without the help of secondary storage.)

Important: The *.scf* files are an artifact of the implementation that are visible to the user but are *not* part of the interface. As such, IBM reserves the right to eliminate them in the future. Do *not* make any programming decisions based on their content or even on their existence.

There are two ways in which changes can be propagated among replicas: *operation logging* and *value logging*:

- In operation logging, each method invocation that modifies a replica also executes at the site of the other replicas.
- In value logging, the change in value of a replica is encoded after a method invocation, and subsequently the values of the other replicas are upgraded to reflect this change.

The following two topics give examples for subclassing an existing class to create a “replicated” class (a class capable of creating replicated objects) that uses either operation logging or value logging.

6.4 Making a “replicated” class that provides operation logging

Following are the steps used to create a “replicated” class (for example, to create the class “ReplicatedDog” from the parent class “Dog”) whose replicated instances will be updated using operation logging. This example shows the C implementation; the C++ implementation is done similarly.

Step 1. Create a new class whose parents are both **SOMRReplicbl** and the original class (here, “Dog”) whose instances are to be replicated. Alternatively, you can directly derive “ReplicatedDog” from **SOMRReplicbl**.

Step 2. In the new class, override all the methods that change the value of instance variables (of “Dog”), override **somrDoDirective**, **somrGetState** and **somrSetState**. What to do in each overriding method is explained in the following steps.

Step 3. Each overriding method from the original class (here, “Dog”) should be coded as follows:

```
#include <somrerrd.h>

dogMethod( ReplicatedDog somSelf, <parameters> ) {
ReplicatedDogData *somThis = ReplicatedDogGetData(somSelf );
Environment *Env = SOM_CreateLocalEnvironment();
_somrLockNlogOp( somSelf,
                  Env,
                 ”ReplicatedDog”,
                 ”dogMethod”,
                 <parameters> );
if (Env->_major == NO_EXCEPTION ) {
      parent_dogMethod( somSelf, <parameters> );
      ... <any other additional code> ...
      _somrReleaseNPropagateOperation( somSelf, Env );
      }
else {
      /* code to handle failure to obtain a lock */
      switch (somriGetErrorCode(Env)){
      case SOMR_MASTER_UNREACHABLE:  ...
      case SOMR_UNAUTHORIZED:  ...
      case SOMR_TIMEOUT:  ...
      case SOMR_TRYLATER  ...
      default:  ...
      }
}
```

where `<parameters>` represents the explicit formal parameters of “dogMethod”. Note that for IDL call style the environment is listed among the <parameters> to

somrLockNLogOP. The parameters to **somrLockNlogOp** must be one of the following types:

- one of the following primitive types: short, long, unsigned short, unsigned long, float, double, char, octet, boolean, and string.
- a pointer to a primitive type, or
- an object pointer.

If an object parameter is used, it must be an instance of one of the following classes:

- **SOMRReplicbl**
- **SOMPPersistent**
- **SOMRLinearizable**

There is an essential difficulty for the Replication Framework when using operation logging with methods that have objects as parameters — that is, the problem of how to repeat the operation in another address space where the object (to which the parameter refers) may not exist. The preceding restriction on object parameters deals with this difficulty in one of three ways:

- If the object is **SOMRReplicbl**, it is assumed that the object is already in the other address space and only the name of the object is passed.
- If the object is **SOMPPersistent**, the object name (used by the Persistence Framework) is passed and the object is restored (activated), if necessary.
- If the object is **SOMRLinearizable**, the object is copied to the other address space. A linearizable object must support the methods **somrGetState** and **somrSetState**; the class **SOMRLinearizable** contains the specification of both of these methods.

Passing any other object as a parameter to **somrLockNlogOp** can produce unpredictable results. If this constraint cannot be satisfied, then the application program should use value logging rather than operation logging to achieve replication, as described in the next section.

Note: If the "ReplicatedDog" class uses "callstyle = idl" in its interface definition, then the first parameter in <parameters> is always an Environment parameter, which is neither a primitive CORBA type nor an object of the right class. Currently the Replication Framework ignores the Environment parameter. However, it must be included in the parameter list <parameters> in order to match the number of arguments expected for the method. Also, currently "Context" object parameters are not supported. These restrictions on Environment and Context parameters apply only to those methods that need to be replicated.

Step 4. Initialize the object (replica) for replicated operation as follows (either in the main program or from inside of **somInit** of "ReplicatedDog", if **somInit** is overridden):

```
#include <somrerrd.h>

ReplicatedDog dog
Environment *Env;
int rc;
dog = ReplicatedDogNew();
Env = SOM_CreateLocalEnvironment();

  _somrSetObjName( dog, Env, "Lassie" );
  rc = _somrRepInit( dog, Env,  'o', 'w');
  if (Env->_major == NO_EXCEPTION) {
     somPrintf(
       "Successfully initialized for replication. rc = %d\n",
                                                               rc);
        ...
       }
  else {
      somPrintf("Initialization for replication failed\n");
      switch(somriGetErrorCode(Env)) {
          case SOMR_MASTER_UNREACHABLE:  ...
          case SOMR_UNAUTHORIZED:  ...
          case SOMR_TRYLATER: ...
          default: ...
      }
```

The **somrRepInit** parameter 'o' indicates operation logging, while 'w' indicates this replica is a writer.

Step 5. Override **somrGetState** and **somrSetState** to format the state of the object as a byte string and to set the state of the object from such a string. Both methods require that the first four bytes of the string contain its length in binary. (Note that for a persistent object, you might use the Encoder/Decoder of the Persistence Framework, described in the previous chapter, with a Media Interface that returns a buffer.) The Replication Framework calls these methods on the replicated object to synchronize new replicas with the existing ones (**somrGetState** on existing replicas and **somrSetState** on new replicas). For help in implementing these two methods, refer to the sample programs shipped with the SOMobjects Developer Toolkit.

Step 6. To have a replicated object respond to *directives*, override the **somrDoDirective** method. See the subsequent section entitled "Handling Directives."

6.5 Making a “replicated” class that provides value logging

This example is similar to the previous one, except that this new class will use value logging to propagate updates among its replicas.

Step 1. Create a new class whose parents are both **SOMRReplicbl** and the original class (“Dog”) whose instances are like those that will be replicated.

Step 2. In the new class, override all the methods that change the value of instance variables (of “Dog”) and override the following methods of **SOMRReplicbl**: **somInit**, **somrApplyUpdates, somrDoDirective**, **somrGetState** and **somrSetState**. What to do in each overriding method is explained in the following steps.

Step 3. Each overridden method from the original class (such as “Dog”) should be coded as follows:

```
dogMethod( ReplicatedDog somSelf, <parameters> ) {
      char *buf;
      Environment *Env = SOM_CreateLocalEnvironment();
      _somrLock( somSelf, Env );
      if (Env->_major == NO_EXCEPTION) {
            parent_dogMethod( somSelf, <parameters> )
            buf = <some algorithm to capture the change
                        in the state of the object>;
            _somrReleaseNPropagateUpdate( somSelf
                                    Env,
                                    ”ReplicatedDog”,
                                    buf,
                                    <buf length in bytes>,
                                    0 );
            }
      else {
            /* code to handle failure to obtain a lock */
            switch (somriGetErrorCode(Env)){
            case SOMR_MASTER_UNREACHABLE:  ...
            case SOMR_UNAUTHORIZED:  ...
            case SOMR_TIMEOUT:  ...
            case SOMR_TRYLATER  ...
            default:  ...
            }
}
```

where `<parameters>` represents the explicit formal parameters of “dogMethod”.

Note that **somrGetState** may be used to capture the value of the object. However, you could propagate updates by differentials in the object value. This point reveals the subtle difference between value logging and merely capturing the value; that is, with value logging you have the possibility of computing and propagating differentials.

Step 4. Override **somInit** to initialize the object for replicated operation as follows:

```
SOM_Scope void  SOMLINK somInit(  ReplicatedDog somSelf)
{
    ReplicatedDogData *somThis = ReplicatedDogGetData( somSelf );
     /* make somInit calls on all parents */
    _somrSetObjName ( somSelf, Env, ”Lassie”);
    _somrRepInit ( somSelf, Env, ’v’, ’w’ );
}
```

The **somrRepInit** parameter ‘v’ indicates value logging, whereas ‘w’ indicates this replica is a writer.

Note that if the application will create more that one distinct instance of “ReplicatedDog” (for example, other names besides “Lassie”), this initialization cannot be done by overriding **somInit**, because there is no way to pass the different object names to the **somInit** method. Instead, the methods **somrSetObjName** and **somrRepInit** must be called separately after the replica is created (for instance, after the call to “ReplicatedDogNew”) as follows:

```
varLassie = ReplicatedDogNew();
_somrSetObjName ( varLassie, Env, ”Lassie”);
_somrRepInit ( varLassie, Env, ’o’, ’w’ );
```

Step 5. Override **somrGetState** and **somrSetState** to format the state of the object as a byte string and to set the state of the object from such a string. Both methods require that the first four bytes of the string contain its length. (Note that for a persistent object, you might use the Encoder/Decoder of the Persistence Framework (described in the previous chapter) with a Media Interface that returns a buffer.)

Step 6. To have a replicated object respond to *directives*, override the **somrDoDirective** method. See the subsequent section entitled “Handling Directives.”

6.6 Handling directives

A *directive* is a message from the Replication Framework to a replica (or more specifically, to the application using the replica). A directive indicates that some condition has arisen asynchronously (not as a reaction to any request by the local replica). The **somrDoDirective** method is provided by the framework to interpret directives sent to a replica. To enable replicated objects to handle directives appropriately for the application, override the **somrDoDirective** method. The overriding method might be coded as follows:

```
SOM_Scope void SOMLINK somrDoDirective( ReplicatedDog somSelf,
                                        Environment *env,
                                        string directive )
{
        if ( !strcmp( directive, "BECOME_STAND_ALONE" ) {
                ... }
        else if ( !strcmp( directive, "CONNECTION_LOST" ) {
                ... }
        else if ( !strcmp( directive, "CONNECTION_REESTABLISHED" ) {
                ... }
}
```

where the ellipses represent application-dependent code. Currently defined directives are as follows:

BECOME_STAND_ALONE — The replica has lost its connection to other replicas and the Replication Framework has given up trying to reconnect to the set of replicas. An application may subsequently try to reconnect by invoking first **somrRepUninit** and then **somrRepInit**.

LOST_CONNECTION — The replica has lost its connection to the other replicas, but the Replication Framework is trying to reconnect; the directive below will be issued after connection is reestablished. You should not attempt to update the object at this time.

CONNECTION_REESTABLISHED — The connection between the replica and its cohorts has been reestablished; processing may continue normally.

LOST_RECOVERABILITY — This directive is issued by the framework when the *.scf* file cannot be updated to reflect the current state of the framework (due to network or file errors). Hence, because the framework uses the *.scf* files for recovery, recovery may be impacted adversely.

Figure 2 depicts the states of a replica and how transitions are made. Note that from the stand-alone state, **somrRepUninit** should be invoked before trying to invoke **somrRepInit** and that neither **somrRepInit** nor **somrRepUninit** should be invoked inside of the method **somrDoDirective** (because doing so deletes the object on which **somrDoDirective** is invoked). In addition, bear in mind that the **BECOME_STAND_ALONE** directive is received because something is amiss out in the network; it is prudent to wait a bit before trying to reconnect.

Note that in the **stand-alone** and **isolated** states the methods **somrLockNLogOp** and **somrLock** cannot succeed. Attempts to invoke these methods yield a return code of **SOMR_TRYLATER.**

Because of their passive nature, read-only replicas do not receive **BECOME_STAND_ALONE** directives, but can receive **LOST_CONNECTION** directives.

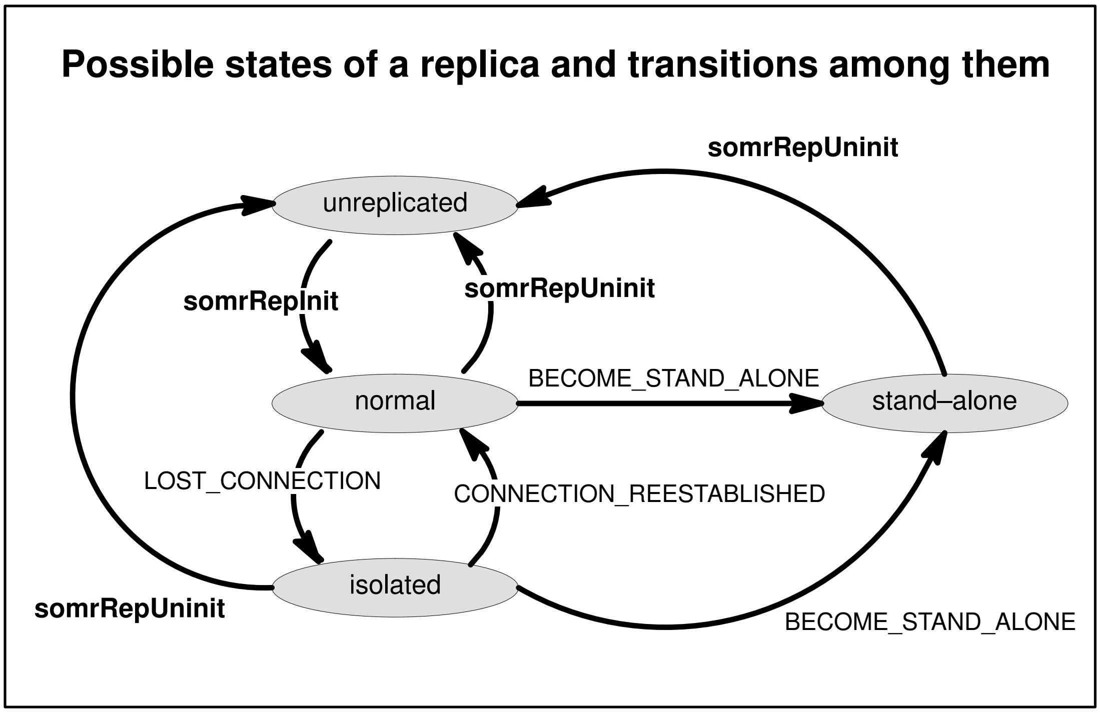


Figure 2. Possible states of a replica and transitions among them.

Chapter 7. Introduction to the Collection Classes

The Collection Classes described in this chapter constitute a large group of classes and methods provided for the programmer's convenience. Collection Classes — sometimes also 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. The collection classes can be used in client code "as is," or they can be used as the basis for deriving new classes.

Reference documentation for these classes and their related methods is contained the *SOMobjects Developer Toolkit: Collection Classes Reference Manual.*

7.1 Categories of Collection Classes

The collection classes are organized into the following categories:

- **Abstract classes** — Define the conceptual operations that are implemented by methods in other (sub)classes.
- **Main collection classes** — Represent each of the implemented data structures for collecting elements into a group.
- **Iterator classes** — Define an iterator class corresponding to each of the main collection classes, enabling clients to iterate through each of the objects in the collection.
- **Mixin classes** — Define characteristics that apply to more than one kind of collection class, such as ordering or linking. A collection class may also require certain mixin characteristics in objects that 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; is of interest primarily to those deriving new collection classes.

Each group of the foregoing classes is discussed more thoroughly in subsequent topics, with particular emphasis on the main collection classes.

IsSame vs. IsEqual comparisons

The distinction between the **IsSame** vs. **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:

1. **IsSame** is true when two objects are really the *same* object. This means "both" objects are literally the same object; that is, both parts of a comparison are testing the same instantiation.
2. **IsEqual** is true when two objects are *equivalent* objects. This would occur when two different instantiations contain the same values, or at least values which, for the sake of the comparison, can be considered the same. Stated differently, the two instantiations are isomorphic.

Class inheritance vs. element inheritance

There are two distinct aspects of inheritance that pertain to each class:

1. The inheritance of the *collection class* itself that is derived from its parent or base class.
2. 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 totally 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.

The subsequent topics describing each collection class also discuss any pertinent inheritance restrictions of the class and its contained elements.

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 to distinguish between initializers*>

The initializers can also 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.

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.

7.2 Abstract Classes

The Annotated C++ Reference Manual[1] describes an abstract class as follows: "The abstract class mechanism supports the notion of a general concept, such as a `shape`, of which only more concrete variants, such as `circle` and `square`, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations."

The concept of an *abstract base class*, which was briefly discussed earlier in this manual, 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 as well. This is especially true when defining basic behavior in a parent class that applies for all children of that class. This concept allows class implementors the flexibility to use either of two approaches:

1. 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.
2. 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 which declares a method interface in the parent that all children *must* override and redefine. Specifically, the **somf_TIterator** class declares the methods **somfFirst** and **somfNext,** which all children derived from **somf_TIterator** must override. The **somf_TIterator** class is discussed in the topic "Iterator classes," as well as in the *SOMobjects Developer Toolkit: Collection Classes Reference Manual*.

The Abstract Classes include the following classes:

somf_TCollection	—	Represents a group of objects.
somf_TIterator	—	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	—	Declares the behavior common to all collections whose elements are ordered.
somf_TSequenceIterator	—	Declares the behavior of all iterators for children of the **somf_TSequence** class. This class is also a child of the **somf_TIterator** class.

1. Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual* (Addison–Wesley Publishing Company, 1992).

7.3 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
- Dictionary
- Set
- Deque, queue, and stack
- Linked list
- Sorted sequence
- Priority queue

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.

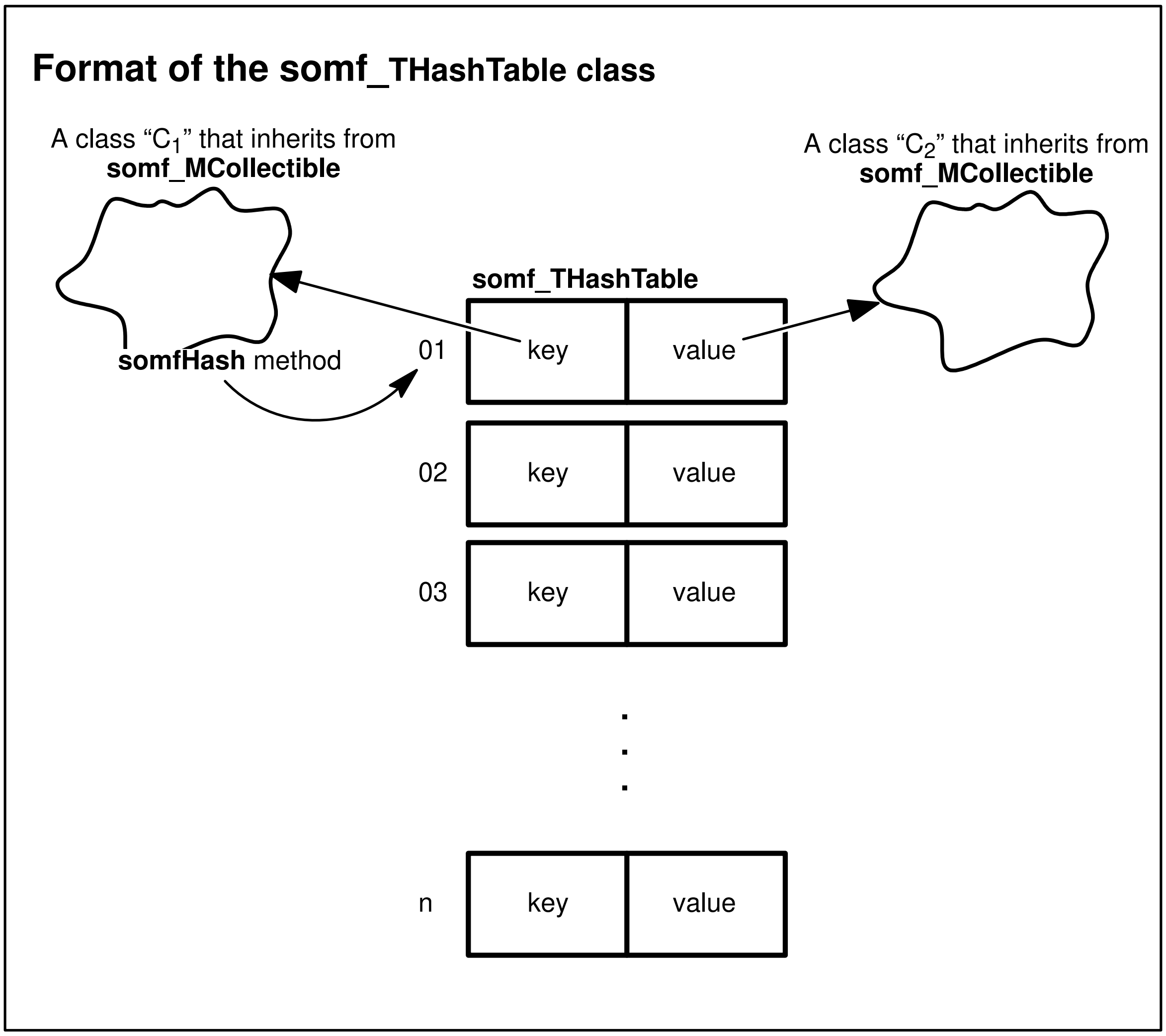


Figure 1. **somf_THashTable** format

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 hash probe) used to map directly into the table.

For example, 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 the **somfHash** method 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 the **somfHash** method to provide the hashing algorithm for the table. The **somf_MCollectible** class is discussed further in the topic “Mixin classes,” as well as in the *SOMobjects Developer Toolkit: Collection Classes Reference Manual,* which also provides additional information about the **somf_THashTable** class.

Objects of the **somf_THashTable** class use the **IsSame** operation to compare objects (see the earlier topic “IsSame vs. IsEqual”). 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 (key, value) 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 that the dictionary uses an **IsEqual** operation to compare objects (see the earlier topic “IsSame vs. IsEqual”). 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. See the descriptions of the **somf_TDictionary** and **somf_MCollectible** classes in the *SOMobjects Developer Toolkit: Collection Classes Reference Manual* for more details.

Note: 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 (key, value) 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. Note that 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:

- A *queue* is a list where the elements are inserted and removed using a first-in, first-out (FIFO) approach.
- A *stack* is a list where the elements are inserted and removed using a last-in, first-out (LIFO) approach.
- A *deque* is a double-ended queue, which permits insertion and removal at either end of the list.

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. See the descriptions of the **somf_TDeque** and **somf_MCollectible** classes in the *SOMobjects Developer Toolkit: Collection Classes Reference Manual* for more details.

Linked list class — somf_TPrimitiveLinkedList

A *linked list* is a collection where each element in the list is linked to the element in front of it and also to the element behind it. Insertion into the list is relative to the elements that are 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 also one of the most restrictive:

- The **somf_TPrimitiveLinkedList** class is the only main collection class that does *not* inherit from **somf_MCollectible.** This cuts down on processing overhead, but it also means that <u>an instance of **somf_TPrimitiveLinkedList** *cannot* be inserted into any other main collection class</u>.
- Elements in the **somf_TPrimitiveLinkedList** class inherit from **somf_MLinkable**, which means the ‘linkability’ of each object is inherent in the object itself. Hence, <u>each object can be inserted into the **somf_TPrimitiveLinkedList** only *once*</u>, because it has only one set of unalterable forward/backward links.

For more information, see the descriptions of the classes **somf_TPrimitiveLinkedList** and **somf_MLinkable** in the *SOMobjects Developer Toolkit: Collection Classes Reference Manual*.

Note: As discussed, although the **somf_TPrimitiveLinkedList** class carries no baggage from the other classes and is as compact as possible, that also means 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 **somfIsGreaterThan**, so that **somf_TSortedSequence** will be able to position them properly.

For more information, see the descriptions of the classes **somf_TSortedSequence** and **somf_MOrderableCollectible** in the *SOMobjects Developer Toolkit: Collection Classes Reference Manual*.

Priority queue class — somf_TPriorityQueue

A *priority queue* is a special case of the sorted sequence. Its ordering is also based on how the elements relate to each other, but it is geared more toward holding larger volumes of data.

Robert Sedgewick[2] describes a priority queue as follows:

> 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 **somfIsGreaterThan**, so that the **somf_TPriorityQueue** will be able to position them correctly.

See the descriptions of the **somf_TPriorityQueue** and **somf_MOrderableCollectible** classes in the *SOMobjects Developer Toolkit: Collection Classes Reference Manual* for more details.

Choosing the best class

If you are unsure which main collection class you should use, the following selection chart may prove helpful.

2. Robert Sedgewick, *Algorithms in C++* (Addison–Wesley Publishing Company, 1992), p. 145.

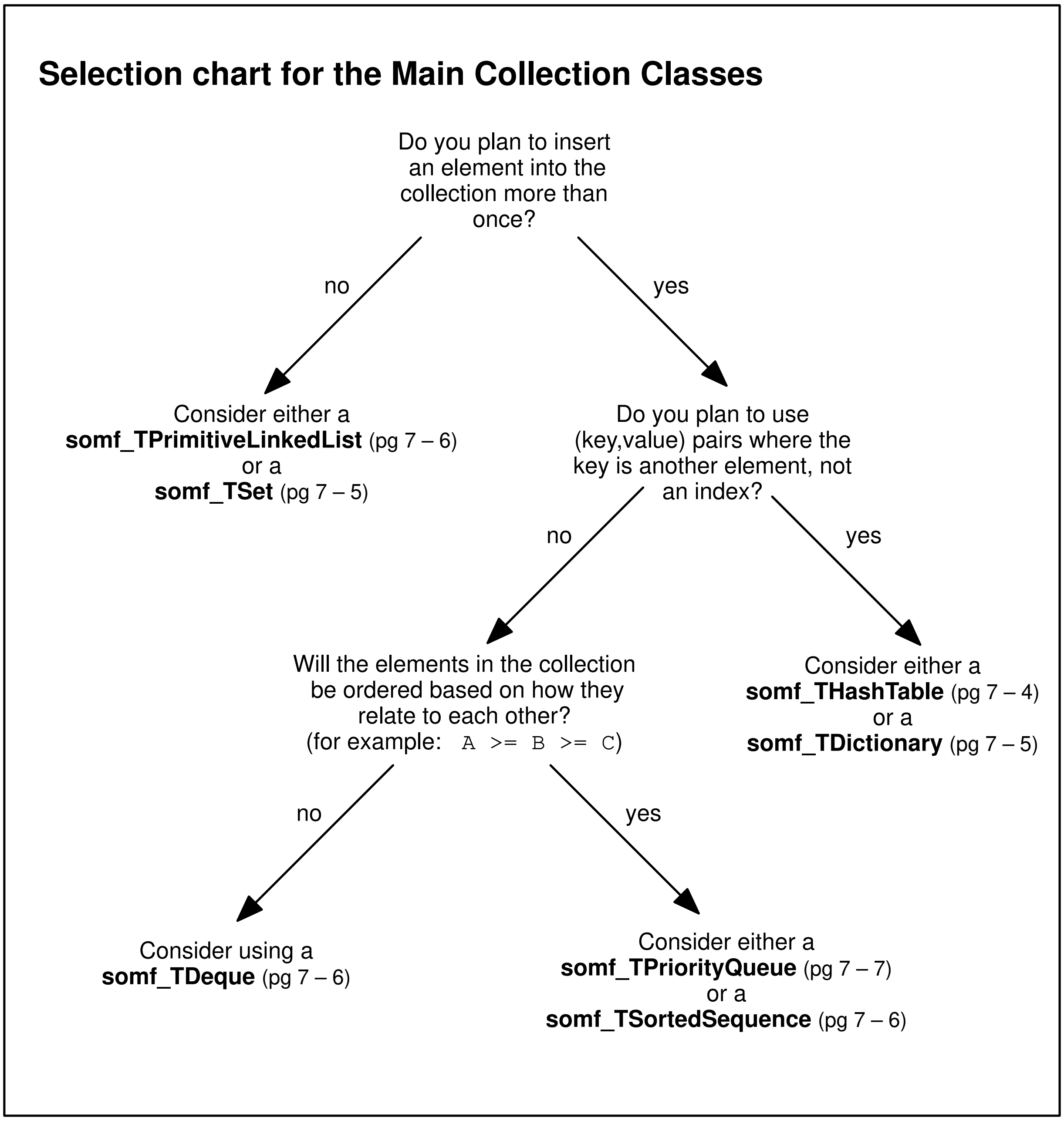


Figure 2. Collection class selection chart

Chapter 8. Introduction to the Emitter Framework

8.1 Introduction

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 documentation file, a description that can drive a class browser, or a pretty-printed interface specification. Given the number of programming languages with which SOM can be used and the many development-support tools that can leverage object interface definitions, the SOM Compiler needs to produce a large number of output forms. Therefore, an important structural feature of the SOM Compiler is that it minimizes the effort involved in developing and maintaining new compiler "back-ends."

Figure 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, but the IDL parser remains unchanged.

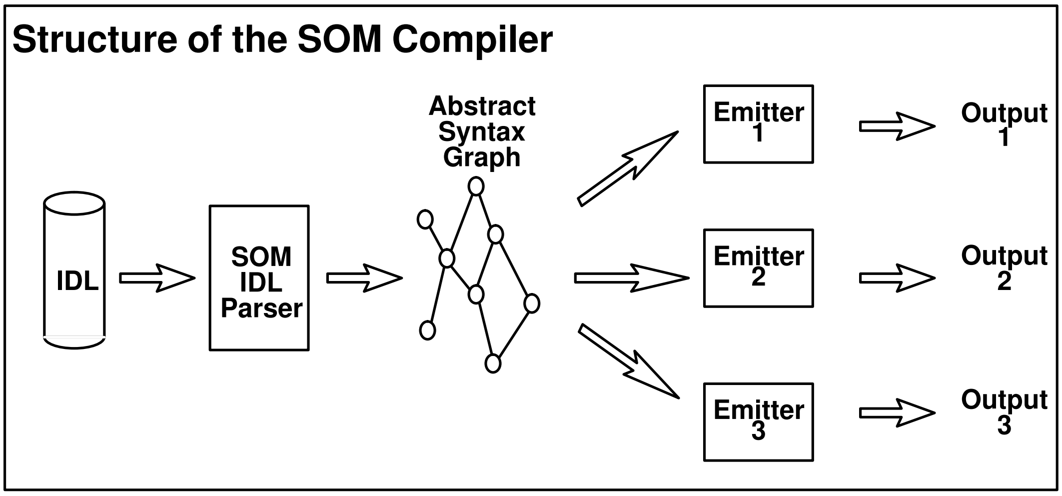


Figure 1. The structure of the SOM Compiler

To make it easier to develop new emitters for use with the SOM Compiler, the SOMobjects Toolkit provides a collection of classes called the Emitter Framework. The Emitter Framework consists of several support classes and a general emitter class that can be subclassed to produce a specific emitter. The SOMobjects Toolkit 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 goals of the emitter classes are to provide an object-oriented framework for emitter development that:

- Insulates new emitter code from changes to SOM's interface definition language (SOM 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.

8.2 The Structure of the Emitter Framework

The Emitter Framework is structured as shown in Figure 2.

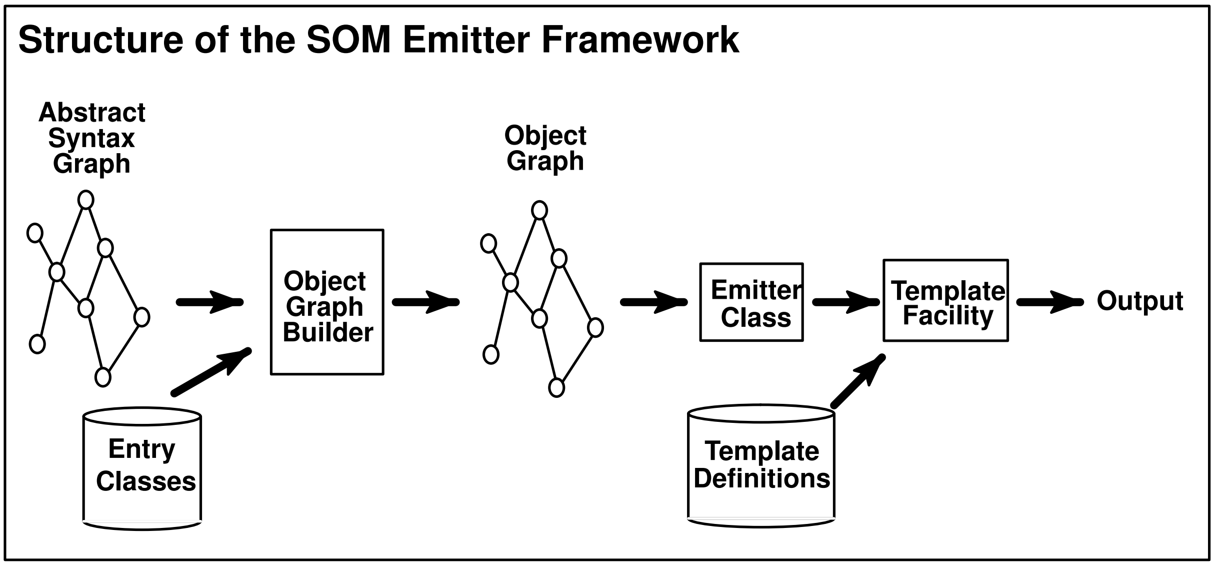


Figure 2. The structure of the SOM Emitter Framework

The object graph builder

The input to the Emitter Framework is an abstract syntax graph of data structures. As shown in (the earlier) Figure 1, 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*.

The subsequent major section "Emitter Framework Classes" 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.

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.

The following section discusses the components of the Emitter Framework and describes the recommended procedure for producing a new emitter.

8.3 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* **(SOMTEntryC** 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.

The remainder of this section discusses each of these classes. For more information, the Reference portion of this book provides detailed information on all of the attributes and methods supported by each class in the Emitter Framework.

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.

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 (see Figure 2).
- 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 (see Figure 2).
- 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 passthrus 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.

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:

Prolog	Describes text to be emitted before any other sections (**prologS**).
Base Includes	Determines how base (parent) class #include statements are emitted (**baseIncludesS**).
Meta Include	Determines how a metaclass #include statement is emitted (**metaIncludeS**).
Class	Determines what information about the class as a whole is emitted (**classS**).
Base	Determines what information about a base (parent) classes of a class is emitted (**baseS**).
Meta	Determines what information about the class's metaclass is emitted (**metaS**).
Constant	Determines what information about user-defined constants is emitted (**constantS**).
Typedef	Determines what information about user-defined types is emitted (**typedefS**).
Struct	Determines what information about user-defined structs is emitted (**structS**).
Union	Determines what information about user-defined unions is emitted (**unionS**).
Enum	Determines what information about user-defined enumerations is emitted (**enumS**).
Attribute	Determines what information about the class's attributes is emitted (**attributeS**).
Methods	Determines what information about the methods of a class is emitted (**methodsS**). More specialized method sections can be specified using **inheritedMethodsS** or **overrideMethodsS**).
Release	Determines how information about the release order statement of a class definition is emitted (**releaseS**).
Passthru	Determines what information about passthru statements is emitted (**passthruS**).
Data	Determines what information about internal instance variables of a class is emitted (**dataS**).
Interface	Determines what information about the interfaces in a module is emitted (**interfaceS**).
Module	Determines what information about a module is emitted (**moduleS**).
Epilog	Describes text to be emitted after all other sections are emitted (**epilogS**).

Some sections apply to a variable number of items that must be dealt with iteratively. This can be true of the *base* section (since a class can have more than one base class), as well as the sections for *base includes, data*, *passthru*, *attribute, constant, typedef, struct, union, enum, interface, module,* and *method*. These repeating sections can be preceded by a *prolog* (information to be emitted *prior* to iterating through the items), and followed by an *epilog* (information to be emitted *after* iterating through the items). The **SOMTEmitC** class provides methods for emitting each of the sections described above.

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.

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 a good separation between decision logic and format specification. Also, because the format specification is isolated, its readability and maintainability are greatly enhanced.

Following is an example fragment of a text template containing two template sections, “classS” and “metaS”. The text template is stored in a *template file* associated with the emitter. (The subsequent paragraphs provide further explanation for preparing the text template.)

```
:classS
class: <className><, classMods, ...>;
?<-- classComment>
:metaS
metaclass: <metaName>
```

New sections are denoted by lines that begin with a colon. The above fragment contains two sections, “classS” and “metaS”. (By convention, section names end in capital “S”.) An emitter uses the section name to specify to the template object which part of the output file to emit. Lines that begin with a question mark are emitted only if at least one symbol appearing on the line is defined with a nonblank value. Other lines are emitted unconditionally.

Symbols (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.

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 <mhello.idl>

interface Hello : SOMObject /* This is the interface for Hello. */
{
    implementation {
    metaclass = M_Hello;
    functionprefix = "hello_";
    filestem = hello;
    . . .
    };
};
```

would produce the following output:

```
class: Hello, functionprefix = hello_, filestem = hello;

// This is the interface for Hello.
metaclass: M_Hello
```

(The formatting of comments varies, depending on the attributes of the emitter's template.)

The entry classes (SOMTEntryC, 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 3.

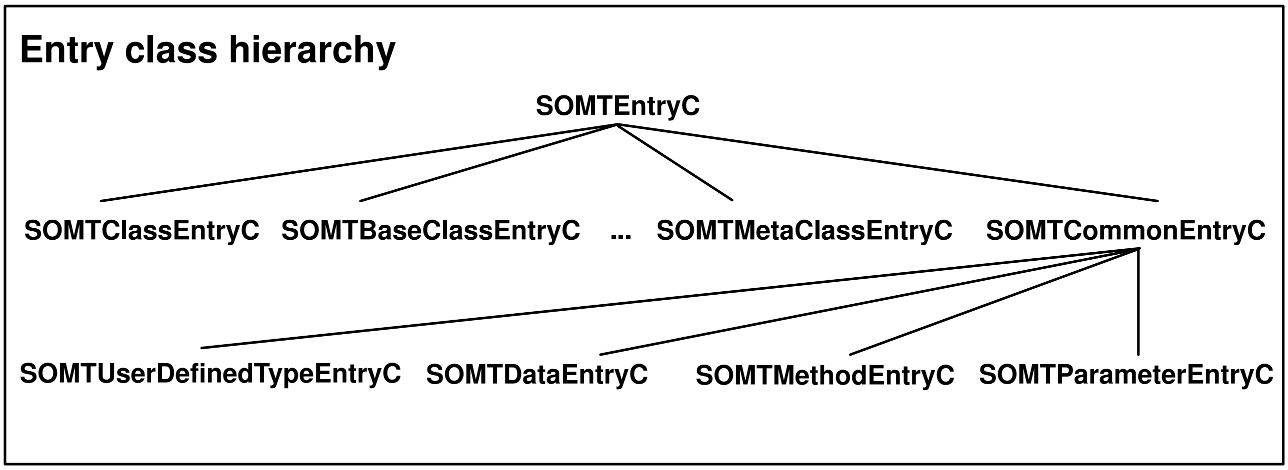


Figure 3. The entry class hierarchy.

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, see the Reference portion of this book.

SOMTEntryC

The **SOMTEntryC** class provides **attributes** for accessing the name of an entry (**somtEntryName, somtIDLScopedName,** and **somtCScopedName**), its entry type (that is, whether it represents a class, method, attribute, typedef, etc.) (**somtElementType** and **somtElementTypeName**), its comment (**somtEntryComment**), the line number in the .idl file where the entry is defined (**somtSourceLineNumber**), its type code (**somtTypeCode**), and whether the entry represents a reference to an entry rather than its definition (**somtIsReference**).

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 **SOMTParameterEntryC**.

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 **SOMTEntryC** 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.

8.4 Writing an Emitter — the Basics

The ‘newemit’ facility

The **newemit** emitter generator is a program, written using the Emitter Framework, that generates a complete, working emitter. This emitter is easily customized as needed.

The following steps outline the recommended approach to writing an emitter.

Running the ‘newemit’ program

The command to execute the **newemit** program takes the following syntax:

newemit [–C | –C++] <*className*> <*filestem*>

Required arguments are the name for a new subclass of **SOMTEmitC** to be created and a file stem for the subclass. The optional –C or –C++ (lowercase is also valid) specifies the language in which the emitter will be written; the default is C. The program produces the following files:

<*filestem*>.idl — An IDL definition of the subclass of **SOMTEmitC** with the specified name. This IDL definition specifies that the **somtGenerateSections** method will be overridden by the new subclass.

<*filestem*>.c or <*filestem*>.C or <*filestem*>.cpp
— The C or C++ implementation file for the new subclass of **SOMTEmitC**. Initially, this implementation file has the same code for **somtGenerateSections** as defined by **SOMTEmitC**. (The C++ extension is .C on AIX or .cpp on OS/2.)

emit<*filestem*>.c or emit<*filestem*>.C or emit<*filestem*>.cpp
— A C or C++ emitter driver program. Notice that the driver-program name is always “emit” followed by the specified file stem. (The C++ extension is .C on AIX or .cpp on OS/2.)

Makefile — A Makefile for creating a DLL for the new emitter.

<*filestem*>.efw — A sample output template file.

The <*filestem*>.c (or .C or .cpp) and <*filestem*>.efw files will be customized to produce a particular output format. For example, to create a documentation emitter class called “DocEmitter” whose related files (written in C by default) have a file stem of *doc*, we would execute the following command:

```
newemit DocEmitter doc
```

The result would be files doc.idl, doc.c, emitdoc.c, Makefile, and doc.efw. The remaining steps involve customizing the doc.c and doc.efw files.

Note: On AIX, the <*filestem*> argument to the **newemit** program should consist of *only* lowercase characters.

Designing the output file

Look at a typical IDL interface definition, and hand-construct the desired output file for that interface. For example, suppose we want our “DocEmitter” to construct a documentation file that simply lists the class name and the return types for the methods it introduces or overrides. Thus, given the following IDL specification,

```
#include <somobj.idl>
interface Animal: SOMObject {
void setSound(in string sound);
void makeSound();
};
```

we would want the output file to look like this:

```
The following methods:
  setSound, of type void
  makeSound of type void
are implemented by class Animal.
```

Constructing an output template

The next step is to construct an output template, based on the sample output file. Separate the sample output file into different *sections*, based on the aspect of the IDL specification to which they most closely correspond (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 "Animal" in the current example can be replaced by the standard symbol <className>.

In sum, the output file shown above could be generalized to the following output template:

```
:methodsPrologS
The following methods:
:methodsS
  <methodName>, of type <methodType>
:classS
are implemented by class <className>.
```

Output templates are typically stored in files having a .efw extension. The **newemit** program creates a generic template file, <*filestem*>.efw. It contains all the standard sections, and each section contains sample template text that exercises all the standard symbols available within that section. This generic template can be edited to contain only those sections needed by the new emitter, with appropriate text. For the current example, file doc.efw would be edited to contain the generalized output template shown above.

Note: Each of the entry classes (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, passthrus, attributes, constants, typedefs, struct, enums, union, interfaces, and nested modules.

Compiling and running the new emitter

Compile the driver program provided by **newemit** and the implementation of your emitter together to create a dynamically linked library (a DLL) for a new emitter. The **newemit** program provides a Makefile to perform this step (simply enter “make”).

The new emitter can now be invoked by the SOM Compiler via the **–s** option (which overrides the SMEMIT variable, for the current **sc** command, with the specified emitter). For example, “DocEmitter”, packaged in emitdoc.dll, can be invoked by running the SOM Compiler with the “–sdoc” option. (Note that the value of the **–s** option is the file stem specified earlier to **newemit**.) Invoking the following **sc** command will produce an animal.doc file just like the one shown at the beginning of this section:

```
sc –sdoc animal.idl
```