SOMobjects Developer's Toolkit

Programmer's Guide, Volume II: Object Services

SOMobjects Version 3.0

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

Second Edition (December 1996)

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

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

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

Notices

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

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

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

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

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

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

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

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

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

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

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

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

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

Trademarks and Acknowledgements

AIX is a trademark of International Business Machines Corporation.
IBM is a registered trademark of International Business Machines Corporation.
OS/2 is a trademark of International Business Machines Corporation.
SOM is a trademark of International Business Machines Corporation.
SOMobjects is a trademark of International Business Machines Corporation.
Windows and Windows NT are trademarks of Microsoft Corporation.

Contents

List of Figures

About the Programmer's Guide for Object Services

The *Programmer's Guide for Object Services* contains information about Object Services. Object Services help you manage objects by letting you name them, operate them securely, manage their persistence, and the like. The documentation covers concepts and tasks related to constructing and using managed objects that make use of the SOMobjects Object Services.

To build a robust, object-oriented application, you often have to be concerned with more than just providing application function. You also have to be concerned with how to manage objects. This is even more true in large distributed systems where there are thousands of objects spread over hundreds of hosts. You need to be able to name objects and keep track of them, ensure that only authorized users can operate on them, follow an orderly approach to creating and deleting them, maintain their state persistently between sessions, and so forth. The object services provide support for object management functionality like this.

The SOMobjects Object Services implement a subset of the CORBAservices defined by the Object Management Group (OMG).

Who Should Use This Documentation

This documentation is for software developers using Object Services, as well as for developers who are providing specializations of object services interfaces.

You will find having the following background helpful:

- Familiarity with the OMG CORBA 1.1 and CORBA IDL specifications
- Familiarity with the OMG Common Object Services (also referred to as CORBAservices), in particular the:
  - Externalization Service
  - Naming Service
  - CosObject Identity Module (introduced in the Relationship Service)
- Knowledge of object-oriented principles
- C or C++ programming experience
- IBM SOM and DSOM knowledge, preferably with programming experience
- Familiarity with distributed systems management and object management concepts

Topics Covered

This documentation provides information about the SOMobjects Developer Toolkit for Object Services. Topics covered include:

- **Object Services Server**
- **Externalization Service**
- **Object Identity Service**
- **Naming Service**
- **Security Service**

Typographic Conventions

This book uses the following typographic conventions:

Bold
Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels and icons that you select.

Italics
Identifies parameters and variables whose actual names or values you supply. Also identifies new terminology.

`Monospace`
Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system or information you should actually type.

Related Documentation

The following books contain information about, or related to, SOMobjects Object Services:

- *Common Object Services Specification Volume 1* (OMG Document Number 94-1-1)
- *CORBAservices: Common Object Services Specification* (OMG Document Number 95-3-31)
- *Programmer's Guide for SOM and DSOM*
- *Programmer's Reference for SOM and DSOM*
- *Programmer's Reference for Object Services*
- *Programmer's Reference for Abstract Interface Definitions*

Chapter 1. Overview of Object Services

This section provides an overview and introduction to the SOMobjects Object Services. This includes a discussion of the programming model for object services that spans individual services.

Understanding Object Services

In general, the Object Services provide a combination of concrete and mixin classes. Mixin classes provide your objects with the attributes and behavior that are needed for them to be managed with object services. The concrete classes are used to instantiate the distinguished objects that are used in support of managing your objects.

The SOMobjects Object Services consist of:

Externalization Service
The Externalization Service enables you to provide containers for data called streams, and enables objects to write data to a stream or read data from a stream (that is, externalize or internalize its state). DSOM uses the Externalization Service to implement pass-by-value parameters.

Naming Service
The Naming Service enables you to name objects. You can assign a name to objects within a particular context, and then instances of naming contexts can be organized into a name hierarchy. During the configuration process, SOMobjects builds a default global name tree and binds certain distinguished objects within that name tree.

Object Identity Service
The Object Identity Service enables you to determine whether two objects are exactly the same. Objects are assigned an identity that can be used by the service to determine if two objects are identical.

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

Relationship to Standards

With the exception of the Security Service, all of the SOMobjects Object Services are implementations of the OMG *CORBAservices: Common Object Services Specification* (OMG Document Number 95-3-31).

Note: The following terms might be used interchangeably in this documentation:

- interface and class
- instance and object
- operation and method

To ensure that polymorphism is preserved to its fullest, most of the SOMobjects implementations of these services have been introduced as subclasses of the OMG interfaces. Thus, the OMG interfaces are introduced as abstract base classes that have been subclassed with a concrete implementation.

To further amplify this point, consider the Object Identity Service. OMG provides a specification for the **CosIdentity::IdentifiableObject** interface. SOMobjects introduces this interface as an abstract base class; the interface is provided without any implementation. SOMobjects subclasses the OMG interface and provides an implementation in the subclass, for example, **somOS::ServiceBase** which inherits from **CosIdentity::-IdentifiableObject**. The methods of **CosIdentity::IdentifiableObject** are overridden in **somOS::ServiceBase** with a concrete implementation.

In this way, other implementations of the standard interface can be provided at a later date without burdening objects that use the alternate implementation with the implementation that SOMobjects provides in this release.

In several cases, SOMobjects provides additional operations in the implementation classes. That is, having subclassed the OMG interface to provide an implementation, SOMobjects has added other operations in order to make the implementation more robust and useful. This is most notable in the case of the Naming Service. The naming interface in **ExtendedNaming::ExtendedNamingContext** (a subclass of **CosNaming::-NamingContext**) has been extended to support properties on name-bindings, and the ability to search for bound objects based on their bound-property values. This gives the Naming Services yellow-page characteristics.

It is important to note that while such extensions increase the utility of the service, these extensions are not part of the original standard. Using them in your application will improve the productivity and functionality of your application; however, it will be at the expense of portability to different vendor ORBs. Because many other service specifications are being standardized by OMG, there is a possibility that any extensions introduced in this version of SOMobjects will become standardized over time, either as specified here or in some other form.

Note: Read “About Programmer’s Reference for Object Services” on page ix in *Programmer’s Reference for Object Services*. This chapter describes how interfaces are documented relative to the standard specification and the concrete implementation.

Managed Objects

With the Object Services being introduced in this version of SOMobjects, SOMobjects is instituting the concept of *managed objects*. A managed object is one that can be managed with one or more of the Object Services. This concept is important because, along with the basic object model, it forms the basis of the programmer’s model for robust, distributed objects.

The managed-object programmer’s model has at its heart several key principles, the most important of which is to mask out the complexity of the underlying distributed information system. This complexity becomes more evident when you consider the effect of scale. A typical large-scale enterprise can have tens-of-thousands of host machines and users, use millions of objects, perform millions of transactions, and handle hundreds-of-millions of database records.

To exacerbate this situation, many institutions have highly heterogeneous information systems (different hardware platforms, operating systems, communication networks, middleware, database systems, and so forth) and a significant investment in legacy information systems. And all of this typically is administered by just a hand-full of administrators.

As such, it is essential that objects in the system be manageable and that they be manageable both in concert with as well as independently of the underlying information system infrastructure. These objectives are achieved with the following principles:

- The SOMobjects Object Services are designed to be abstractions of important information system functions. Object classes should be created with the mixin classes provided with the Object Services described herein. This makes objects independent of the underlying infrastructure.
- The SOMobjects Object Services are designed to be frameworks that can be tightly integrated with different underlying infrastructures. In this way, existing, robust information and infrastructure technology can be leveraged. More importantly, administration of managed objects can be coordinated with administration of the underlying information system.

Object Frameworks

An object framework has two distinct characteristics: it provides an abstraction of a particular service to object programmers and it enables a federation of different implementations of that abstraction. Thus, different service providers can produce implementations of a service and, following the rules of the framework, can provide their implementation alongside other implementations. An enterprise administrator or application programmer can pick-and-choose the implementation that best meets their needs without affecting the rest of their programs.

Object-oriented programming provides the initial condition for frameworks with polymorphism. Polymorphism is the ability for different implementations to have the same interface. Polymorphism is essential to a framework as it establishes a major element of transparency to client programmers. However, polymorphism does not ensure the ability to federate those implementations.

For instance, having different subclasses of a generic factory class ensures that each factory implementation has the same interface (ignoring any additional methods that a particular specialization might introduce). However, it does not ensure that the right factory implementation is used at the right time. This latter characteristic is the direct responsibility of the framework design and any programs that are intended to use the service framework.

The SOMobjects Object Services are designed as frameworks. As such, you will often need to understand more than just the interface to the service, you will also have to understand the framework's rules of good behavior. If you create an instance of a managed object, you need to examine the respective services for how objects are registered with the service frameworks. If you create a managed-object specialization, you need to examine the services for any distinguished operations that you necessarily must override or any constraints on behavior that you introduce. If you invoke methods on a managed-object, you must examine the services for any special method sequences that must be followed.

The framework concepts and tasks, and the rules of good behavior that come with them, are described in fuller detail for each of the services in their respective chapters.

Client Programming versus Class Programming

It is important to differentiate between client programing and class programing. Client programming means that you are invoking methods on an object: an instance of a class provided by someone else. In this case, you are subject to the interface and any protocol specified for the object you are calling.

Class programming means that you are providing a class or subclass, usually a particular implementation or specialization. A client program creates an instance of your class and invokes methods on it in accordance with any interface that you subclass or introduce. In this case, you are subject to the interface of any parent classes that you subclass.

Because the SOMobjects Object Services are frameworks, they introduce concrete and abstract methods, and standalone and mixin classes. Most of the defined methods are intended to be used by client programs. However, some of them are only intended to be used within the framework itself. When this is the case and the method is defined publicly, it is so that class programmers can specialize the method so that their particular class implementation can be federated. Even though a public specification for the method is defined, client programs should not invoke these methods directly.

Object Services Server

Based on the CORBA architecture, object identity in the context of a specific object implementation is established by the object adapter. The object adapter in DSOM is the **SOMOA** object which collaborates with an instance of **SOMDServer**. Because the implementation of a managed object is heavily mitigated by the Object Services, a strong relationship exists between the Object Services and the **SOMDServer**. To enable the integration of multiple Object Services in a single managed-object class, SOMobjects is provided with a specialization of **SOMDServer** that is specific to the Object Services. This specialization is known as the **somOS::Server** or Object Services Server.

The Object Services Server should be used in any process in which any of the object services are used. This primarily applies to server processes, but in some cases it might apply to client processes as well.

There might be application function that must be performed as part of the server program or server object in a server process. When this is the case, the **somOS::Server** can be specialized in much the same way as a normal **SOMDServer**.

The Object Life Cycle Model

The life cycle model for an object governs the meaning and conditions by which an object is created, managed, and destroyed. Each framework introduces nuances to the life cycle of an object.

The SOM Life Cycle Model

The SOM kernel introduces a flexible object life cycle model. With SOM alone, without using DSOM or Object Service frameworks, you can use the C language *className***New** or C++ **new** macro, or the direct **somNew** method (or its variants) on the class object. This is described in “Creating Instances of a Class” on page 72 of *Programmer’s Guide for SOM and DSOM*. In addition, you can use initializers that you introduce. Then, when you no longer need them, you use the **somDestruct** method to destroy objects. As the sole user of any object, you need not coordinate with other possible users of that object. See “Initializing and Uninitializing Objects” on page 195 of *Programmer’s Guide for SOM and DSOM*.

The DSOM Life Cycle Model

DSOM introduces some variations to SOM’s basic life cycle model. With DSOM you use the Factory Service to locate a factory object. Unless you introduce a specific factory object, the factory service normally returns a class object that you can manipulate with the SOM life cycle model for creating objects. The implication of an object’s being distributed is that it can be shared which means that more care must be taken when creating and, especially, when destroying an object. Because you are presented with a proxy object instead of the actual target object in your client address space, you need to **release** it when you are done with the object because some other client, sharing the target object, might still need it. All the clients sharing an object must be coordinated to ensure that the object's life cycle is properly preserved while it is needed. Consequently, the life cycle model is more complicated and restrictive. For more information see "Distributed SOM" on page 229 of *Programmer's Guide for SOM and DSOM*.

The Object Services Life Cycle Model

The SOMobjects Object Services place their own conditions on the life cycle model for objects. This is driven mostly by the tendency for managed objects to have persistent state or persistent references, and the need to coordinate the life cycle of the managed object with the object services used. Each service imposes slight variations on the general model. However, the general model is as simple as possible. As with DSOM, you should begin by using the factory service to locate an appropriate factory. Your next step depends on the type of factory you locate. If you are returned a class object, the process is more involved. Essentially, you perform a **somNewNoInit** on the class object followed by **init_for_object_creation** on the newly created object and any other initializers introduced for that class of object. This object creation is described in detail in "Overview of the Object Life Cycle Model" on page 40 and "Managing the Object Life Cycle" on page 41.

Although the object creation process involves many steps, you can make the process more convenient for other client programmers by introducing a factory object that performs these multiple steps within a single convenience method. In this case, you are responsible for implementing the specifics of object creation and giving your convenience method a signature that clients can use. You should register your factory with the **regimpl**, see "The regimpl Registration Utility" on page 32 of *Programmer's Guide for SOM and DSOM*.

When a managed object is persistent, it is subject to a sub-life cycle model governing its presence in memory and coordinating its persistent state between memory and persistent storage. However, this sub-life cycle is normally transparent to client programmers and affects only system and class programmers.

The Relationships between Objects and Object Services

If you plan to use certain object services, you should explore how they affect the general life cycle model. The following sections of this book can help you understand the relationships between objects and object services:

- **How to Initialize Streamable Objects**
- **How to Use a Stream**
- **Creating Contexts**
- **Object Services Server**
- **Overview of the Object Life Cycle Model**
- **Managing the Object Life Cycle**

Chapter 2. Externalization Service

The Externalization Service provides containers for data called *streams*. A stream can contain data for many objects. When you write an object's data into the stream, it is called *externalizing* the object. When you read an object's data from the stream, it is called *internalizing* the object.

The Externalization Service as provided with SOMobjects 3.0 is a partial implementation[1] of the *OMG Object Externalization Service* specification (OMG TC Document 940915). Specifically, only the classes in the **CosStream** module are provided.

Class Descriptions

The classes defined by the OMG specification all begin with **Cos**, for example **CosStream::Streamable**. The OMG classes are provided in IDL files beginning with the letters "omg", for example "omgestio.idl". The OMG classes are abstract and contain no implementation. The IBM-supplied implementation of each of these classes is a subclass of each OMG class and has the same name, except that it begins with the letters **som**, for example **somStream::Streamable**. The IBM-supplied classes are provided in IDL files beginning with the letters "som", for example "somestio.idl".

The following figure shows the relationships between the Externalization Service objects and classes.

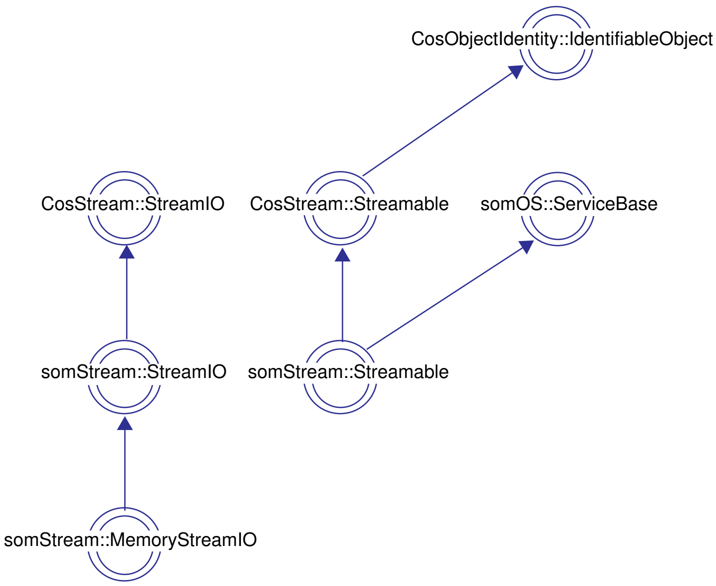

Figure 1. Object Externalization Service Class Diagram

How to Make a Streamable Object

To be externalized or internalized, the class of the object must inherit from the **somStream::Streamable** class. The following example shows the IDL for an **Item** class

1. Some changes are made to the OMG definitions. See "Variances from the OMG Specification" on page 12

that inherits from **somStream::Streamable**. **Item** overrides both **externalize_to_stream** and **internalize_from_stream**. Every subclass of **somStream::Streamable** that contains data must override these and add code to write and read its data. If a class has more than one parent that inherits from **somStream::Streamable**, then it must override **externalize_to_stream** and **internalize_from_stream** even if it has no data. The overridden method should call each parent's method.

```
#include <somestrm.idl>
interface Item : somStream::Streamable {
  attribute string description;
  attribute float cost;
  #ifdef __SOMIDL__
  implementation {
    description: noset, noget;
    externalize_to_stream: override;
    internalize_from_stream: override;
    init_for_object_creation: override;
    somDestruct: override;
    releaseorder: _get_description, _set_description,
                  _get_cost, _set_cost;
    memory_management = corba;
    dllname = "purchase.dll";
  };
  #endif // __SOMIDL__
};
```

The following example shows the **externalize_to_stream**[2] method implementation for the sample **Item** class using C++ bindings. To make this sample code complete, the environment needs to be checked for errors after every call. For clarity, the checks are omitted here.

```
SOM_Scope void SOMLINK
externalize_to_stream(Item *somSelf,
          Environment *ev,
          CosStream_StreamIO* stream)
{
  ItemData *somThis = ItemGetData(somSelf);
  ItemMethodDebug("Item","externalize_to_stream");

  if (!((somStream_StreamIO*)stream)->already_streamed(ev,
                      (CosStream_Streamable*)somSelf,_Item)) {
    stream->write_string(ev,somThis->description);
    stream->write_float(ev,somThis->cost);
    Item_parent_somStream_Streamable_externalize_to_stream(somSelf,
         ev,stream);
  }
}
```

The **externalize_to_stream** method first calls the **already_streamed** method. This informs the **somStream::StreamIO** that the data of the specified object for the specified class is about to be written. The **already_streamed** method returns TRUE if the data for that part of the object has already been written (or read). The **already_streamed** method is intended to solve the "diamond top" problem. The diamond top problem occurs when an object inherits from a streamable parent class by more than one ancestor path, so that when the object is externalized, the parent class **externalize_to_stream** method is called multiple times. The use of **already_streamed** is optional. You can choose not to use it if you know that none of the descendants of your class will inherit from it more than once, or if

2. For historical reasons, the method and parameter names use the word "stream", but "streamio" would be more accurate.

you don’t care that the data is written more than once, or if you use a different mechanism to solve the diamond top problem.

Next, the **externalize_to_stream** method writes the data introduced by the class. You may decide to not write some data. For example, some data is not important, such as temporary flags or buffers. Next, the parent methods are called for all the parents that inherit from **somStream::Streamable**. The order in which you write the data and call your parents is not important. However, the **internalize_from_stream** method must read the data and call its parents in the same order as the **externalize_to_stream** method writes the data and calls its parents.

The following example shows the **internalize_from_stream** method implementation for the sample **Item** class using C++ bindings.

```
SOM_Scope void SOMLINK
internalize_from_stream(Item *somSelf,
                       Environment *ev,
                       CosStream_StreamIO* stream,
                       CosLifeCycle_FactoryFinder* ff)
{
  ItemData *somThis = ItemGetData(somSelf);
  ItemMethodDebug("Item","internalize_from_stream");

  if (!((somStream_StreamIO*)stream)->already_streamed(ev,
                         (CosStream_Streamable*)somSelf,_Item)) {
    if (somThis->description)
      SOMFree(somThis->description);
    somThis->description = stream->read_string(ev);
    somThis->cost = stream->read_float(ev);

Item_parent_somStream_Streamable_internalize_from_stream(somSelf,
                                         ev,stream,ff);
  }
}
```

The **internalize_from_stream** method is very similar to the **externalize_to_stream** method. It is important to free any memory allocated for the attributes before they are read from the **somStream::StreamIO**. The memory returned from the **read_string** method is owned by the caller (according to CORBA memory management semantics) so it does not need to be copied.

This sample object does not contain any references to other objects. For a discussion about writing and reading object references see “How to Externalize Objects with References” on page 11.

How to Initialize Streamable Objects

When you create a streamable object, you should initialize it using the **init_for_object_creation** method. This method initializes the identity related attributes of the object. If you use, for example, the **somDefaultInit** method instead, then the object will still be usable, but the first time an identity related method (such as **is_identical**) is used, the **init_for_object_creation** method will be invoked.

How to Use a Stream

The **CosStream::StreamIO** class (see “Object Externalization Service Class Diagram” on page 7) is abstract, i.e. it does not implement any methods. The **somStream::StreamIO** class declares the IBM extensions to **CosStream::StreamIO**, and it is also abstract. The

somStream::MemoryStreamIO class is an implementation that uses a contiguous memory block as the buffer to store the data.

The following program example creates and initializes a streamable object of the **Item** class. It then creates a memory stream, externalizes an **Item** to the stream, and internalizes the data into another **Item** object.

```
#include <somd.xh>
#include <somestrm.xh>
#include <item.xh>

void main()
{
  Environment ev[1];
  Item *item1, *item2;
  somStream_MemoryStreamIO *strm;

  SOM_InitEnvironment(ev);
  SOMD_Init(ev);
  item1 = (Item*)(void*)_Item->somNewNoInit();
  item1 = (Item*)(void*)item1->init_for_object_creation(ev);
  item1->_set_description(ev,"Coffee");
  item1->_set_cost(ev,3.59);
  strm = new somStream_MemoryStreamIO;
  item1->externalize_to_stream(ev,strm);

  item2 = (Item*)(void*)_Item->somNewNoInit();
  item2 = (Item*)(void*)item2->init_for_object_creation(ev);
  item2->internalize_from_stream(ev,strm);
}
```

This example creates a second **Item** and internalizes it from the stream so that its state is a copy of the original **Item**. You can use the **_get_buffer** method on the stream to get a copy of the memory buffer so that you can store the data into a file or transmit it to another process. You can use the **_set_buffer** method to restore the stream contents so that you can internalize objects from it.

The **somStream::MemoryStreamIO** implementation stores the data in the format native to the process in which the buffer resides. The code page of the character data, the endian[3] format, and the floating point format can vary. If the buffer is sent to another process that is using a different code page, endian format, or floating point format, the results are unpredictable.

DSOM Considerations

The client application needs to be a DSOM server (peer-to-peer) if any of the streamable objects or the stream itself exists in the client process. And, because of the callback situations that can arise between the stream object and the streamable objects, you should register your servers as multi-threaded servers (**regimpl** option **-m** on).

3. The endian format determines the order of the bytes for numeric data types: short, unsigned short, long, unsigned long, float, and double. Big endian format stores the bytes from high-order to low-order. Little endian format stores the bytes from low-order to high-order. Most Intel-based hardware uses little endian.

How to Externalize Objects with References

It is often the case that your streamable object contains references to other objects. If these references are a vital part of the object state, you need to decide how to handle them in the **externalize_to_stream** and **internalize_from_stream** methods. You have basically two choices, either the object itself can manage the references, or it can rely on an instance manager to manage the references.

Managing References using the Object

In some cases, it is possible for the object itself to manage its references within the **externalize_to_stream** and **internalize_from_stream** methods. To do this the object must be able to write and read the data of the referenced objects. For instance, if the referenced objects are streamable, the **externalize_from_stream** method of the object can simply call **externalize_to_stream** on each of its references, and **internalize_from_stream** can do likewise. If the referenced objects are not streamable, the object may be able to use get/set methods to write/read their data itself. In the **internalize_from_stream** method, the object must know the class of each referenced object, or have some other means to re-create each referenced object.

One limitation of this design is that the referenced objects may not be shared by other streamable objects. If two streamable objects have a reference to the same object, after they are externalized and internalized, the two streamable objects would no longer share the object; they would each have their own copy of it. If several streamable objects contain references to one another in such a way that a cycle is formed, then calling the **externalize_to_stream** method would cause an infinite loop.

Another limitation of this design is that the deep vs. shallow semantics of the externalization is hard-coded into the object. It is not possible to sometimes externalize a deep copy of an object, and to sometimes externalize a shallow copy of the same object.

Managing References using an Instance Manager

A much more flexible design is acheived when the streamable object uses an instance manager object (or service) to handle the object references it contains.

The **CosExternalization::Stream** class, as specified in the OMG specification, can act as an instance manager. In this design, the streamable object uses the **externalize** method to write any contained references[4]. The **externalize** method maintains a table of every object written to the stream so that if the same object is written more than once, a “repeated reference” number is written to the stream. This **internalize** method works similarly. This solves the problem of shared references and cyclical references. Furthermore, by supplying alternate implementations of the **CosExternalization::Stream** class, the deep vs. shallow semantics can be controlled independently of the streamable objects. The **CosExternalization::Stream** class is not provided in this release.

The **somOS::Server** class can act as an instance manager. In this design, the streamable object **externalize_to_stream** method uses the **object_to_string** method to convert each of its contained references to strings, and it uses the **write_string** method to write each

4. Actually the streamable object call the **write_object** method of StreamIO which simply forwards the call to the **externalize** method of the Stream.

one to the stream. Similarly, the **internalize_from_stream** method uses the **read_string** method and the **string_to_object** method to restore each of its references. This solves the problem of shared references and cyclical references. However, it only provides shallow semantics. The references must be persistent. See Chapter 5, Object Services Server on page 35 for information about persistent references.

A third alternative is that you provide your own instance manager implementation.

When the Stream is Implicitly Reset

The **reset** method on **somStream::StreamIO** sets the buffer position to the beginning of the buffer. If the buffer has unused memory, then it shrinks it using **SOMRealloc**. The stream is implicitly reset whenever any of the following happens

- a read follows a write
- a write follows a read
- the **set_buffer** method is called
- the **clear_buffer** method is called

You only need to explicitly call the **reset** method if you want to write or read the same data again, for instance, in error recovery.

Variances from the OMG Specification

The implementation of the Externalization Service has required some minor refinements and variances from the *OMG Object Externalization Service* specification.

- Some classes such as **somStream::StreamIO** inherit from **SOMObject** instead of from nothing. This is a requirement of the IDL compiler.
- The **CosStream::StreamIO** class does not have **write_graph** and **read_graph** methods, because the Relationship Service is not provided.
- The **CosStream::StreamIO** class does not have **write_object** and **read_object** methods, because the Life Cycle Service is not provided.

Chapter 3. Object Identity Service

This section describes the motivation, applicability, and consequences of using the SOMobjects Object Identity Service.

somOS::ServiceBase Class

The **somOS::ServiceBase** class, when mixed in with other classes through inheritance, provides the notion of *identity*.

Intent

Identity allows object instances to be distinguished from one another. Identity, in this context, refers to the exact same instances as opposed to objects that may have the exact same state (equality) but actually be two different instances of the same class.

Motivation

A basic premise in object-oriented technology is that objects are metaphors for real-life things (for example, an instance of the Dog class is a software representative of a particular dog). To the extent needed to differentiate real-life things (is this dog and that dog the same dog?), it also is necessary to tell their object-oriented representatives apart.

Object references are, for most situations, inadequate for doing object comparisons.

Comparing object references:

- Relies on the Object Request Broker (ORB) implementation.
- Violates encapsulation by allowing clients to depend upon non-interface properties of objects.

Object references are intended to be opaque values that are constructed by a particular ORB implementation. The object reference produced for an object by one ORB may be different than one constructed by another ORB for that same object. Conversely, two different ORBs could produce the same reference for two different objects.

In the case where only remote objects or proxies are involved, it is possible to have two different proxies that refer to the exact same object instance. Comparing object references in this case returns False even though they refer to the same instance.

Performance and Efficiency

Other considerations, especially where remote or distributed objects are involved, are performance and efficiency.

For instance, imagine a collection that supports the **identify_any** and **identify_all** operations that take in an object reference as an argument and return the label associated with the object in its collection. The collection must search through all of the objects in its collection and compare each object with the object that was passed in. If the two objects are the same object, it returns the label that is bound with that object in the collection. The **identify_any** operation returns at this point, while the **identify_all** operation proceeds to the next object in its collection until the entire list has been checked.

An issue that the collection must deal with is that it should not have to go out and touch each and every individual object to perform the object comparison. Doing so has a significant potential performance impact, both in terms of communication latency going across the network, as well as the potential overhead of resurrecting dormant objects that 99+ percent of the time are not the desired objects.

Another requirement is that if caching is employed to minimize any performance hit, cached information should remain as small as possible (for example, caching 16 bytes for 100,000 objects requires 1.5 Mbytes of additional information in the collection).

Ideally, establishing identity should minimize the need for remote method calls and occur as close to the client as possible. Any additional information cached near the client to help establish identity should be as small as possible.

Solution Scenario

The **somOS::ServiceBase** class offers a solution. When mixed in with another class whose instances need to be identifiable, it provides a unique identity for each object instance. It provides a method used to report whether two objects are identical, as well as an attribute that, when cached near the client, can be used for quick, first-order identity comparisons near the client. This service can be used by local and distributed SOM objects.

When an instance of a descendant class of the **somOS::ServiceBase** class is created and the **init_for_object_creation** method is called, a random number is generated and stored in the **constant_random_id** attribute. The **constant_random_id** is not guaranteed to be unique; its actual value is not as important as the fact that it is set at (or near) object creation time and then never changes throughout the lifetime of the object. This value is used as a first-order approximation of whether two objects are the same object; therefore, the more random the value, the more efficient its use.

When an object is added to the collection and the object being added is identifiable, the collection can get the **constant_random_id** from that object and store the information in the collection. This serves as a cache for the object identity and results in fewer traversals across the network to the objects.

When the **identify_any** or **identify_all** operation is invoked on the collection, the following occurs:

1. The operation tests whether the passed object is identifiable; if not, the method request is terminated.
2. The **constant_random_id** is retrieved from the passed object.
3. The operation iterates through each object in its collection. For each object, if the object is identifiable, the collection compares the **constant_random_id** from the passed object with the **constant_random_id** cached for the objects in the collection. If the values are the same for both objects, the **is_identical** operation is invoked on the passed object passing in the object reference for the object in the collection. The results of this operation determine whether the two objects are the same.
4. For any object in the collection that is not identifiable, the collection ignores that object and skips to the next one. Because the object is not identifiable and the passed object is, this is a partial indicator that they are not the same object; they at least have different metadata. If the object is not identifiable, there is no known way for the collection to deduce its identity.
5. When both objects are identifiable, the **is_identical** operation could have been invoked on either object passing in the other object as an argument. The **is_identical** operation is invoked on the passed object because the passed object presumably had to be resurrected to get the **constant_random_id** from it at the beginning of the process. There is a good chance that it is active anyway by virtue of being the subject of the **identify_*** operation.

6. The **identify_any** terminates at the first object that matches. The **identify_all** continues through the entire list, finding any objects that match the passed object.

Applicability

If you need to consistently compare the identity of an object instance locally or remotely,mix-in the **somOS::ServiceBase** class.

Members of collections or containers are good candidates for being identifiable objects because most list-searching operations require an identity comparison.

Consider mixing in a base class, if one exists, to give all objects of a certain type standard identity characteristics.

Structure

Figure 2 provides the class diagram for **somOS::ServiceBase** in the object modeling technique (OMT) notation.

somOS::ServiceBase inherits from **CosObjectIdentity::IdentifiableObject**, which is an abstract class. In other words, only the **IdentifiableObject** interface is inherited; no implementation is inherited. **somOS::ServiceBase** overrides all of the inherited operations from **IdentifiableObject** because it provides the actual implementation. The **constant_random_id** is tagged with the no-data modifier in **CosObjectIdentity::-IdentifiableObject** because **somOS::ServiceBase** provides the instance data for that abstract attribute.

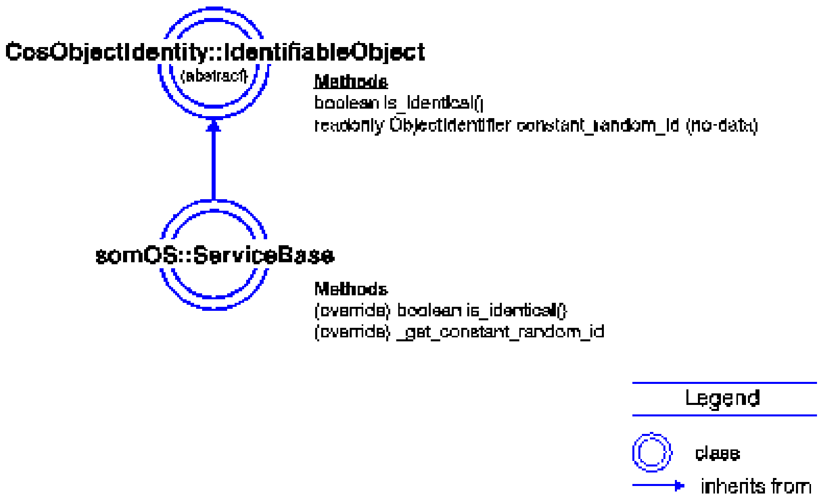

Figure 2. somOS::ServiceBase Class Diagram

Considerations

Some considerations of the **somOS::ServiceBase** mix-in class are as follows:

- A benefit is that it is easy to inherit the interface and implementation from **somOS::-ServiceBase** to provide objects with the notion of identity. There is no need to override any of the operations; **somOS::ServiceBase** provides a complete, ready-to-use implementation.

- Deciding whether or not to cache the **constant_random_id** is a speed-versus-size trade-off. Consider the extra memory it requires per object versus the performance penalty of a potentially remote **is_identical** method invocation. If there are not many objects subject to identity operations or they occur rarely, it may not be necessary to cache it.

Chapter 4. Naming Service

This chapter discusses the Naming Service, which gives users and programmers the ability to refer to objects by name. These names can have a syntax and form that are readily understood and manipulated by human beings. With the Naming Service, you can organize computing resources so that they can easily be found, identified, and categorized either in context or by explicit characterization.

Introduction

The Naming Service is the principle mechanism by which clients can locate objects they intend to use. The interface is based on the *Common Object Services Specification Volume 1* (OMG Document Number 94-1-1), with enhancements to support *properties* and *constraint*-based search.

The Naming Service provides methods to support binding (associating) a name to an object and later *resolving* the bound name, setting and getting properties on names, and performing searches through *filters*. The Naming Service provides filters that use properties and constraint expressions. Filters allow clients to search for objects with certain characteristics. For example, one could search for objects whose size is less than 24K bytes.

The *naming context* is central to the Naming Service. A naming context is a collection of name/object associations (bindings). The fundamental concept in the Naming Service is that the entire name space is composed of naming contexts bound together to form a directed graph. The naming context is itself an object; therefore, creation of an association between two naming contexts is just a matter of binding one naming context (object) with a name into the other naming context object. Naming contexts reside in a naming server. Therefore, a Naming Service is simply a graph of naming contexts that reside in one or more DSOM server processes. Client applications manipulate name bindings by means of the naming context APIs through remote method calls.

Our enhancements introduce the notion of *properties*, which are *name-value* pairs. The property name is a CORBA string, and the value can be of any CORBA type, including constructed types such as structures and sequences. Therefore, you can create any arbitrary property and assign it to a binding in the Naming Service.

The enhancements also introduce *index* methods with which you can create *indexes* on specific properties in the context. Although an index increases the size of a context, it also improves the performance of any searches that involve that property. Therefore, if there is a tendency to use the same property, an index for that property may be helpful. You can create indexes at any time before or after you add bindings that contain a particular property. If bindings that contain the property already exist when the index is created, the index is built from the existing binding information. Likewise, if bindings that contain an indexed property are added later, the index is updated with the addition.

Abstract and Concrete Interfaces

The Naming Service consists of a hierarchy of interfaces with both abstract and concrete implementations, which together provide an OMG-compliant Naming Service along with IBM Naming Extensions that enhance the Naming Service functionality. Refer to Figure 3 and Figure 4.

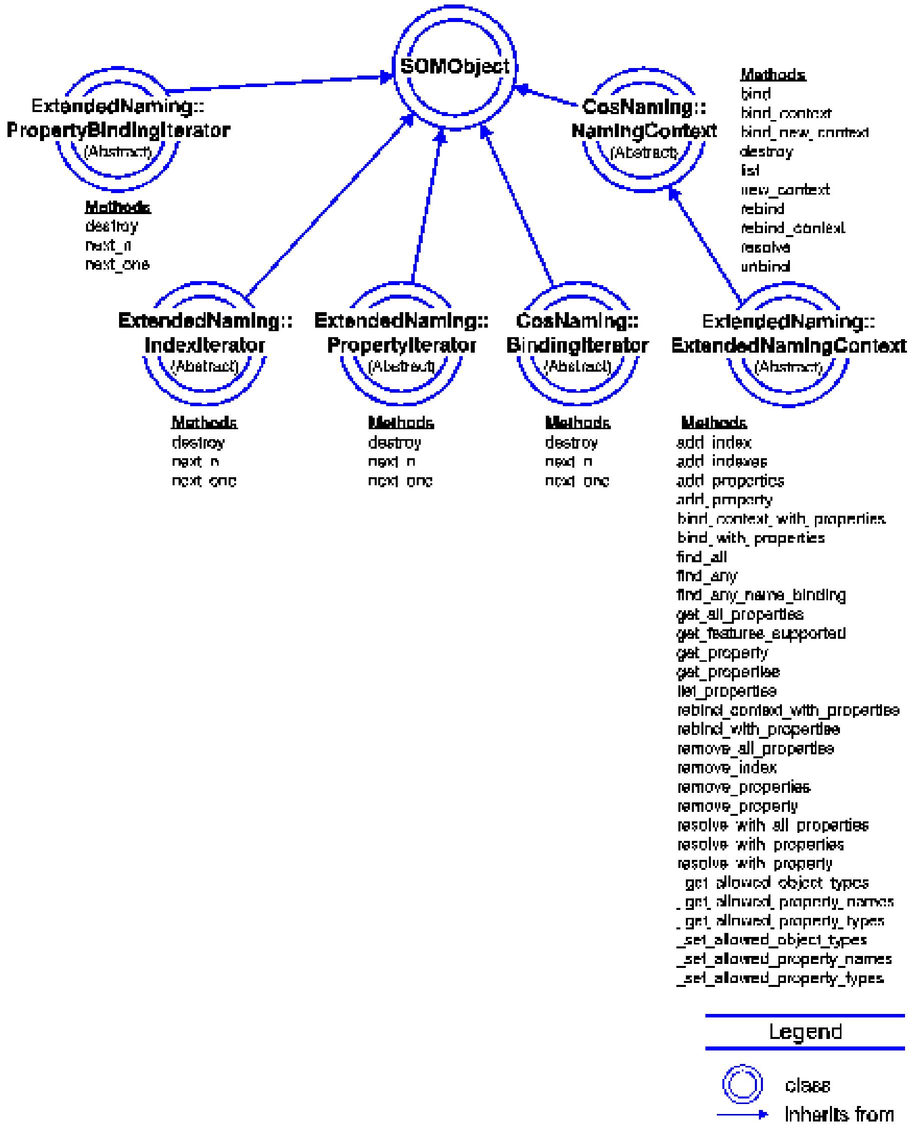

Figure 3. Derivation for SOMobjects 3.0 Naming Service, Part One

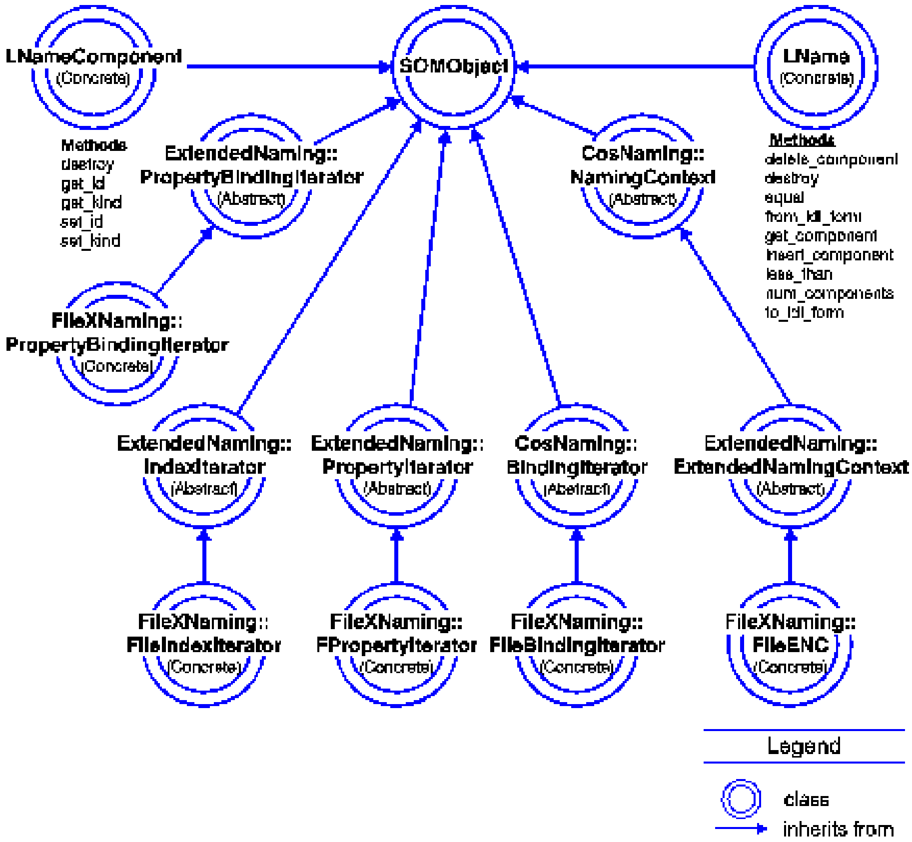

Figure 4. Derivation for SOMobjects 3.0 Naming Service, Part Two

The Naming Service includes enhancements to the OMG Naming specification. These enhancements provide for increased control over naming context objects, primarily *Property* support. Property support is the ability to manipulate dynamic name-value pairs associated with the name-object bindings within a naming context. The IBM extensions to the OMG Naming Service are defined within the **ExtendedNaming** Module, and they are provided as abstract classes, along with the OMG **CosNaming** Module's interfaces. Users can subclass these abstract interfaces to provide their own concrete implementations, if desired.

The **ExtendedNamingContext** interface is presented as an abstract class in the **ExtendedNaming** module. An implementation that is both lightweight and supports all features (such as properties and indexes) is provided in **FileXNaming::FileENC**. The **FileXNaming::FileENC** implementation provides persistence of the naming graph by creating files in the directory that the environment setting **SOMDDIR** points to. (For more information on abstract classes, see *Programmer's Reference for Abstract Interface Definitions*.)

Some of the implementations of the **ExtendedNamingContext** interface may not support all features, such as *Properties*, *Sharing*, and *Indexes*. A **get_features_supported** method is introduced in the **ExtendedNamingContext** interface to allow users to efficiently determine the features that an implementation supports.

Concepts about Naming

A Naming Service is a graph of naming contexts that reside in one or more DSOM server processes. Naming contexts, Names, and Properties are some of the important concepts discussed.

Naming Contexts

Naming contexts are modeled after folders or directories, and names are modeled after documents or files. The naming context is itself an object that contains name-object associations (bindings). Because the naming context itself is an object, it can be bound to another context, thus creating a name graph. Figure 5 depicts an example name graph. In this example, the shaded circles represent Naming Contexts. The Leaf nodes are objects bound into naming contexts. Naming context ***nc2*** is bound in the root as "printers." Objects ***o1***, ***o2*** and ***o3*** are bound as ***p1***, ***p2*** and ***p3*** in ***nc2***.

Given an initial root naming context, clients manipulate the naming graph by operating on the naming context object. The **ExtendedNamingContext** interface allows the following operations:

- Creating and deleting contexts
- Binding and unbinding names
- Resolving names
- Listing bindings
- Adding, updating, and deleting properties
- Resolving names with properties
- Searching based on predicates
- Listing properties associated with a name

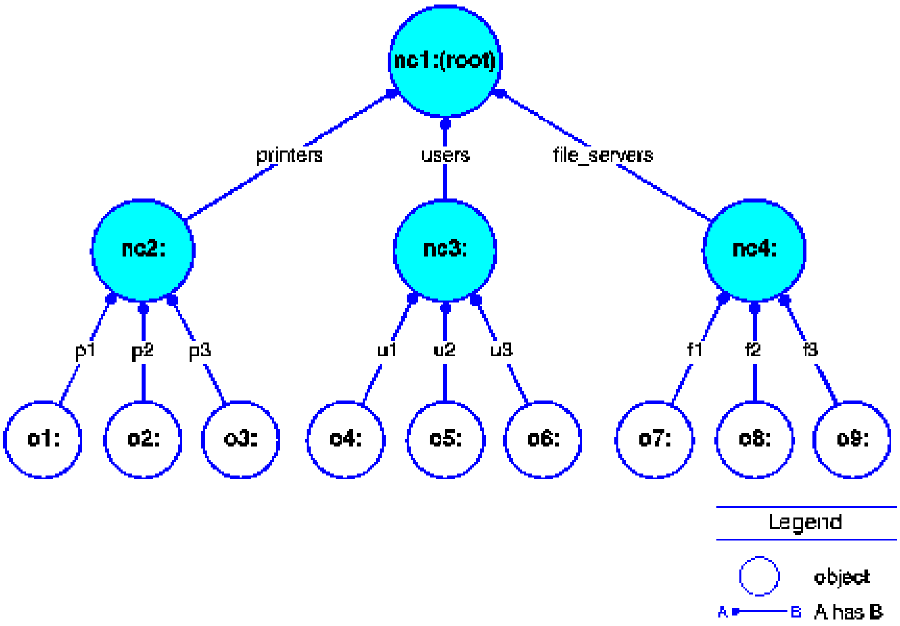

Figure 5. Example Name Graph

Names

The *Common Object Services Specification Volume 1* (OMG Document Number 94-1-1) defines a *name* as an ordered sequence of *name components*. A Name component is an IDL struct with the two elements **id** and **kind**. The following is the definition of a Name defined in the **CosNaming** module:

```
typedef string Istring;
struct NameComponent
     Istring id;
     Istring kind;
};
typedef sequence <NameComponent> Name;
```

A name that has a single component is a *simple name*. A name that has multiple components is a *compound name*. Names are always specified relative to the naming context on which the method is performed. The differences between these two types of names are illustrated in the name-tree example in Figure 6, where T is a name component in naming context ***nc1*** referring to the binding of naming context ***nc2***. (In this example, ***nc1***, ***nc2***, ***o1*** and ***o3*** are object identifiers, not names. They are provided to help clarify the relationship of the objects involved in the example.) The compound name <T;U> is relative to ***nc1*** traversing through ***nc2*** to ***o1***. Notice that both T and U are name components relative to their respective naming contexts; ***nc1*** in the case of T and ***nc2*** in the case of U. Compound names are composed of multiple name component. Notice also that the starting context is important: T relative to naming context ***nc1*** refers to ***nc2***; whereas, T relative to naming context ***nc2*** refers to ***o2***.

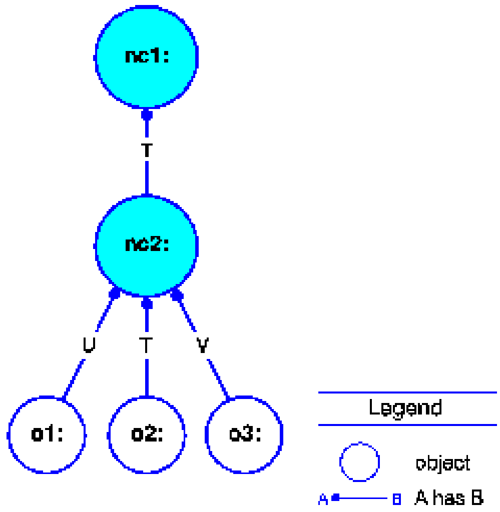

Figure 6. Component and Compound Name Examples

A name component has two parts: an **id** field and a **kind** field. The intent is to separate the name of the object from the semantics of the object type. For example, two printer objects, “moe” and “curly,” could have the **kind** field set to “printer”. The Naming Service does not interpret or manipulate these values in any way. The unique identification of a name requires both the **id** and the **kind** fields. Figure 7 illustrates this point. Although the **kind** fields of objects ***01*** and ***02*** are identical, the Naming Service treats name1 and name2 as two distinct names.

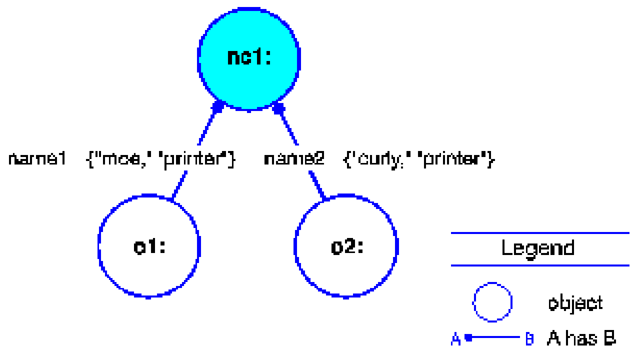

Figure 7. A Name Uniquely Identified by id and kind Fields

Properties

A property consists of a **property_name** and a **property_value**. The string *property_name* names a property, and *property_value* is an *any* (the value assigned to the property). Clients can manipulate properties individually or in batches using a **PropertyList**. The following are the definitions of **Property** and **PropertyList** as defined in the **ExtendedNaming** module:

```
typedef struct PropertyBinding_struct {
     CosNaming::Istring property_name;
     boolean sharable;
} PropertyBinding;
```

```
typedef struct Property_struct {
     PropertyBinding binding;
     any value;
} Property;
typedef sequence<Property> PropertyList;
```

Note: In this example, **PropertyBinding** contains the flag **sharable** that is no longer in use.

Roots and Namespaces

SOMobjects produces a default name space at configuration time. That name space has two roots: a local root context and a global root context. Actually, there is a local root context in each host machine, but only one global root context in the workgroup.

SOMobjects organizes the name tree on the same principles as the UNIX file system; that is, a single naming context is designated in every host as the local root. Because everything is contextual, all absolute names are resolved from this root. In addition, SOMobjects designates another naming context as the root of the global name tree.

The global name tree is a distributed name tree and is shared among all hosts in the workgroup. The root context of the global name tree is bound into each local root context as the "**.:**" name.

The default tree trunk for the name space provided with SOMobjects is depicted in Figure 8. This name tree is constructed when SOMobjects is configured (using **som_cfg**). You can provide additional name contexts within or below this tree for your purposes, create entirely independent name trees, or modify the tree provided. However, if you add your own independent name tree, consider how other applications can discover that tree if they have to refer to anything in it. Also, if you discard or change the name tree provided by SOMobjects, certain distinguished objects are contained in the tree, and other services might be affected if they are unable to locate those objects.

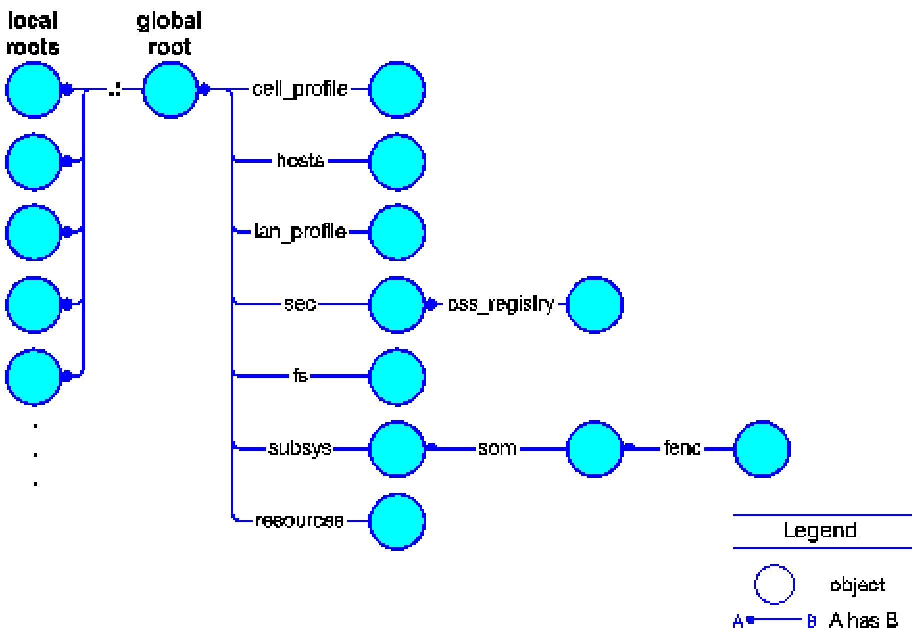

Figure 8. Name Space Structure

Finding the Local Root Naming Context

Before applications can begin using the Naming Service, they must get a reference to a naming context object. One can obtain an initial reference to a naming context object using the ORB interface. The ORB interface supports methods that list services and obtain the initial reference to the service. The **resolve_initial_references** method takes in an *ObjectId* (a string) and returns an object reference. Clients are responsible for narrowing the returned reference.

```
#include <somnm.h>
#include <somd.h>

ExtendedNaming_ExtendedNamingContext rootNC;
Environment ev;
...
rootNC = (ExtendedNaming_ExtendedNamingContext)
 ORB_resolve_initial_references(SOMD_ORBObject, &ev, "NameService");
```

The returned root naming context provides a starting point for applications to begin using the Naming Service. You can obtain other naming contexts by resolving their names on the root naming context.

When you are finished using the result of **resolve_initial_references**, invoke **release**, not **somFree**.

Using the Bind Process to Register with the Naming Service

Once the application has resolved a naming context, it can use binding methods to name objects in a naming context. The **ExtendedNamingContext** interface supports the following eight methods to do binding:

- **bind**
- **rebind**
- **bind_context**
- **rebind_context**
- **bind_with_properties**
- **rebind_with_properties**
- **bind_context_with_properties**
- **rebind_context_with_properties**

The **bind** and **bind_with_properties** methods name an object in a naming context. Here, a binding that names *obj* as *ashoo* is created in *nc*.

```
#include <somnm.h>

ExtendedNaming_ExtendedNamingContext nc;
Environment ev;
CosNaming_Name name;
Person obj;
...
name._length = name._maximum = 1;
name._buffer = SOMMalloc(sizeof(CosNaming_NameComponent));
name._buffer[0].id = "Ashoo";
name._buffer[0].kind = NULL;

CosNaming_NamingContext_bind(nc, &ev, &name, obj);
```

The following code fragment not only creates a binding, but also associates two properties with the binding:

```
ExtendedNaming_PropertyList pl;

     /* build the property list */
pl->_maximum = 10;
pl->_length = 2;
pl->_buffer = (Property *)SOMMalloc (pl->_maximum*sizeof
     (ExtendedNaming_Property));
     /* property one */
pl->_buffer[0].binding.property_name = "colorOfEyes";
     /* set the property value */
pl->_buffer[0].value._type = TC_string;
ptr = (char **)SOMMalloc(sizeof(char *)) ;
*ptr = (char *)strdup("black");
pl->_buffer[0].value._value = (void *)ptr;
     /* property two */
pl->_buffer[1].binding.property_name = "pet";
     /* set the property value */
pl->_buffer[1].value._type = TC_string;
ptr = (char **)SOMMalloc(sizeof(char *)) ;
*ptr = (char *)strdup("Flakes");
pl->_buffer[1].value._value = (void *)ptr;

CosNaming_Context_bind_with_properties (nc,&ev,&name,inObj,pl);
```

The preceding methods raise an **AlreadyBound** exception if an object is already bound to the specified name. Only one object can be bound to a particular name in a context. If you want to replace the bound object with a different object for applications, use the **rebind** and **rebind_with_properties** methods. These methods unbind the name and rebind the name to an object passed as the argument:

```
SOMObject newObj;
CosNaming_NamingContext_rebind(nc, &ev, &name, newObj);
```

Because a naming context is also an object, it can be bound to another naming context. Use the **bind_context** and **bind_context_with_properties** methods to bind naming contexts. Naming contexts that are bound using these methods participate during the resolution of Compound Names. Naming contexts bound using **bind** and **bind_with_properties** methods do not participate in the name resolution process of compound names.

When compound names are passed as arguments to the **bind** methods, the resolution process first traverses the naming graph to the last naming context. The last component in the compound name designates a *simple name,* which is then bound to the leaf context. A **NotFound** exception is raised if any of the intermediate naming contexts cannot be resolved.

A bind method that is passed a compound name is defined as follows:

```
namingContext --> bind (<c1; c2; c3 ... ; cn>, obj)
          ==> (namingcontext -> resolve (<c1; c2; ... ; cn-1>))
                   --> bind (<cn>, obj)
```

Note: The semicolon character is simply a notation used here and is not intended to imply that names are sequences of characters separated by semicolons.

Resolving Names

The **ExtendedNamingContext** interface supports four methods to retrieve an object bound to a name in a given context: **resolve**, **resolve_with_property**, **resolve_with_properties** and **resolve_with_all_properties**. The **id** and the **kind** fields of the given name must exactly match the bound name. It is the responsibility of the application to narrow down the returned object to the appropriate type. The following example shows how a previously bound name, {“ashoo”, NULL} is resolved.

```
ExtendedNaming_ExtendedNamingContext nc;
Environment ev;
CosNaming_Name name;
Person retObj;
...
name._length = name._maximum = 1;
name._buffer = SOMMalloc(sizeof(CosNaming_NameComponent));
name._buffer[0].id = "ashoo";
name._buffer[0].kind = NULL;
retObj = (Person )CosNaming_NamingContext_resolve(nc, &ev,&name);
```

Names are always defined relative to a naming context. There are no absolute names. It is important to realize that resolving a compound name implies traversing more than one context in a naming graph.

In Figure 6, there are at least two ways to get to **o1**: by doing a compound **resolve** on **nc1** or by doing a simple **resolve** on **nc2**. The following code fragment does a compound **resolve** on the root:

```
name._length = name._maximum = 2;
name._buffer = SOMMalloc(sizeof(CosNaming_NameComponent)*2);
name._buffer[0].id = "T";
name._buffer[0].kind = NULL;

name._buffer[1].id = "U";
name._buffer[1].kind = NULL;

retObj = (Person )_resolve(root, &ev, &name);
```

The following code fragment does a simple **resolve** on **nc1** (which is bound as "U" in **nc2**):

```
name._length = name._maximum = 1;
name._buffer = SOMMalloc(sizeof(CosNaming_NameComponent));
name._buffer[0].id = "U";
name._buffer[0].kind = NULL;

retObj = (Person )_resolve(nc, &ev, &name);
```

The Naming Service imposes no policies on the partitioning of the name space. Applications can employ conventions so that they can meaningfully partition the name space to allow good load balancing. Later sections discuss various schemes to partition the name space.

The **resolve_with_property** method and its variations operate exactly like the **resolve** method in returning the bound object. In addition, however, they also return the associated property or properties.

Creating Contexts

To create a new naming graph or to extend an existing naming graph, you must first create a naming context. The Naming Service provides several principal ways to create a new context:

- Create an independent context object that you can later bind to an existing name tree using the **bind_context** method on an existing naming context object. The **new_context** method creates a new naming context in the same server as the context on which this method was run.

- Perform a **bind_new_context** method on an existing naming context. This creates a new context in the same server as the targeted naming context and binds it in the targeted naming context.

The following code fragment shows how to create a new naming context using the **new_context** method:

```
#include <somnm.h>
ExtendedNaming_ExtendedNamingContext *newNC;
Environment ev;
...
newNC = CosNaming_NamingContext_new_context(currentNC, &ev);
```

The **bind_new_context** method is a combination of **bind** and **new_context**. It is used to create a new context and then bind it to the context on which the method is being invoked. The following code fragment illustrates this:

```
CosNaming_Name name;
...
newNC = CosNaming_NamingContext_bind_new_context(currentNC,
                                                  &ev, &name);
```

Associating Properties to a Name Binding

With the current implementation of the Naming Service you can locate objects for use by looking for objects having certain external attributes. Applications can add these characteristics to a name binding through properties. Properties can be associated with bound naming context objects as well as the bound objects. Use the following structures to specify the properties for a binding:

```
typedef struct PropertyBinding_struct {
     CosNaming::Istring property_name;
     boolean sharable;
} PropertyBinding;

typedef struct Property_struct {
     PropertyBinding binding ;
     any value;
} Property;
```

Note: In this example, **PropertyBinding** contains the flag **sharable** that is no longer in use.

You can associate a property with a binding in two ways:

- When a binding is created, applications can associate properties with the bindings. This semantic is supported by the following methods: **bind_with_properties**, **rebind_with_properties**, **bind_context_with_properties** and **rebind_context_with_properties**.
- The second mechanism is through the **add_property** and **add_properties** methods.

Using these methods, clients can add properties after a binding has been created. If a property is already defined, the property value is overwritten. The property structure has an **any** defined as the property value. You must construct an **any** of the value before using these methods. The following example shows how to construct a property and then add it to a binding:

```
ExtendedNaming_ExtendedNamingContext nc;
Environment ev;
CosNaming_Name thisName;
Property prop;
long *ptr;
```

```
...
prop.binding.property_name = "latencyTime";
         /* set the property value */
prop.value._type = TC_long;
ptr = (long *)SOMMalloc(sizeof(long)) ;
*ptr = (long)9;
prop.value._value = (void *)ptr;

ExtendedNaming_ExtendedNamingContext_add_property (nc, &ev,
                                             &thisName, &prop);
```

The binding name “thisName” is not required to have been previously bound. If “thisName” does not exist, it is bound to OBJECT_NIL.

Properties are associated with Names, not with the bound objects. This means that an object can be bound with the same Name in two different contexts having the same property name and different property values. This is illustrated in Figure 9.

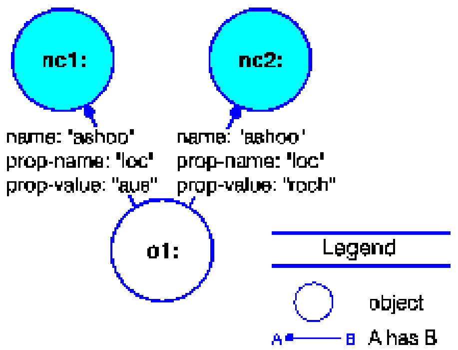

Figure 9. An Object Bound with the Same Name in Different Contexts

The **o1** object is registered with the Naming Service in naming context **nc1** as “ashoo” and has one property with the name “loc” and the value “aus”. The same object is registered under naming context **nc2** with the name “ashoo” and the property “loc”. However, the property value is “roch”.

Listing and Getting Property Values

With the **list_properties** method, applications can retrieve all the properties defined for a name. The following example code shows how you can do this:

```
ExtendedNaming_ExtendedNamingContext nc;
Environment ev;
CosNaming_Name Name;
ExtendedNaming_PropertyBindingList pbl;
ExtendedNaming_PropertyBindingIterator pbIterator = NULL;

...
pbl._maximum = pbl._length = 0;
pbl._buffer = NULL;
howMany = 10;

_list_properties (nc, &ev, &name, howMany, &pbl, &pbIterator);
```

The **list_properties** method returns at most “howMany” properties in `PropertyBindingList pbl`. If the name binding contains additional properties, a **PropertyBindingIterator** is returned with the additional properties. The iterator is a remote object that exists on the server in which the naming context that contains the bindings resides. If the name binding does not contain additional properties, OBJECT_NIL is returned.

With the returned iterator, you can iterate through the property list. You can iterate through the bindings using the **next_one** and **next_n** methods. After you have finished using the iterator, destroy it by running the **destroy** method.

The following example code demonstrates how to iterate through the property bindings and how to destroy the iterator:

```
ExtendedNaming_PropertyBinding propBind;
somPrintf ("No. of properties: %d\n", pbl._length);
for (i = 0; i < pbl._length; i++)
         printf ("[%s]\n", pbl._buffer[i].property_name);

              /* step through the iterator */
if (!_is_nil(pbIterator, &ev)) {
     while (ExtendedNaming_PropertyBindingIterator_next_one
                              (pbIterator, &ev, &propBind)){
         somPrintf ("[%s]\n", propBind.property_name);
     }
}
              /* done with iterator */
ExtendedNaming_PropertyBindingIterator_destroy(pbIterator, &ev);
```

The **ExtendNamingContext** interface also supports the following three methods for the retrieval of property values: **get_property**, **get_properties** and **get_all_properties**. A **PropertyNotFound** exception is raised if any of the properties cannot be found in the specified name bindings. The following is an example of usage of the **get_property** method:

```
Property prop;
...
_get_property(nc, &ev, &name, "pet", &prop);
```

The returned **Property prop** contains the property name and its value.

Searching the Name Space

One approach to providing filters is through properties and constraints. Client applications can use constraint expressions to describe the characteristics of the bound object they are seeking. Constraints are expressed in a *Constraint Language*, which provides operators and symbols that allow complex building expressions involving properties and their values to be built. The BNF for the constraint expression is provided in “BNF for Naming Constraint Language” on page 31.

Three methods on the **ExtendedNamingContext** interface provide the search functionality: **find_any**, **find_all** and **find_any_name_binding**. These methods accept a constraint string that serves as the predicate for the search. Although the constraint grammar is quite flexible, it has two limitations:

- Property names can contain only standard alpha-numeric characters (plus a number of special characters, but no spaces).
- The operators can operate only on basic data types.

Although you can store complex data structures and ad hoc property names in a name binding, you can perform searches only on simple IDL data-types and well-formed property names.

In the following example, a constraint describes an object that **cost**s less than $5, a **MachineType** of `PowerPC`, and created on **Server** `Moe`:

```
constraint = "cost < 5 and MachineType == 'PowerPC' and Server ==
```

```
          'Moe'"
```

Assuming that property values are defined for these property names, you can use the **find** methods to locate the name bindings that satisfy the specified criteria. The **find_any** method returns the first bound object that satisfies the constraint in the context this method is run on in a naming context not more than the specified distance from the target naming context. The search is not *deterministic* in the sense that multiple invocations does not always return the same result. If none is found, a **BindingNotFound** exception is raised, and OBJECT_NIL is returned.

```
ExtendedNaming_ExtendedNamingContext nc;
Environment ev;
string constraint;
unsigned long distance;
distance = 2;
constraint = "MachineType == 'PowerPC' and Frequency < 65";
outObj = _find_any(nc, &ev, constraint, distance);
```

There is also support to query and get all bindings that satisfy the constraint. This method returns a **CosNaming::BindingIterator** object if there are more qualifying bindings than "howMany" are found. The following example illustrates this. If more than 100 bindings satisfy the specified constraint, 100 bindings are returned in “bl” and the remainder in the iterator “bi”. The iterator resides in the server process.

```
howMany =100;
_find_all(nc, &ev, constraint, distance, howMany, &bl, &bi);
```

A **BindingNotFound** exception is raised if no name-bindings are found to satisfy the given constraint.

The **ExtendedNamingContext** interface also introduces some index methods with which you can create indexes on specific properties in the context. Indexes can greatly improve the performance of searching in a context. However, maintaining an index has some negative impact on the performance of setting property values and requires additional storage. Therefore, creation of indexes is left under user control. Clients can use the **add_index**, **remove_index** and **list_indexes** methods to manipulate indexes.

The Names Library

It is expected that the representation of names evolves to accommodate other services. To allow names to evolve without affecting existing clients, a *names library* is provided. The names library supports the two interfaces **LNameComponent** and **LName**. These interfaces implement names as pseudo-objects. Because pseudo-objects cannot be passed across IDL interfaces, methods are provided to convert a name pseudo-object to a structure, and a structure to a name pseudo-object.

The following example demonstrates how you can use the names library to build and manipulate names. This example builds a name with two components: ({"usr", "dir"}, {"lib", "dir"})

```
#include <somnm.h>
LName anLName;
CosNaming_Name aName;
LNameComponent *lnc;
Environment ev;
ExtendedNaming_ExtendedNamingContext *nc;
CosNaming_Binding bnd;
...
     /* create an lname pseudo-object */
anLName = create_lname();
```

```
     /* create the first name component object */
lnc = create_lname_component();
_set_id(lnc, &ev, "usr");
_set_kind(lnc, &ev, "dir");

     /* insert first component */
_insert_component(anLName, &ev, 0, lnc);

     /* create the second name component object */
_set_id(lnc, &ev, "lib");
_set_kind(lnc, &ev, "dir");

     /* insert second component */
_insert_component(anLName, &ev, 1, lnc);

     /*   cannot use anLName as an argument to any of the
          Name Service apis. Need to convert the pseudo-object
          created into a name structure */
aName = _to_idl_form(anLName, &ev);

     /* invoke a naming api */
outObj = _resolve(nc, &ev, &aName);
```

Here is a fragment of code that creates a name pseudo-object.

```
     /* invoke another method that returns a name structure */
_find_any_name_binding(rootNC, &ev, constraint, (unsigned
                                               long)1, &bnd);

anLName = create_lname();
_from_idl_form(anLName, &ev, &(bnd.binding_name));

     /* print the library name object */
somPrintf("<");
for(i=1;i<=numComps;i++) {
     lnc = _get_component(anLName, &ev, i);
               /* print id  and kind */
     somPrintf("[%s/%s]", LNameComponent_get_id(lnc,ev),
                   LNameComponent_get_kind(lnc,ev));
     _somFree(lnc);
     if (i<numComps)
          somPrintf(";");
}
```

The **LName** and the **LNameComponent** objects are transient. When applications use **create_lname** or **create_lname_component** functions, the library name objects are created locally and not in the name server. This means that the lifetime of these objects is limited by the lifetime of the client process that created them.

BNF for Naming Constraint Language

The Naming Service allows searches based on properties attached to a name object binding. Service providers register their service and use *properties* to describe the service offered. Potential clients can then use a constraint expression to describe the requirements that service providers must satisfy. Constraints are expressed in a constraint language. Using the constraint language, you can specify arbitrarily complex expressions that involve property names and potential values.

The constraint language described below is excerpted from Appendix B of the *COSS Life Cycle Services* specification. It has been slightly modified to support future enhancements.

```
ConstraintExpr   : Expr
                 ;
```

```
Expr                :   Expr "or" Expr
                    |   Expr "and" Expr
                    |   Expr "xor" Expr
                    |   '(' Expr ')'
                    |   NumExpr Op NumExpr
                    |   StrExpr Op StrExpr
                    |   NumExpr Op StrExpr
                    ;

NumExpr             :   NumExpr "+" NumTerm
                    |   NumExpr "-" NumTerm
                    |   NumTerm
                    ;

NumTerm             :   NumFactor
                    |   NumTerm "*" NumFactor
                    |   NumTerm "/" NumFactor
                    ;

NumFactor           :   Num
                    |   Identifier
                    |   '(' NumExpr ')'
                    |   '-' NumFactor
                    ;

StrExpr             :   StrTerm
                    |   StrExpr "+" StrTerm
                    ;

StrTerm             :   String
                    |   '(' StrExpr ')'
                    ;


                    ;
Op                  :   "==" | "<=" | ">=" | "!=" | "<" | ">"
                    ;
Identifier          :   Word
                    ;
Word                :   Letter { AlphaNum }+
                    ;
AlphaNum            :   Letter
                    |   Digit
                    |   "_"
                    ;
String              :   "'" { Char }* "'"
                    ;
Num                 :   { Digit}+
                    |   { Digit}+ "." { Digit}*
                    ;
Char                :   Letter
                    |   Digit
                    |   Other
                    ;
Letter              :   a | b | c | d | e | f | g | h | i
                    |   j | k | l | m | n | o | p | q | r
                    |   s | t | u | v | w | x | y | z | A
                    |   B | C | D | E | F | G | H | I | J
                    |   K | L | M | N | O | P | Q | R | S
                    |   T | U | V | W | X | Z
                    ;
Digit               :   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
                    ;
Other               :   <Sp> | ~ | ! | @ | # | $ | % | ^ | &
```

```
                              |    *  |  (  |  )  |  -  |  _  |  =  |  +  |  [  |  {
                              |    ]  |  }  |  ;  |  :  |  "  |  \  |  |  |  ,  |  <
                              |    .  |  >  |  /  |  ?
                              ;
    Sp                        :    " "
                              ;
```

The following precedence relations hold in the absence of parentheses, from lowest to highest:

- *or* and *xor*
- *and*
- *not*
- + and -
- * and /
- Otherwise, left-to-right precedence

The following are some example constraints:

```
(1) name == 'ashoo'
(2) name == 'ashoo' and pet == 'flakes'
(3) Fee <= 5 or LowFreq >= 20
(4) DeviceType == 'Car' and Cost < 30000 and
           color == 'white' and Year > 1990
```

Chapter 5. Object Services Server

The Object Services Server is responsible for instituting persistent object references and managing object metastate on behalf of the SOMobjects object services. It handles the majority of this responsibility transparently to client applications or class programmers. However, there are provisions in the server that you can or must be involved in, depending upon the nature of your application and managed objects.

For instance, if your objects are not created with a factory, you can ensure that persistent references are automatically created for your objects by sub-classing from **somOS::ServiceBasePRef.** Likewise, if you want your class to differentiate between object creation and object reactivation during initialization or similarly between object passivation and object destruction, override the **somOS::ServiceBase** initializers and destructors. Finally, if your client application wants to manage object passivation, invoke **passivate_object** at the right time.

For the most part, if you are only writing client applications, your interest is limited. You only have to ensure that the Object Services Server is specified in the Implementation Repository for the server process where you want to create or operate on managed objects, and then to follow the Life Cycle model prescribed by the server. If you are a class programmer specializing a managed object class, you probably should learn about the Object Services Server and any special conditions that you must support in your class implementation.

Before reading this chapter, it is important that you read and understand the DSOM framework and its concepts of server processes. Also, it is a good idea to read and understand the concept of a managed-object as described in “Managed Objects” on page 2. In particular, remember that a managed object is merely any object subject to the SOMobjects object services.

Overview

The Object Services Server is a specialization of the DSOM framework that supports the specific needs of the SOMobjects object services. As a server, it participates with the DSOM object adapter to export and import object references. With its unique knowledge of the object services frameworks, it participates in the handling of object metastate.

The main elements of the Object Services Server are:

- the server-class
- an object services base class
- the server program

At the machine, the Object Services Server comes into existence as a server process when the server program is executed. Essentially, the Object Services Server (program) contains the Object Services Server object and managed object. The Object Services Server exploits the DSOM framework and, therefore, relates to certain other DSOM components; specifically, to the DSOM object adapter and DSOM object references. The inheritance relationships for the Object Services Server components are depicted in Figure 10. The Object Services Server is a specialization of **SOMDServer**. All managed objects within the server should be derived from an object service mix-in class, which, in turn, should be derived from **somOS::ServiceBase**. An object reference is an instance of **SOMDObject**. And the DSOM Object Adapter is an instance of SOMOA, which is derived from the Basic Object Adapter (BOA).

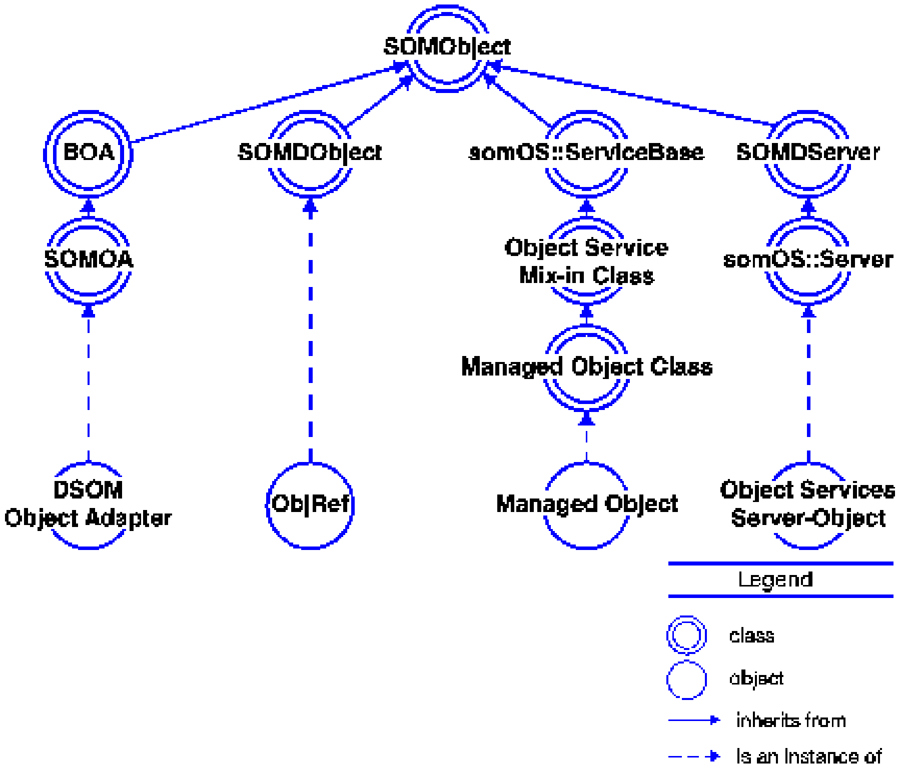

Figure 10. Object Services Server is a Specialization of the DSOM Framework

The Object Services Server participates in the exportation and importation of object references. Exportation and importation is accomplished by specializing **SOMDServer** and providing unique implementations of the **somdRefFromSOMObj** and **somdSOMObjFromRef** methods. This process is depicted in Figure 11. When an object is exported from the server process, the **SOMOA** invokes **somdRefFromSOMObj** on the Object Services Server object to map the object to an object reference. Likewise, when an object reference is imported into the server process, the **SOMOA** invokes **somdSOMObjFromRef** on the Object Services Server object to map the object reference back to the in-memory object.

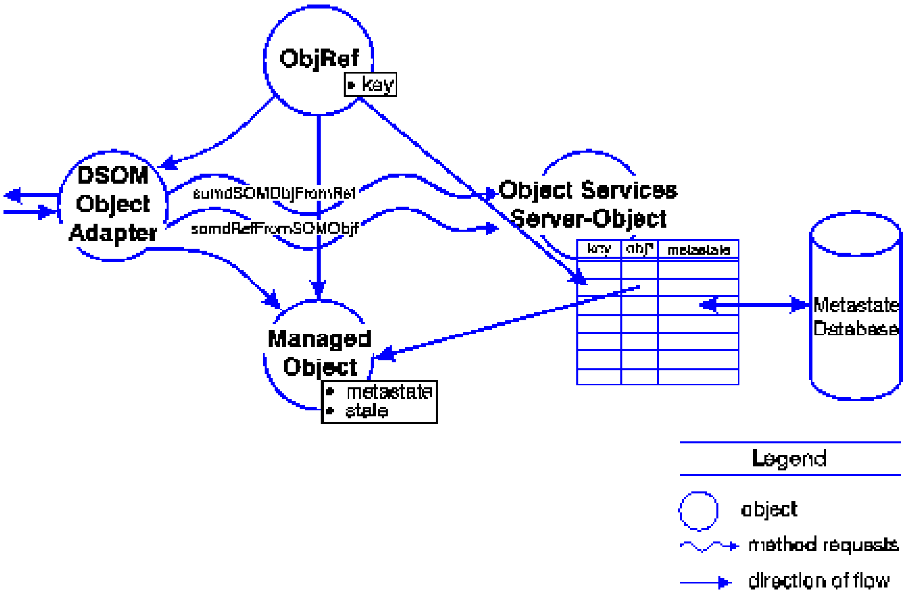

Figure 11. Object Services Server Participates in the Exporting and Importing of Object References

The Object Services Server is capable of maintaining object references persistently, which is useful for objects with persistent state. The Object Services Server manages any metastate that relates to the persistent object as part of the object's reference data keyed by the object's reference. Thus, the Object Services Server is capable of automatically reactivating any passivated objects when they are first referenced.

Role of somOS::ServiceBase

The **somOS::ServiceBase** is the base-class for the Object Services Server. All managed objects **must** be derived from **somOS::ServiceBase** in order to be managed properly by the Object Services Server. The **somOS::ServiceBase** introduces behavior to the managed object that enables the server to manage it.

In most cases, you do not have to be concerned about whether your managed object is derived properly from **somOS::ServiceBase**. All mix-in classes offered by the object services already are derived from **somOS::ServiceBase**. The only thing you must do is explicitly create persistent object references when you create instances of your class. This is discussed in more detail as part of "Automatically Producing Persistent Object References" on page 40.

In addition, the Object Services Server introduces a life cycle programming model that is more specific than the more general model supported by SOMobjects. This model is supported by the **somOS::ServiceBase** with the introduction of the following initializers and destructors:

- **init_for_object_creation**
- **init_for_object_reactivation**
- **init_for_object_copy**
- **uninit_for_object_destruction**
- **uninit_for_object_passivation**
- **uninit_for_object_move**

As you can see, the **somOS::ServiceBase** introduces initializers and destructors whose intentionality is explicit. Explicit intentionality reduces any ambiguity that can occur during the object life cycle. The object Life Cycle Model supported by the Object Services Server and its impact on class programmers is discussed in more detail in “Overview of the Object Life Cycle Model” on page 40.

Finally, the **somOS::ServiceBase** provides an implementation of the Object Identity Service. Object Identity provides several methods that allow object instances to be distinguished from one another. For more information about object identity, see **Chapter 3, Object Identity Service**.

Persistent versus Transient Object References

As defined by CORBA, an object reference is a value that identifies an object. In SOMobjects, object references are themselves objects that represent the identity of an object in an exportable manner; that is, an object reference is distinguished from an object pointer. Such a distinction allows an object reference to be marshalled and communicated across the distributed system without losing the identity to the object (as might occur if the object pointer were exported).

The object reference is an indirection to the object that it identifies. In addition, it is transparently inserted into the reference path for remote clients in the form of a DSOM proxy. As previously stated, the server-object is responsible for mapping between the object pointer and the object reference for an object.

To uniquely identify an object in a way that is address-space neutral, the server-object normally creates a key to the object. The key is embedded in the object reference and is used to map the reference to an object pointer.

With regard to the lifetime of the mapping, the server-object may or may not retain mapping information indefinitely. This property defines the essential difference between transient and persistent object references. Specifically, if the server-object retains the mapping information beyond the lifetime of the server process in which the referenced object exists, the reference is persistent. Conversely, if the mapping is lost when the server exits, the object reference is transient.

To understand this more completely, consider how DSOM manages its transient references. Refer to **Figure 12**, and assume that a client in process **A** has a reference to an object in DSOM Server process **B**. In scene 1, the client object in process **A** has a pointer to a proxy to object ***M***. The proxy is an object reference that can be marshalled over the network to process **B**. The server-object is responsible for mapping the object reference to a pointer to object ***M*** within process **B**.

In scene 2, process **B** is exited. Any transient state in process **B**, including the mapping of object references to pointers is lost. The proxy to ***M*** still contains information for locating DSOM Server process **B**, although process **B** no longer is executing.

In scene 3, DSOM Server process **B** is restarted. The proxy to ***M*** can relocate DSOM Server process **B**; however, because the transient mapping to the pointer to ***M*** has been discarded, the server-object cannot re-establish addressability back to ***M***. Even if another object is reinstantiated where object ***M*** resided, its identity is in the object reference, and the new object is not ***M***. Likewise, even if ***M*** is a persistent object and is reactivated somewhere else in memory, it is difficult to reassociate the reference with the object because the identity of object ***M*** is in the object reference.

More specifically, DSOM carries the address of object ***M*** as part of the identity of the object in its reference. Therefore, if ***M*** is removed from memory, its identity refers to an empty memory location. Worse, if another object is created in ***M***'s old memory address, its identity refers to the wrong object, even if the process itself is not exited.

It is important to retain information about object reference mapping persistently and in a manner independent of the memory address of the object. When a server object retains the mapping between an object reference and an object persistently, the references it produces are called *persistent object references*.

The Object Services Server supports both transient and persistent object references. You can instruct the server to produce a persistent object reference for a given object by invoking the **make_persistent_ref** method on the server-object. You can invoke this method from either within the object itself, or outside the object by some other object; for example, from an object factory. Invoking this method instructs the Object Services Server to build an entry for the object in its metastate database.

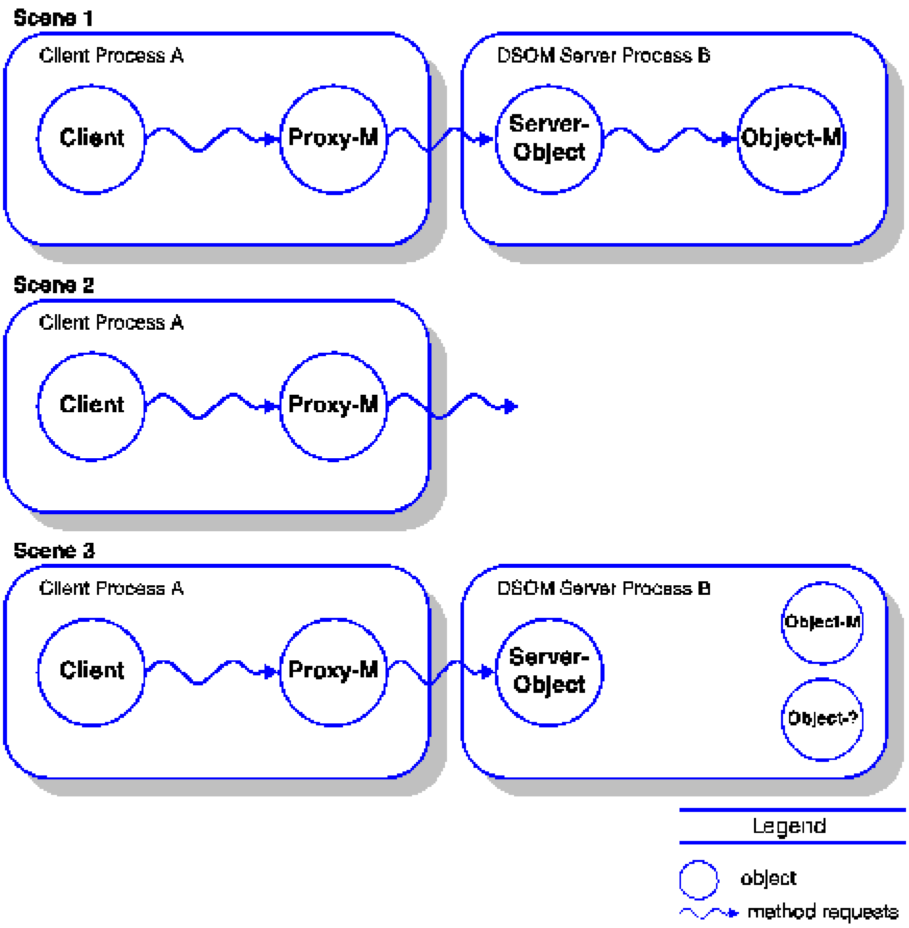

Figure 12. Consequences of Transient Object References

This function provides the Object Services Server with some important capabilities. Objects created within the server can be removed from memory (passivated) and automatically reactivated the next time any method request is invoked on the object. You can use the metastate to reconstruct an association to the object when it is recreated in memory (reactivation). And the metastate can be used by a Persistence Service to restore the persistent state of the object. See "Overview of the Object Life Cycle Model" on page 40 for a more complete definition of object passivation and reactivation.

Automatically Producing Persistent Object References

To help simplify the creation of persistent objects, a specialization of **somOS::ServiceBase** is supplied. It automatically registers the object with the server-object and requests that a persistent reference be created for it when the object is created. This specialization is referred to as **somOS::ServiceBasePRef**. If you are a class programmer and want persistent object references automatically created for instances of your class, you should mix-in **somOS::ServiceBasePRef**.

This class also automatically destroys the persistent object reference for the object when the object is destroyed. See "Overview of the Object Life Cycle Model" on page 40 for a more complete definition of object destruction.

Maintaining Strict CORBA Compliance

CORBA states that an object is "an entity that has state." This definition has interesting implications for the case in which an object has a persistent object reference and transient state. In the scenario described in Figure 12, if the object reference to ***M*** were persistent, in scene 3 the server-object is able to reassociate the reference to object ***M*** in memory (even if the memory address for ***M*** had changed).

However, if ***M***'s state were transient, it would be lost as a result of process **B** exiting in scene 2. The client in process **A** might not even know that process **B** had exited and returned, except that the state contained in object ***M*** in scene 1 would be absent (or reset) in scene 3. The absence of state could change the behavior of object ***M*** very dramatically. In addition, it may appear that ***M*** is no longer the original object. It is a different object because it has a different state and, therefore, has violated strict compliance with CORBA.

There are scenarios in which maintaining a persistent object reference to an object with transient state is perfectly acceptable, even if the state is *lost* from time-to-time (as in the case just described). For example, the state of a print-queue object is transient. Print jobs are printed and discarded from the queue. The reference to the print-object is persistent because it is recognized when the printer object reactivates after a shutdown phase.

However, if your class does not maintain the state for its instances persistently and it is important for you to maintain strict compliance with CORBA, you can mix-in the **somOS::ServiceBaseCORBA** class. If you mix-in this class, an **INV_OBJREF** standard exception is raised if a method is invoked on an instance of your class that has been passivated.

Overview of the Object Life Cycle Model

If you are a class programmer or a client programmer, you should understand the Object Life Cycle Model supported by the Object Services Server.

We have discussed three distinct elements of an object:

- Its reference (and any metastate used to map the reference to the object)
- The in-memory object instance
- The object's persistent state

On the one hand, these elements are highly related because they comprise the object in the largest sense. On the other hand, they are distinct because each has independent life cycles.

An in-memory object can exist with or without a corresponding reference. A persistent reference can be created or destroyed (perhaps multiple times) within the lifetime of an object. The persistent state of an object can continue to exist, even if the in-memory object instance is passivated.

There is a distinction between when an object is first created in its broadest sense and when it is merely being recreated in-memory. The former case is *object creation*; the object in its broadest sense is being created. The latter case is *object reactivation*.

Likewise, there is a distinction between when an object is finally being destroyed in its broadest sense and when it is merely being removed from memory. The former case is *object destruction*; the object in its broadest sense is being destroyed. The latter case is *object passivation*.

An object without an object reference is an object without an exportable identity. The object may exist but indirect clients cannot refer to it. Likewise, a reference to an object that does not exist is an identity without an object. Therefore, in general, the object reference should be created at object creation and destroyed at object destruction.

This reduces the Life Cycle Model down to two independent cycles:

- The life cycle of the object in its broadest sense — object creation and object destruction
- The life cycle of the in-memory instance — object reactivation and object passivation

Managing the Object Life Cycle

Because an object, in the model supported by the Object Services Server, is subject to two distinct life cycles, it must have a way to participate in the life cycles to ensure that the right things happen at the right time. To assist in this, two distinct initializers and two distinct uninitializers have been introduced:

- **init_for_object_creation**
- **init_for_object_reactivation**
- **uninit_for_object_destruction**
- **uninit_for_object_passivation**

These initializers and destructors provide you with the capability to perform the appropriate kind of initialization and uninitialization, depending on whether the object is being created or destroyed, or reactivated or passivated. The following are examples of tasks you can perform with them:

- During **init_for_object_creation**, you can register the object in any frameworks or containers to which the object should belong. Likewise, you can allocate any memory the object needs from the heap or create and initialize persistent storage for your object.
- During **uninit_for_object_passivation**, you can store state of the object to persistent storage or release any memory used in the heap.
- During **init_for_object_reactivation**, you can allocate any memory the object needs from the heap or restore the object's state from persistent storage.
- During **uninit_for_object_destruction**, you can de-register your object from any frameworks or containers of which it has been a part. Likewise, you can destroy any persistent storage your object uses or release any memory used in the heap.

All the preceding examples relating to persistence assume you are managing your own persistence. The introduction of specialized initializers deprecated the use of **somDefaultInit**. Likewise, the use of **somDestruct** is limited to removing the in-memory object (not uninitializing it). Consequently, if you are a client programmer, the following is the prescribed model for creating a new object:

1. Either locate the class object of the object or create it. (You can also perform this step by using the **find_any** method from the Naming Service.)
2. Invoke **somNewNoInit** on the class object to create a new instance. (You can also perform this step by using the **somdCreate(..., FALSE)** convenience function.)
3. Invoke **init_for_object_creation** on the new instance to initialize it. (You can also perform this step by using the **somdCreate(..., FALSE)** convenience function.)

To destroy an object:

1. Invoke **uninit_for_object_destruction** on the instance you want to destroy to uninitialize it.
2. Invoke **somDestruct** on the instance to remove it from memory.

Initializing and uninitializing the object is a distinct step from creating it and destroying it in memory. To simplify the procedure, you can write your own factory object to perform both operations when creating an object.

CAUTION:

Both **somNewNoInit** and **init_for_object_creation** return an object pointer to the newly created object. Always discard the object pointer returned from **somNewNoInit** or **somCreate** after it is used to invoke the **init_for_object_creation** on the new object and a new pointer is returned. If the object is given a persistent object reference, the persistent reference is relevant only to the object pointer returned from **init_for_object_creation**. The pointer returned from **somNewNoInit** is, at best, for a transient object reference.

If you are a client programmer and want to deliberately passivate an object, you can do so by invoking **passivate_object** on the server-object (**somOS::Server**). This, among other things, invokes **uninit_for_object_passivation** followed by **somDestruct** on the object instance.

Objects are automatically reactivated on the first method request that occurs on a passivated object. The server-object, among other things, invokes **init_for_object_reactivation** on the object instance.

As stated in “Role of somOS::ServiceBase” on page 37, there are three initializers and three destructors. Two of each have been discussed. The following are the remaining ones:

- **init_for_object_copy**
- **uninit_for_object_move**

This initializer and uninitializer can be used when an object is copied or moved to another server process. For instance, with pass-by-value, when a replica of your object is created in the target process, **init_for_object_copy** can be invoked to provide you the opportunity to perform the proper initialization for your new copy. In the SOMObjects 3.0 product, these methods are not used.

Service Initialization and Diamond Inheritance

Although they are not declared, as such, in IDL, the initializers and destructors introduced by **somOS::ServiceBase** must be treated as true initializer methods. By convention, if you override any of these initializers to perform your own initialization or uninitialization function, it is important that you give your parent methods an opportunity to perform their own initialization or uninitialization. Therefore, in the initializers, you should invoke each of your parent implementations of the same initializer (if it exists) *before* you perform your own initialization. (If you multiply inherit and any of your parents is not derived from **somOS::ServiceBase**, the initializer method does not exist in that parent for you to invoke. However, you may need to call **SomDefaultInit** on such parents.)

Likewise, in the destructors, invoke each of your parent implementations *after* you perform your own uninitialization.

With multiple inheritance comes the potential for *diamonds* in the class hierarchy, as illustrated in Figure 13.

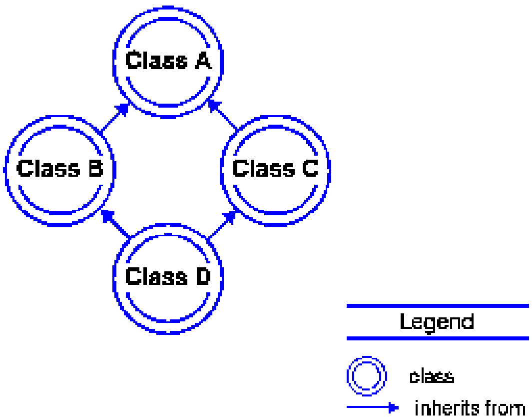

Figure 13. Multiple Inheritance and Diamonds

This inheritance diamond can occur above, if you are multiply inheriting, or below, if you are being multiply inherited. If a class is at the top of an inheritance diamond (**Class-A**) and parent method calls are being made, the method in that class probably is invoked twice — once coming from **Class-B** and again coming from **Class-C**.

If this happens and you are performing initialization, do not perform certain kinds of initialization twice. For instance, if you allocate the same variable twice, a memory leak can occur. Because you cannot predict who might sub-class from you or how, protect against the occurrence of multiple invocations. Do one of the following:

- Detect that you are getting called again and ignore the request. It is probably legitimate to avoid calling your parents in this case.
- Detect whether memory has already been allocated for each variable, and avoid allocating it again.

CAUTION:

Ensure that everyone has the opportunity to perform initialization. If you multiply inherit classes that are derived from **somOS::ServiceBase**, you must override all initializers and uninitializers and invoke each of your parent implementations, even if you do not have another reason to override them. This principle also applies to the reinit and capture methods of **somOS::ServiceBase**, but never use them directly. Because these methods are used by the object services, treat them like initializers. Finally, this principle also applies to **internalize_from_stream** and **externalize_to_stream** if you multiply inherit classes derived from **CosStreamable**.

Configuration of Object Services Servers

Each server maintains two binary databases, one for metastate and one for persistent attributes. The databases are located in the database directory specified by the **SOMDDIR** variable in the configuration file under the **somd** stanza heading.

A master database contains the association of the implementation ID for a server and its unique binary databases. This database is called **SOMOSDB.DAT**. This file is also located in the directory specified by **SOMDDIR**. The master database is created by the first server when it is initialized, which is probably at the time **som_cfg** is executed to configure the runtime environment. (For more information, see “Initializing the Server Manually” on page 45.) For each server, there are two entries in the master database, one for each of the binary databases.

The **adb** file extension specifies an attribute persistence database, and the **mdb** file extension specifies a metastate database.

Each of the binary databases has unique names. The databases cannot be shared between servers. If a server does not find its own databases, it terminates with a log message. In this case, you can restart the server in initialization mode by specifying a *-i* parameter. When specified, the server re-initializes or creates the two required databases and updates the master database. See “Initializing the Server from a Program” on page 45 and “Initializing the Server Manually” on page 45.

Parameters to Configure in the Server Configuration File

To maintain maximum flexibility, each Object Services Server can use a configuration file. This file contains customizable parameters for the server, and you can modify these parameters to improve the performance of the overall system. A sample server configuration file named **SOMOS.INI** is located in the **som\etc** directory.

Depending on your system configuration, you can use either one or more configuration files within a system. For most systems, one configuration file for all the servers is sufficient.

The following two parameters are currently available for configuration:

- **CACHE_NUM_BUCKETS** specifies the number of buckets in the cache. These buckets are searched first before going to the file system. It has a default value of 50.
- **CACHE_BUCKET_ELEMENT_MAX** specifies the maximum number of elements each bucket can contain within the cache. It has a default value of 0 (no limit).

Two stanzas are currently defined for the server within the configuration file:

- The [somos] stanza for metadata
- The [somap] stanza for attribute persistence

For a server to use the parameters specified in a configuration file, the **regimpl** database for that server must specify the server configuration file. If the server configuration file is not specified in the **regimpl** database, the server uses default values for each parameter. Likewise, if a configuration file is specified, but does not contain values for all configurable parameters, default values are used for any parameter not specified.

The following is a sample configuration file:

```
[somos]
CACHE_NUM_BUCKETS                    = 100
CACHE_BUCKET_ELEMENT_MAX             = 0
[somap]
```

```
CACHE_NUM_BUCKETS                     = 200
CACHE_BUCKET_ELEMENT_MAX              = 0
```

To specify a configuration file, edit the **regimpl** database specification for the server, and add the name of the configuration file for the server. The **som_cfg** program registers the Naming Service and the Security Service to use the Object Services Server. For each of these servers, no configuration file is specified, and the system uses default values.

Initializing the Server

You can initialize the server either manually or from a program.

Initializing the Server Manually

The *-i* command line parameter is provided for manual initialization of the Object Services Server. When specified, this parameter initializes the server and sets up its persistent databases for use. The server must be initialized once before use; otherwise, no database exists, and the server terminates with an error log message.

To initialize the server, type the following:

```
somossvr -i -a myServer
```

where *myServer* is the implementation alias.

The server must have been registered with **regimpl** first before it can be initialized. If an Object Services Server has already been initialized and is started again with the *-i* parameter, the server re-initializes its databases. That means all persistence information is lost, and the server starts up with no persistence information. This can be used if a server is out of synch with the rest of the system. Because valuable data can be lost forever, use the *-i* parameter with care once the system is configured and is operational.

Initializing the Server from a Program

To provide a programmatic way for customers to initialize or reinitialize the Object Services Server, the following API is provided:

```
somos_init_pesist_dbs();
```

The following is an example of a C program that calls this API. The server must have been configured in **regimpl** before this call, which can also be performed from within a C program:

```
#include <somosutl.h>

  Environment *global_ev;              /* Global ev */
  char        *impl_alias = NULL;      /* Implementation alias */
  int ret;

  /* ... */

  ret = somos_init_persist_dbs( impl_alias, global_ev );
  if (ret != 0) {
    ret = getExceptionValue( global_ev );
  }
```

The **somos_init_persist_dbs()** function initializes the databases that the Object Services Server requires and prepares the server for use. The server program will fail with an error log message if database initialization has not been performed.

Creating Your Own Server Program

In order to provide customers the most flexibility in creating their own server programs and customize the Object Services Server, the general flow of the server program is explained in the following.

Only users who require further flexibility and require their own version of the server program need to create their own version of the server program. For most users, the standard supplied version of the **somossvr.exe** program is sufficient.

The main server program for the Object Services Server is fairly simple and consists of a few SOM, DSOM, and some special functions required in order for it to be an Object Services Server program. The following example shows the general flow of the Object Services Server program:

```
/*
 *   somossvr.c
 *   This file provides the implementation of the SOMOS Server
 *   program. All operations except for those starting with
 *   "somos_" are required by all DSOM Server programs.
 */

/***********************************************************/
/*   Includes                                              */
/***********************************************************/
#include <stdio.h>
#include <somd.h>
#include <implrep.h>
#include <somosutl.h>

/***********************************************************/
/*   Macros                                                */
/***********************************************************/
#define CHECK_EV(ev)   ((ev)->_major != NO_EXCEPTION)

/***********************************************************/
/* External Variables                                      */
/***********************************************************/
extern   char    *optarg;
extern   int     optind;

/***********************************************************/
/* Prototypes                                              */
/***********************************************************/
void usage( void );

/***********************************************************/
/* Usage Function                                          */
/***********************************************************/
void
usage( void )
{
  fprintf( stderr,

            "somossvr [-i] [-d] -a [impl_alias | impl_uuid]\n" );
  exit( SOMOS_USAGE_ERROR );
}

/***********************************************************/
/* Main Program                                                */
/***********************************************************/
int
```

```
main
(
  int         argc,
  char        **argv
)
{
  Environment *global_ev;              /* Global ev            */
  char        *impl_alias = NULL;      /* Implementation alias */
  char        *impl_id = NULL;         /* Implementation id    */
  int         c;                        /* Parameter            */
  boolean     debug_mode = FALSE;      /* Debug                */
  boolean     initialize_mode = FALSE; /* Initialization mode  */
  int rc;                               /* Return code          */

  somos_init_logging();

  /*   Get options, See usage for valid options */
     while ((c = somos_getopt(argc, argv, "a:id")) != -1)
  {
    switch (c)
    {
    case 'a':
        impl_alias = optarg;
        break;
    case 'd':
        debug_mode = TRUE;
        break;
    case 'i':
        initialize_mode = TRUE;
        break;
    default:
        usage ();
        break;
    }
  }
  if ( optind != argc ) {
    if ( impl_alias != NULL ) {

      /* Cannot pass both implementation alias and id */
      usage();
    }
    impl_id = argv[optind];
  }
  if ( !impl_alias && !impl_id ) {

    /* Cannot continue, must supply either the
     * implementation alias or id */
    usage();
  }

  /* Setup SOMOS internals */
  somos_setup();

  /* Get the global environment */
  global_ev = somGetGlobalEnvironment();

  /* Initialize the DSOM run-time environment */
  SOMD_Init( global_ev );

  SOMD_NoORBfree();

  /* Create a SOMOA object and initialize the global
   * variable SOMD_SOMOAObject */
  SOMD_SOMOAObject = SOMOANew();
```

```
  if ( debug_mode ) {

    /* Turn method tracing on */
    SOM_TraceLevel = 1;
  }

  /* Find implementation by alias or id */
  if ( impl_alias ) {
    SOMD_ImplDefObject = _find_impldef_by_alias(
                                   SOMD_ImplRepObject,
                                   global_ev, impl_alias );
  }
  else {
    SOMD_ImplDefObject = _find_impldef( SOMD_ImplRepObject,
                                        global_ev,
                                        impl_id );
  }
  if ( CHECK_EV( global_ev ) ) {

    /* Could not find implementation definition.
     * Cannot continue! */
    somos_exit( SOMOS_FIND_IMPLDEF_FAILED );
  }

  /* Initialize any required object services */
  somos_init_services( initialize_mode );

  /* Register the implementation with the SOMOA */
  _impl_is_ready( SOMD_SOMOAObject,
                  global_ev,
                  SOMD_ImplDefObject );
  if ( CHECK_EV(global_ev) ) {
      somos_exit( SOMOS_IMPL_IS_READY_FAILED );
  }

  /* Initialize any required object services -
   * after impl_is_ready */
  somos_init_services_afterimpl(initialize_mode);
  printf( "%s%s%s\n","somOS::Server (",
          __get_impl_alias(SOMD_ImplDefObject,global_ev),
          " ) - Ready" );

  /* Enter the somoa main loop - will not return */
  rc = _execute_request_loop( SOMD_SOMOAObject,
                              global_ev,
                              SOMD_WAIT );
  if(rc || CHECK_EV(global_ev))
    somos_exit(SOMOS_REQUEST_LOOP_ERROR);
  else
    somos_exit(0);
}
```

The first section in **main** has to do with error logging and parameter passing. The **somos_init_logging** function initializes the error logging facility. The **somos_getopt** function is provided to aid in the passing of parameters to the server program. In the sample server program -a, -d, and -i are filtered out. These parameters have the following functions within the server program:

```
-a  - to pass the implementation alias
-i  - to initialize the server databases the first time the server
        is started
-d  - to run the server in debug mode
```

After the parameter passing, the server program does error checking and initial setup. This is done with the **somos_setup** call.

After this, the global environment is initialized and the DSOM run-time environment created with the standard DSOM calls **somGetGlobalEnvironment** and **SOMD_Init**. Then the SOMOA object is created, and the implementation is found either by an alias or an ID.

Internal services in the **somOS::Server** are initialized with the **somos_init_services** call, which is followed by **impl_is_ready**. This, in turn, is followed by more initialization of internal services in the server after **impl_is_ready**. If everything is successful, the server enters into the request loop with the **execute_request_loop** call. The server does not return from this call and remains within the request loop to process all incoming method requests to the Object Services Server.

The **somos_exit** function is called if the server program fails, or if the server is terminated. This function preforms special uninitializations on behalf of the Object Services Server and various object services. The object services can perform special uninitializations by registering their individual exit callback procedures using the **somos_register_exit_callback** function defined in the **somosutl.h** file. Prior to calling any exit callback procedure for an object service, the server passivates all objects.

The preceding program can be compiled with any C or C++ compiler that SOMobjects supports. In order to resolve the **somos_** API calls, the program must be linked with the **somtk** library provided in the SOMobjects Developer Toolkit. Function prototypes and return codes are specified in the **somosutl.h** file.

It is important that these API calls are performed in the order just given for your own version of the Object Services Server to function properly.

Registering the Server with regimpl

In order to use the Object Services Server, you must configure an implementation with **regimpl**. You must use the Object Services Server as your implementation server for any of the Object Services. in your system. The **som_cfg** program automatically configures the Naming Service and the Security Service to use the Object Services Server. When configuring a new implementation that uses the Object Services Server, the server must be initialized first before any method requests can be processed. For more information about initialization, refer to “Initializing the Server” on page 45.

The following example shows the registration parameters required to register a new server in the **regimpl** database:

```
Implementation alias:      myServer
   ImplDef Class name:     ImplementationDef
   Program name:           somossvr.exe
   Multithreaded:          Y
   Server secure:          N
   Server class:           somOS::Server
   Configuration file:
   IPC protocol:           Y
```

Depending on the system and the particular services for which the Object Services Server is needed, you can specify it as either a single-threaded server or a multi-threaded server. A configuration file is optional; specify it only when optimizing the environment. For most systems, it is recommended that a configuration file not be specified. For the Object Services Server to function properly, you must configure it with the **somOS::Server** class.

After the server implementation has been configured, the user can add classes to the implementation in order to support their environment.

Note: When a new implementation of the Object Services Server is specified in **regimpl**, the specific server has not been initialized. The databases required by the server for metadata and attribute persistent have not been initialized. To perform this initialization, start the server with an *-i* parameter first, before it is available for any method calls. An API call is also provided to perform this initialization, as explained in the previous section.

Chapter 6. Security Service

This chapter discusses the Security Service. The Security Service supports authentication functionality only.

Included in this chapter is an introduction to the concepts of Security Service, principals, and authentication. Also discussed are the security server, security perspectives of the end user and administrator, and configuring a server as secure.

Concepts

Security Service ensures that only authenticated users can invoke privileged operations. Implementing the service requires that all users be authenticated. Users must be known to the system by a well-known name, such as a login name or user ID (UID), and must be able to be validated by some means, such as passwords. Any DSOM server application that exports objects to other applications and requires that object integrity not be compromised must use the Security Service. Figure 14 shows how the Security Service fits into the SOM/DSOM environment. This figure depicts the case that the security server is configured on an OS/2 platform using the OS/2 LAN Server registry.

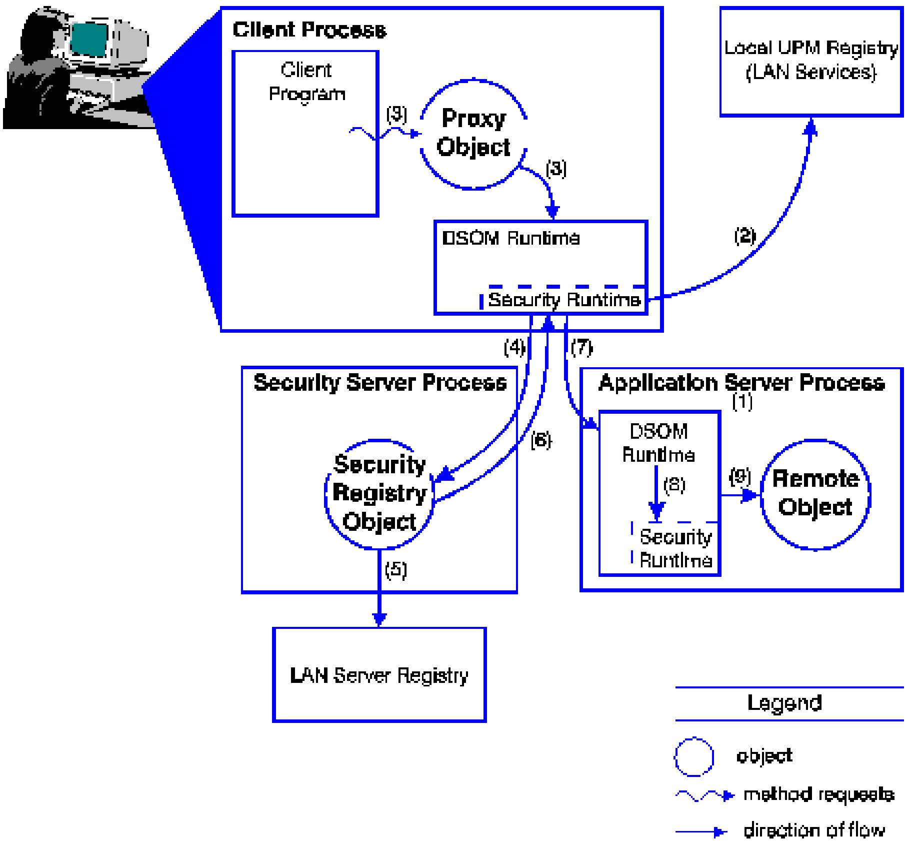

Figure 14. Security Service in a SOM/DSOM Environment

Principal

A *principal* represents the identity of a user (or entity) executing the program. The entity must be registered in a user registry and must be authenticated. For example, users wanting to use a system must log in by identifying themselves with names and passwords. Although there are many ways to authenticate principals, this Security Service authenticates by password only.

Establishing an Authenticated Session

Authentication is the process of validating a user with a password. In this product, users are authenticated with a platform-specific user registry. The platform on which the security server is configured determines which user registry is used. In the case of OS/2, the LAN Server 4.0 registry is used. In the case of AIX, the native AIX registry is used. In the case of Windows/NT, the NT registry is used.

If users do not have an account in the user registry, they still can execute the DSOM applications. In this case, the DSOM run time treats users as unauthenticated users. The success of program execution depends upon whether the server is configured to run in secure mode (see the section on Security Perspective: “Administrator” on page 55).

An authenticated session is established between a user and the application server, which is configured to run in secure mode. The following steps, which correspond to the numbers in **Figure 14** on page 51, show how an authenticated session is established (this flow is specific to the case where the security server is configured on an OS/2 platform, but the flows are similar for other platforms):

1. Configure your server in secure mode. (Refer to “Configuring a Server as Secure” on page 55.)
2. The security runtime obtains the login context from the environment or local UPM registry. (UPM is the User Profile Manager that accompanies LAN Server 4.0 on OS/2 clients. The local UPM registry is a cache of the user’s credentials on the OS/2 client.)
3. When any method is invoked on a remote object (through its proxy), the client principal is authenticated.
4. The client principal is authenticated with the security registry.
5. The credentials for the client principal are verified with the native verification service for the operating system.
6. The security registry returns an indication of whether the principal is authenticated. A security token is created and retrieved for authenticated principals.
7. The method request then is forwarded to your application server.
8. The security token is validated by the server to ensure that the request is authentic.
9. If the request is authentic, it is forwarded to the intended object.

All further messages from the client are tagged into a special token to indicate the authenticity of the message.

Security Server and Security Domain

The *security* server is a DSOM server that supports the authentication functionality. The actual authentication of users is based upon the user-registry mechanism specific to the platform where the security server is configured. For instance, if the security server is configured on an OS/2 machine, then the LAN Server registry will be used. This implies that all users executing DSOM applications and wanting to run authenticated must have an account in the LAN Server registry.

The security server does not maintain any passwords or user account information. For any account administration, the LAN Server (or other platform specific) tools must be used.

A security domain is one in which all DSOM applications communicate with only one security server. The security server authenticates clients by verifying the login name and password from the user registry. In the case of OS/2, the security server must run on the same machine as that of the LAN Server or on a box on which the LAN Server registry is replicated. In the latter case, it is essential that the machine also be physically secure. The functionality required to facilitate interdomain authentication is not supported.

Security Perspective

This section explains security from the perspective of both the end user and the administrator.

End User

The end user is usually a client program or a server that is behaving as a client to another server. In order for any DSOM application to run authenticated, the user must log in successfully. The user can log in using one of two different means, depending on the client platform and the options specified in the client's configuration file. The following attribute can be set in the security stanza of the client's configuration file:

```
LOGIN_INFO_SOURCE=<source-option>
     where <source-option> is either:
          DEFAULT
     or, one or more of:
          ENV
          PROMPT
          UPM
```

The source options, ENV, PROMPT and UPM can be specified in any order and any combination. However, PROMPT is only valid when the client is on an AIX platform, and UPM is only valid when the client is on an OS/2 platform. Where more than one source is specified, the security service attempts to obtain the information from each source in turn until a source yields all the required information (for example, login name and password), or until all the sources are exhausted without success. Unrecognized sources are ignored and if insufficient information is acquired then the user is not authenticated.

If DEFAULT is specified then the security service will default to using UPM on OS/2 and ENV on NT and AIX. In the configuration file, `somenv.ini`, that accompanies SOMobjects 3.0, this is the default.

If ENV is specified then the security service will acquire the user name and password from environment variables. The user name is obtained from USER on OS/2 and AIX, and from USERNAME on NT. USER and USERNAME are normally set up by the respective operating systems. The password is obtained from PASSWD. It is up to the client application to set up the PASSWD environment.

The environment variables may be set programmatically in the client program (before the program initializes the DSOM runtime). Here is a “C” example:

```
int main()
{
     /* Variable declarations, etc */
     /* ... */

     putenv("PASSWD=mypasswd");

     /* ... */

     SOMD_Init();

     /* Use DSOM */
     /* ... */

}
```

This approach requires the client program source to be edited and recompiled if the program is to run with different login information.

A more flexible alternative is to set up the variables outside the client program.

On AIX:

```
export PASSWD=mypasswd
```

On OS/2:

```
set PASSWD=mypasswd
```

On NT:

```
set PASSWD=mypasswd
```

or use the NT registry.

If PROMPT is specified, the security service prompts the client for the login information. This interactive source option is available on AIX only. See "Authentication on AIX" on page 55 for further details.

If UPM is specified, the security service obtains the login information from UPM. This source option is available on OS/2 only.

If you do not specify this attribute in the client's configuration file, that is equivalent to:

```
LOGIN_INFO_SOURCE=DEFAULT
```

Authentication on OS/2: When the security server is configured on an OS/2 platform, it uses the LAN Server 4.0 user registry for authenticating users. Users must be registered with the LAN Server user registry to access a secure server with DSOM.

If the client is using OS/2, users can log in using UPM. If the LOGIN_INFO_SOURCE attribute in the configuration file is set to use UPM as a source, the security service obtains login information from the UPM local registry.

If the user logs in to multiple LAN server domains, only the first entry in the UPM local registry is picked up. The end user also must ensure that the same login name and password are valid on the machine or node on which the security server is running. The application the user is invoking runs only under one identity, and the application cannot make authenticated associations with applications executing in a different domain.

Users who are logged on to the LAN server, but do not want to run authenticated, can do so by setting the following attribute in the security stanza of the configuration file:

```
LOGIN_INFO_SOURCE=
```

If the attribute is set to nothing (as shown), the security runtime ignores the user's log in into the LAN server.

Authentication on AIX: When the security server is configured on an AIX platform, the security service validates security information against the native AIX user registry. Users must be registered with the user registry on that AIX host to access a secure server with DSOM. Also, the security server must be run with superuser (that is, "root") privilege so that it can access the user registry. One way to achieve this is to have the security server started by an administrator logged-in to AIX as **root** or with **su**.

If LOGIN_INFO_SOURCE is set to PROMPT at the client on an AIX platform, the security service prompts the client to run the **somlog** program. The **somlog** program will in turn prompt for the client's login name followed by the password. **somlog** does not display the password.

Authentication on NT: When the security server is configured on an NT platform, the security service validates security information against the NT user registry. Users must be registered with the user registry on that NT host to access a secure server with DSOM. Also, the security server requires NT **SecTcbPrivilege** in order to access the user registry. One way to achieve this is to have the security server started by an administrator with "Act as part of the operating system" User Rights Policy. This can be conferred using the Policies pull-down menu of the User Manager (administration tool) for the administrator.

Administrator

The administrator has two responsibilities:

- Security server
- Security mode of a server application

Security Server: The administrator must ensure that the security server always is running. This is crucial because the clients get authenticated by the security server.

The security server is a DSOM server and is started automatically by DSOM if it isn't already up; for example, during SOMobjects 3.0 configuration. However, the security server requires **superuser** or **SecTcbPrivilege** to perform login validation on AIX and NT, respectively, and might have to be started manually by an administrator with the required privilege. The command to start the server is:

```
somossvr -a securityServer
```

The *securityServer* is the DSOM alias assigned to the security server during the configuration of SOMobjects 3.0. If you change the alias to something else, you need to modify the previous statement accordingly.

Secure Mode of a Server Application: A server running in secure mode implies that it accepts method requests from authenticated clients only. An administrator can configure a server to run in secure mode using the **regimpl** command (refer to the registration steps using **regimpl** in "Configuring a Server as Secure" on page 55). If the secure mode flag is set to YES, then only authenticated clients can invoke methods. If, for any reason, the flag is changed, the application server must be restarted for the flag to take effect.

Configuring a Server as Secure

For a server process to be run securely, it must be registered as a secure server in the SOM implementation repository. Do this by invoking the **regimpl** command:

```
regimpl -u -i <...> -s [ on | off ]
```

-s

Set to on to make the server secure. This ensures that only authenticated clients can invoke calls on the objects the server supports.

See “The regimpl Registration Utility” on page 32 of *Programmer’s Guide for SOM and DSOM* for additional information on **regimpl**.

Note: Ensure that the global naming server is secure.

Glossary

A

abstract class. A class that serves as a base class for the definition of subclasses. Regardless of whether an abstract class inherits instance data and methods from parent classes, it always introduces methods that must be overridden in a subclass.

affinity group. An array of class objects that were all registered with the SOMClassMgr object during the dynamic loading of a class. Any class is a member of at most one affinity group.

aggregate type. A user-defined data type that combines basic types (such as, char, short, float, and so on) into a more complex type (such as structs, arrays, strings, sequences, unions, or enums).

apply stub. A procedure corresponding to a method that extracts the arguments from the va_list, invokes the method, and stores its result. Also are registered with class objects when instance methods are defined. Invoked using the somApply function.

B

base class. See parent class.

Basic Object Adapter (BOA). A type of object adapter defined by CORBA to support a wide variety of common object implementations.

behavior (of an object). The methods that an object responds to. These methods are those either introduced or inherited by the class of the object. See also state.

BOA (basic object adapter) class. A CORBA interface, which defines generic object-adapter (OA) methods that a server can use to register itself and its objects with an ORB (object request broker).

BOA. See Basic Object Adapter.

C

casted dispatching. A form of method dispatching that uses casted method resolution.

class object. The run-time object representing a SOM class. In SOM, a class object can perform the same behavior common to all objects, inherited from SOMObject.

class variable. Instance data found within an object that is a class.

client code. An application program, written in the programmer's preferred language, which invokes methods on objects that are instances of SOM classes. In DSOM, this could be a program that invokes a method on a remote object.

compound name. In the Naming Service, a name that has multiple components. Name components are IDL structures.

constraint. In the Naming Service, an expression used to describe the characteristics of a bound object being searched for. Constraints are expressed in Constraint Language.

CORBA. The Common Object Request Broker Architecture established by the Object Management Group. The SOM Interface Definition Language used to describe the interface for SOM classes is fully compliant with CORBA standards.

D

data token. A value that identifies a specific instance variable within an object whose class inherits the instance variable derived class See subclass and subclassing.

descriptor. An ID representing the identifier of a method definition or an attribute definition in the Interface Repository. The IR definition contains information about the method's return type and the type of its arguments.

dispatch method. A method invoked in order to determine the appropriate method procedure to execute.Using dispatch methods facilitates dispatch-function resolution in SOM applications and enables method invocation on remote objects in DSOM applications.

DLL. dynamic link library.

dynamic dispatching. Method dispatching using dispatch-function resolution

Dynamic Invocation Interface (DII). The CORBA-specified interface, that is used to dynamically build requests on remote objects. DSOM applications can also use the somDispatch method for dynamic method calls when the object is remote.

dynamic link library. A piece of code that can be loaded (activated) dynamically. This code is physically separate from its callers. DLLs can be loaded at load time or at run time. Widely used term on OS/2 and other operating systems.

E

emitter. Generically, a program that takes the output from one system and converts the information into a different form. Using the Emitter Framework, selected output from the SOM Compiler is transformed and formatted according to a user-defined template.

encapsulation. An object-oriented programming feature whereby the implementation details of a class are hidden from client programs, which are required to know the only interface of a class in order to use the class's methods and attributes.

entry class. In the Emitter Framework, a class that represents some syntactic unit of an interface definition in the IDL source file.

Environment parameter. A CORBA-required parameter in all method procedures, it represents a memory location where exception information can be returned by the object of a method invocation.

F

factory. An object that is capable of creating another object.

I

ID. See somId.

Implementation Repository. A database used by DSOM to store the implementation definitions of DSOM servers.

implementation. The specification of what instance variables implement an object's state and what procedures implement its methods (or behaviors). In DSOM, a remote object's implementation is also characterized by its server implementation (a program).

index. In the Naming Service, an index that the user can create on specific properties in a naming context. It improves the performance of searches that involve a property.

inheritance hierarchy. The sequential relationship from a root class to a subclass, through which the subclass inherits instance methods, attributes, and instance variables from all of its ancestors, either directly or indirectly.

in-memory object. An object instantiated in memory. Differs from an object whose state can be stored in a persistent database for which no in-memory object has been instantiated.

instance method. A method valid for an object instance (versus a class method, which is valid for a class object). An instance method that an object responds to is defined by its class or inherited from an ancestor class.

instance token. A data token that identifies the first instance variable among those introduced by a given class. The somGetInstanceToken method invoked on a class object returns that class's instance token.

instance. (Or object instance or just object.) A specific object, as distinguished from a class of objects. See also object.

Interface Repository (IR). The database that SOM optionally creates, providing persistent storage of objects representing the major elements of interface definitions.Creation and maintenance of the IR is based on information supplied in the IDL source file.

Interface Repository Framework. A set of classes that provide methods whereby executing programs can access the persistent objects of the Interface Repository to discover everything known about the programming interfaces of SOM classes.

IR. Interface Repository.

L

location services daemon. A process whose primary purpose is to give DSOM clients the communications information they need to connect with an implementation server.

M

macro. An alias for executing a sequence of hidden instructions. In SOM, typically the means of executing a command known within a binding file created by the SOM Compiler.

managed object. An object subject to any of the SOMobjects object services.

metaclass. A class whose instances are classes. In SOM, any class descended from SOMClass is a metaclass. The methods a class inherits from its metaclass are sometimes called class methods (in Smalltalk) or factory methods (in Objective-C) or constructors.

metastate. The state introduced to an object and used by an object service framework.

method descriptor. See descriptor.

method ID. A number representing a zero-terminated string by which SOM uniquely represents a method name. See also somId.

method pointer. A pointer type that identifies one method on a single class. Method pointers are not ensured to be persistent among multiple processes.

method procedure. A function or procedure, written in an arbitrary programming language, that implements a method of a class. A method procedure is defined by the class implementor within the implementation template file generated by the SOM Compiler.

method table. A table of pointers to the method procedures that implement the methods that an object supports. See also method token.

method token. A value that identifies a specific method introduced by a class. A method token is used during method resolution to locate the method procedure that implements the identified method.

module. The organizational structure required within an IDL source file that contains interface declarations for two (or more) classes that are not a class-metaclass pair. Such interfaces must be grouped within a module declaration.

multiple inheritance. The situation in which a class is derived from (and inherits interface and implementation from) multiple parent classes.

N

name binding. In the Naming Service, a name-to-object association. Different names can be bound to an object in the same or different naming contexts at the same time.

name. In the Naming Service, an ordered sequence of name components, which are IDL structures composed of id and kind strings. A simple name has a single component.

names library. In the Naming Service, a library of names from that and other services. It allows names to evolve without affecting existing clients. Names are implemented as pseudo-objects, which are converted to and from structures.

naming context. In the Naming Service, an object that contains name-object associations (bindings).

naming scope. See scope.

Naming Service. A service that provides the ability to refer to objects by name. It organizes computing resources so that they easily can be located, identified, and categorized either in context or by explicit characterization.

nonstatic method. A special kind of SOM method.

O

object adapter. Defined by CORBA as being responsible for object reference, activation, and state-related services to an object implementation.

object definition. See class.

object implementation. See implementation.
object instance. See instance and object.
object passivation. The process of deleting the in-memory instantiation of an object, especially an object with persistent state even after being passivated.
object reactivation. The process of re-instantiating an object in-memory, especially when the object exists in persistent form even before being reactivated.
object reference. A CORBA term denoting the information needed to reliably identify a particular object. This concept is implemented in DSOM with a proxy object in a client process, or a SOMDObject in a server process. See also proxy object and SOMDObject.
object request broker (ORB). See ORB.
object services base class. The base class for object services mix-in classes.
object services mix-in class. Any mix-in class introduced by an object service that is intended to be mixed-in to a managed object.
Object Services Server. A server that, with the DSOM object adapter, exports and imports object references. Aa a specialization of the DSOM framework, supports SOMobjects Object Services, handling such tasks as metastate and persistent object references.
object services server-object. The Object Services Server specialization (somOS Server) of the default DSOM framework server-object (SOMDServer).
objref. An abbreviation for object reference, specified by CORBA to be a value that unambiguously references an object.
OIDL. The original language used for declaring SOM classes. The acronym stands for Object Interface Definition Language. OIDL is still supported by SOM, but it does not include the ability to specify multiple inheritance classes.
OOP. object-oriented programming.
operation. See method.
ORB. (object request broker). A CORBA term designating the means by which objects tranparently make requests (that is, invoke methods) and receive responses from objects, whether they are local or remote.
overridden method. A method defined by a parent class and reimplemented (redefined or overridden) in the current class.
override. The technique by which a class replaces (redefines) the implementation of a method that it inherits from one of its parent classes. An overriding method can elect to call the parent class's method procedure as part of its own implementation.

P

parent class. A class from which another class inherits instance methods, attributes, and instance variables. A parent class is sometimes called a base class or superclass.
parent method call. A technique where an overriding method calls the method procedure of its parent class as part of its own implementation.
persistent object. An object whose state can be preserved beyond the termination of the process that created it. Typically, such objects are stored in files.
persistent reference. An object reference that can survive the process or thread that created it.
principal. The user on whose behalf a particular (remote) method call is being performed.
procedure. A small section of code that executes a limited, well-understood task when called from another program. In SOM, a method procedure is often referred to as a procedure. See method procedure.
process. A series of instructions (a program or part of a program) that a computer executes in a multitasking environment.
pragma. A compiler directive, usually specified in code by #pragma.
property. A name-value pair associated with a name binding. The name can be any CORBA String and the value is a CORBA any.

R

readers and writers. A reader is a process that does not intend to update the object, but wants to watch as other processes update it. A writer is a process that wants to update the object as well as continually watch the updates performed by others.

receiver. See target object.

run-time environment. The data structures, objects, and global variables that are created, maintained, and used by the functions, procedures, and methods in the SOM run-time library.

S

scope. That portion of a program within which an identifier name has visibility and denotes a unique variable. An IDL source file forms a scope. An identifier can only be defined once within a scope.

server object. An artifact in the DSOM framework to assist in the mapping of object references to in-memory objects, and in-memory objects to object references. The mapping is used in the exportation and importation of object references.

shadowing. A technique that is required when any of the entry classes are subclassed. Shadowing causes instances of the new subclasses to be used as input for building the object graph, without requiring a recompile of emitter framework code.

signature. The collection of types associated with a method (the type of its return value, if any, as well as the number, order, and type of each of its arguments).

simple name. In the Naming Service, a name that has a single component. Name components are IDL structures.

SOM Compiler. A tool provided by the SOM Toolkit that takes as input the interface definition file for a class (the .idl file) and produces a set of binding files that make it more convenient to implement and use SOM classes.

SOMClass. One of the three primitive class objects of the SOM run-time environment. SOMClass is the root (meta)class from which all subsequent metaclasses are derived. SOMClass defines the essential behavior common to all SOM class objects.

SOM-derived metaclass. See derived metaclass.

SOMDObject. The class that implements the notion of a CORBA object reference in DSOM. An instance of SOMDObject contains information about an object's server implementation and interface, as well as a user-supplied identifier.

SOMDServer. The default implementation of a server-object provided by the DSOM framework.

somId. A pointer to a number that uniquely represents a zero-terminated string. Such pointers are declared as type somId. In SOM, somIds are used to represent method names, class names, and so forth.

SOMOA. The DSOM implementation of a CORBA object adapter.

SOMObject. One of the three primitive class objects of the SOM run-time environment. SOMObject is the root class for all SOM (sub)classes. SOMObject defines the essential behavior common to all SOM objects.

somOSServiceBase. The module and interface name for the managed object base class.

somSelf. Within method procedures in the implementation file for a class, a parameter pointing to the target object that is an instance of the class being implemented. It is local to the method procedure.

state (of an object). The data (attributes, instance variables and their values) associated with an object. See also behavior.

static linkage. Occurs when a program uses data or functions that are defined elsewhere. Simply declaring the existence of external data or functions does not create this linkage, actual usage of external data or functions is required.

static method. Any method you can access through offset method resolution. Any method declared in the IDL specification of a class is a static method. See also method and dynamic method.

stub procedures. Method procedures in the implementation template generated by the SOM Compiler. They are procedures whose bodies are largely vacuous, to be filled in by the implementor.

superclass. See parent class.

symbol. Any of a set of names that are used as placeholders when building a text template to pattern the desired emitter output. When a template is emitted, the symbols are replaced with their corresponding values from the emitter's symbol table.

T

target object. The object responding to a method call.The target object is always the first formal parameter of a method procedure. For SOM's C-language bindings, the target object is the first argument provided to the method invocation macro, _methodName.

transient object. In CORBA, an object whose existence is limited by the lifetime of the process or thread that created it. In SOMobjects, more accurately an object with a transient state.

transient reference. An object reference whose existence is limited by the lifetime of the process or thread that created it.

U

usage bindings. The language-specific binding files for a class that are generated by the SOM Compiler for inclusion in client programs using the class.

W

writers. See readers and writers.

Index

P

R

S

U

Printed in U.S.A.

.