OS/390

SOMobjects Object Services

SC28-1995-01

OS/390

SOMobjects Object Services

SC28-1995-01

Note

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

Second Edition, September 1997

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

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

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

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

FAX (United States & Canada): 1+914+432-9405
FAX (Other Countries):
Your International Access Code +1+914+432-9405

IBMLink (United States customers only): KGNVMC(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL
Internet: mhvrcfs@vnet.ibm.com

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

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

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

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

Contents

Figures

Notices

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

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate MVS 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 MVS 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.

Programming Interface Information

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

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

Trademarks

AIX is a trademark of International Business Machines Corporation.
CompuServe is a trademark of CompuServe, Inc.
FrameViewer is a trademark of Frame Technology.
IBM is a registered trademark of International Business Machines Corporation.
OS/2 is a trademark of International Business Machines Corporation.
SOM is a trademark of International Business Machines Corporation.
SOMobject is a trademark of International Business Machines Corporation.
VisualAge C++ is a trademark of International Business Machines Corporation.

Summary of Changes

Summary of Changes
for SC28-1995-01
OS/390 Version 2 Release 4

This book contains information previously presented in *OS/390 SOMobjects Object Services*, which supports Version 1 Release 3.

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

New Information

There is no new information with this release.

Changed Information

Minor editorial changes have been made and are indicated by a vertical line to the left of the change.

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

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

About This Book

OS/390 SOMobjects Object Services contains information about the Object Services for SOMobjects. Object Services help you manage objects by letting you name them, operate them securely, manage their persistence, and the like. This 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.

Who Should Use This Book

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 CORBA IIOP 1.0
- Familiarity with the OMG Common Object Services, in particular:
  - Externalization Service
  - Life Cycle Service
  - Naming Service
  - Persistent Object 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

How This Book is Organized

This book provides information about SOMobjects Object Services. Topics covered include:

- Externalization Service
- Object Identity Service
- Life Cycle Service
- Naming Service
- Object Services Server
- Persistent Object 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.
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.

Where to Find More Information

The OS/390 SOMobjects publications library includes the following:

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

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

CORBA Publications

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

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 SOM 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. The Persistence Service uses this service to store and restore objects.

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.

Life Cycle Service The Life Cycle Service enables clients to create and remove local or distributed objects without having to be aware of the location of the objects. The Life Cycle Service also provides a mechanism that can be used to customize your development environment and manage the creation of objects on your servers.

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.

Persistence Service The Persistence Service enables you to store an object's state persistently without becoming dependent on any specific data-storage medium. Objects that store their state persistently can be long lived. That is, client programs can be stopped and started and the object continues to maintain its state. The Persistence Service works with the Externalization Service as the basis for keeping the persistent object independent of any particular data-store.

Relationship to Standards

All of the SOMobjects Object Services are implementations of *OMG CORBAservices: Common Object Services Specification*.

Notes:

The following terms might be used interchangeably in this documentation:

1. interface and class
2. instance and object
3. 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. And 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” in *Programmer's Reference for Object Services*. This topic 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 the 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 infor-

mation 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 programming and class programming. 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 will create an instance of your class and invoke 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 specializatio n 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 classNameNew 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” in *OS/390 SOMobjects Programmer's Guide*. 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” in *OS/390 SOMobjects Programmer's Guide*.

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” in *OS/390 SOMobjects Programmer's Guide*.

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” and “Managing the Object Life Cycle”.

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** utility. See “The regimpl Registration Utility” in *OS/390 SOMobjects Configuration and Administration Guide*.

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
- Life Cycle Service
- Life Cycle Relationship with the Object Services Server
- Life Cycle Relationship with the DSOM Factory Service
- Creating Contexts
- Object Services Server
- Overview of the Object Life Cycle Model
- Managing the Object Life Cycle
- Scenarios for Creating and Using Persistent Objects
- Guidelines for Application Developers
- Creating a Recoverable Server
- Remote Factory

Chapter 1. Externalization Service

Introduction

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. Internalization can create a new object, or it can read the data into an existing object, depending on the method you use.

The storage medium used by the stream is referred to generically as the *buffer*. It may be a block of memory, a disk file, or any other storage device. The buffer is the transportable part of the stream. For example, to copy several objects from one process to another, you first externalize the objects to the stream. Then you get the stream's buffer, a block of memory, and copy it to the other process. In the other process, you create a stream and set its buffer to the block of memory. Then, you use the internalize method to create the new copies of the objects.

There are many applications of Externalization Service. DSOM uses Externalization to implement pass-by-value parameters. The Persistence Service uses Externalization to store and restore objects. Externalization could also be used to duplicate objects in a network of computers, for performance or reliability.

An important feature of the Externalization Service is machine architecture independence. The externalize and internalize methods perform automatic conversion for code page, big/little endian, and floating point format. For example, you can externalize objects on a machine that uses ASCII and little endian numeric format, and then internalize objects from that same buffer on a machine that uses EBCDIC and big endian numeric format. Some stream representation implementations accomplish this by storing the format information in the buffer header, others always store the information in a known format. Refer to the description of each stream for details.

Note: The Externalization Service as provided with SOMobjects 3.0 is an implementation of the *OMG Object Externalization Service* specification (OMG TC Document 940915).

Class Descriptions

The classes defined in the *OMG Object Externalization Service* specification all begin with **Cos**, for example **CosStream::Streamable**. The OMG classes are provided as specified by OMG[1] in IDL files beginning with the letters "omg", for example, the "omgestio" member in SOMMVS.SGOSIDL.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, the "somestio" member in SOMMVS.SGOSIDL.IDL.

The somStream::Streamable Class

Not all types of objects can be externalized or internalized. This is because the stream might not be able to access all the data in an object, cannot distinguish between essential and non-essential data, and cannot determine when to perform a deep or shallow copy[2] . To be externalized or internalized, the class of the object must inherit from the **somStream::Streamable** class. These objects are called *streamable*.

The Stream and StreamIO Classes

Up to this point, the stream has been treated as if it were a single object. However, the implementation of the stream actually consists of two objects:

- a **somExternalization::Stream** object
- a **somStream::StreamIO** object

There are two reasons for this separation:

1. Separate classes for different kinds of users.

   A stream actually has two kinds of users. One is the client which initiates the externalization or internalization, and the other is the streamable object which writes or reads its internal data. The **somExternalization::Stream** class has the methods that are useful to the client, and the **somStream::StreamIO** class has the methods that are useful to the streamable objects.

   Separate classes for orthogonal roles.

   The **somExternalization::Stream** object and the **somStream::StreamIO** object have distinct and independently extensible roles. The **somExternalization::Stream** object determines the semantics of the stream (see “somExternalization::Stream Implementations” on page 1-10), while the **somStream::StreamIO** object determines the storage medium and representation of the stream data. You can combine any two implementations of these to meet the semantics and representation requirements of your application.

Figure 1-1 on page 1-3 shows the relationships between the Externalization Service objects and classes.

---

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

2 More discussion of deep/shallow in “somExternalization::Stream Implementations” on page 1-10.

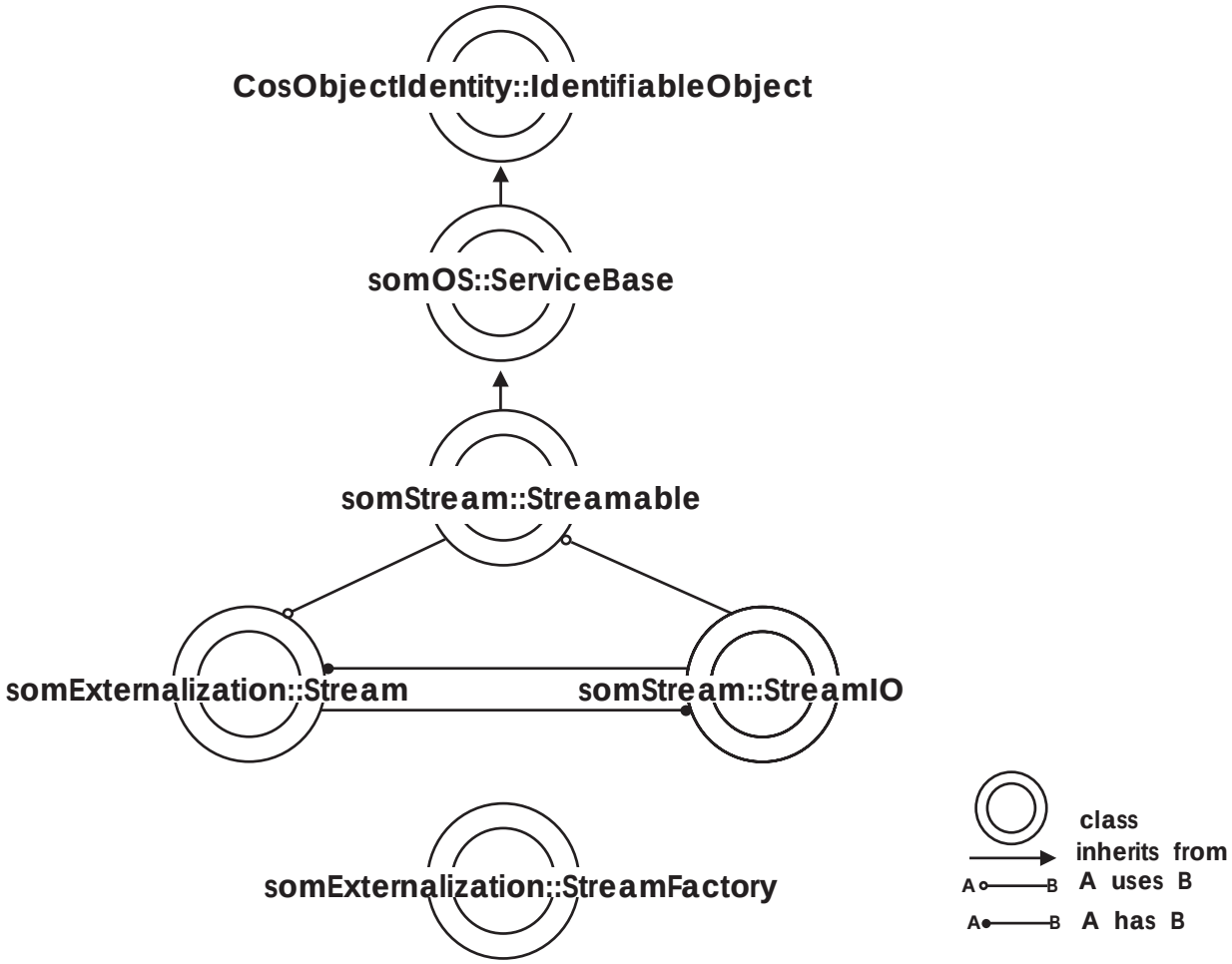

Figure 1-1. Object Externalization Service Class Diagram

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 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 that 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 would 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**[3] 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 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.

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

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 “Stream Semantics and Object Tags” on page 1-8.

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. When the **internalize** method is used to read new streamable objects from the stream, they are initialized using the **init_for_object_creation** method, not **somDefaultInit**.

How to Use a Stream

The following program example creates and initializes a streamable object of the **Item** class. It then creates a stream, externalizes the object to the stream, and uses the **internalize** method to create a new object.

```
#include <somd.xh>
#include <somestrm.xh>
#include <item.xh>
void main()
{
  Environment ev[1];
  Item *item1, *item2;
  somExternalization_Stream *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 somExternalization_Stream;
  strm->externalize(ev,item1);
  item2 = (Item*)(void*)strm->internalize(ev,NULL);
}
```

When the stream in this example is created, the representation type is not specified so the default implementation **somStream::StandardStreamIO** is used. If you need to create a stream with a specific implementation, use the **somExternalization::StreamFactory**. The **somExternalization::StreamFactory** method **create_with_types** takes two string parameters that are the names of the classes. For example, to create a **somExternalization::OSStream** object that uses a **somStream::MemoryStreamIO** object, use the following.

```
  somExternalization_Stream *strm;
  somExternalization_StreamFactory *sf;
  sf = new somExternalization_StreamFactory;
  strm = sf->create_with_types(ev,"somExternalization::OSStream",
                      "somStream::MemoryStreamIO");
```

The **somExternalization::StreamFactory** uses the **SingleInstance** metaclass, so it is not recommended that you free the factory. If you locate the **somExternalization::StreamFactory** using Life Cycle Service the object returned to you is the **somExternalization::StreamFactory** class object. You must call **somNew** on the class object to get the instance object.

A third way to create a stream is to create an instance of the **somStream::StreamIO** implementation of your choice. When you use the **write_object** method to externalize an object or **read_object** to internalize, **somStream::StreamIO** detects the absence of a **somExternalization::Stream** object, and creates (using **somNew**) a **somExternalization::Stream** object to use internally.

Whatever method you use to create the stream, the **somExternalization::Stream** object and **somStream::StreamIO** object are doubly linked together by a read-only attribute in each one. You can get this attribute, but it cannot be changed. When either the **somExternalization::Stream** or the **somStream::StreamIO** is freed, it also frees the other one. You can free one or the other, but do not free both.

Stream Representation

The **somStream::StreamIO** implementation determines the representation and storage medium of the data in the stream. IBM supplies three **somStream::StreamIO** implementations. All three of these implementations use a contiguous memory block as the buffer to store the data. You can call **_get_buffer** to get a copy of the memory buffer, and **_set_buffer** to replace the buffer with a block of memory you provide.

somStream::StandardStreamIO

The **somStream::StandardStreamIO** implementation stores data in its buffer in compliance with the *OMG Object Externalization Service* specification Standard Stream Data Format. The buffer has no prolog or header other than the tag on the first object in the buffer. Every data item in the buffer is prefixed with a one byte tag which indicates the type of the data. The numeric values are always stored in big endian format (see “Data Formats and Code Pages” on page 1-12). The floating point numbers are stored in IEEE 754 format. The character and string data are stored in the ASCII code page ISO Latin I (850).

somStream::StringStreamIO

The **somStream::StringStreamIO** implementation stores all the data in character form. The buffer does not contain any null characters (value zero), except the last character in the buffer is always a null. The following diagram shows the header of the buffer.

'S'	**'S'**	**code page id**
0x53	0x53	5 bytes

The first two characters are constant to aid in debugging. See “Data Formats and Code Pages” on page 1-12 for information about code page identifiers. The data items in the buffer do not have any tags or delimiters around them. The numeric types are written as fixed width character fields padded left with blanks. The widths of the numeric types are in the following table. Strings are length-prefixed with a five digit number and the terminating null is not stored.

Numeric Type	Width
short	6
long	11
float	15
double	23

somStream::MemoryStreamIO

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 format, and the floating point format can vary. The following diagram shows the header of the buffer.

'M'	'S'	endian	float format	code page id
0x4 D	0x5 3	0x42 or 0x4C	0x2E or 0x4D	5 bytes

The first two characters are constant to aid in debugging. The endian format is 0x42 ('B') for big endian or 0x4C ('L') for little endian. The floating point format is 0x2E for IEEE-754 or 0x4D for IBM-390 format. See “Data Formats and Code Pages” on page 1-12 for information about code page identifiers. The data items in the buffer do not have any tags or delimiters around them. Strings are length-prefixed with a four-byte long and the terminating null character is not stored.

Stream Semantics and Object Tags

The **somExternalization::Stream** implementation determines the format of objects in the stream, and this is referred to as the semantics of the stream. When an object is externalized, the **somExternalization::Stream** object (or subclass thereof) chooses an object format, selects the corresponding enum tag, and writes the tag to the stream buffer using the **write_object_tag** method. The *OMG Object Externalization Service* specification defines only three formats (see following table). The IBM -supplied implementation **somExternalization::Stream** uses only the OMG formats. The **somExternalization::OSStream** implementation also uses the fourth format. The tag value is an octet, so your own **somExternalization::Stream** implementation may use up to 255 object formats, but it is unlikely that you would need so many.

enum tag	OMG tag	Object Format Description
1	0xF0	Externalized object key and data
2	0x04	Externalized repeated reference data
3	0x05	Externalized NIL data
4	n/a	Externalized stringified reference

The first column of the table is the enum tag value which is passed to the write_object_tag method and returned by the **read_object_tag** method. The representation of the tag, i.e. that actual value that is written to the stream buffer, may vary depending on the **somStream::StreamIO** implementation. The second column of the table lists the tag values prescribed by the *OMG Object Externalization Service* specification for the representation of the tags in the buffer for the Standard Stream Data Format.

The following describes the use of each object format.

- The object key and data format is used on the first time an object is externalized to the stream within a given context (see the **begin_context** and **end_context** methods). The tag is written to the buffer followed by a

CosLifeCycle::Key that can be used by the **internalize** method to re-create the object. The key that is used is the external_form_id attribute of the streamable object. Following the key, the data for the object (if any) is written by its **externalize_to_stream** method.

- The repeated reference data format is used when an object is externalized more than once within the same context. The tag is written to the buffer followed by the object number which is an unsigned long. The object number is the index of the object in the context, zero is the first, one is the second and so forth.
- The NIL data format is used when the object is NULL (or equal to OBJECT_NIL). Only the tag is written to the buffer.
- The stringified reference format is used by the **somExternalization::OSStream** implementation of the **externalize_ref** method when the object has persistent references. See Chapter 5, “Object Services Server” on page 5-1 for information about persistent references.

Externalization of Object References

It is often the case that your streamable object contains references to other objects. You need to decide how to handle these references in the **externalize_to_stream** methods. If the object reference is not important, you may choose to do nothing with it in the **externalize_to_stream** method.

If you choose to write the object reference to the stream, use either the **write_object** or the **write_object_value** method. If the object reference is a *strong* reference, in the sense that your object is responsible for creating and destroying it, then you should call **write_object_value**. However, if the reference is a *weak* reference, in the sense that your object only uses the referenced object, then you should call **write_object**.

If the referenced object is not streamable, some **somExternalization::Stream** implementations will fail in the **write_object** or **write_object_value** method because they need to call the **externalize_to_stream** method on the object. If the contained object is not streamable, then your **externalize_to_stream** method instead should write a stringified reference using **write_string**, or get the essential data from the referenced object and write it to the stream.

Internalization of Object References

Your streamable object can read object references from the stream using the **read_object** method. Usually, you will want to first free the previous object reference using either **somFree** or **release** (depending on ownership of the object) as in the following example.

```
if (_myobj) _somFree(_myobj);
_myobj = stream->read_object( ev, ff, NULL);
```

In this case, the factory finder, ff, is used to create a new object. In some applications, however, you do not want to destroy and recreate the object. For example, the object may be shared (in a graph of objects), or location or performance may be important. You can use the last parameter of the **read_object** method to read data into an existing object, as follows.

```
_myobj = stream->read_object( ev, ff, _myobj);
```

This technique should be used with care. If the stream contains format tag 1, then the object is created if necessary and the data is read. However, if _myobj is not NULL and the stream contains format tags 2, 3, or 4, then the **read_object** method only verifies that _myobj is identical to the object reference indicated in the stream.

Delegation of Write and Read Object Methods

The methods relating to writing and reading objects on the **somStream::StreamIO** object are delegated to corresponding methods on the **somExternalization::Stream** object as follows.

- The **write_object** method in the **StreamIO** object simply calls the **externalize_ref** method on the **Stream** object.
- The **write_object_value** method in the **StreamIO** object simply calls the **externalize** method on the **Stream** object.
- If an existing object is passed to the **read_object** method, the **StreamIO** object calls the **internalize_existing** method on the **Stream** object. Otherwise, if OBJECT_NIL is passed, the **StreamIO** object calls the **internalize** method on the **Stream** object.

somExternalization::Stream Implementations

The **somExternalization::Stream** implementation determines the semantics of the stream which is the formats used to externalize objects. In general, all **somExternalization::Stream** implementations do the same thing for the externalization of *strong* references. Such references are stored in the buffer using format tags 1, 2, or 3. The **somExternalization::Stream** implementations differ in the implementation of the **externalize_ref** method which is used to externalize *weak* references. Some alternatives are:

- Call **externalize** (handle the same as "strong" references).
- Write the stringified reference (use format tag 4).
- If the reference is persistent, then write the stringified reference, else call **externalize**.

The **somExternalization::Stream** implementation uses the first alternative. This might be called a "deep copy". The second alternative might be called a "shallow copy." No implementation for this is provided, though you could create it. The **somExternalization::OSStream** implementation uses the third alternative.

Graphs of objects

If you have many objects that reference each other, the references form a directed graph. The graph may have cyclical and/or shared references. The *root* of the graph is a subset of objects from which all the other objects in the graph can be reached, either directly or indirectly. By externalizing the root objects, all the objects in the graph are externalized.

To re-create the objects, you call the **internalize** method for each root object. All the objects and references in the graph are reconstructed.

The Externalization Service accomplishes this by using the repeated reference object format. When an object is externalized it is compared to all the objects already written to the stream in the current context. This is implemented by calling the **is_identical** method for each object in an internal table. (For efficiency, the table also stores the constant_random_id of each object, so that the is_identical method is only called if the constant_random_id value matches.) If a match is found, then a repeated reference is written, else the object is written and added to the table. The **internalize** method also uses a table and each object read from the stream is added to the table. If a repeated reference is read from the stream, then the object number is used to return an object reference from the table.

Reading Objects with Procedural Code

Some applications may need to use procedural code to read the object data from the stream. To read the data, the procedural code must first "skip over" the object format header using the **read_object_tag** method. In the following example, myObject is a streamable object that writes a long integer and a string to the stream. To read this data, the procedural code must first call **read_object_tag**.

```
myStream->externalize(ev, myObject);
streamio = myStream->_get_streamio(ev);
tag = streamio->read_object_tag(ev,&objnum,&key);
n1 = streamio->read_long(ev);
s1 = streamio->read_string(ev);
```

The reverse is also possible. The procedural code calls the **write_object_tag** method and writes data to the stream which later can be internalized by a streamable object.

When the Stream is 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

An example of a read following a write would be calling the **internalize** method after the **externalize** method. 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. When the stream is reset (implicitly or explicitly) the **end_context** method is implicitly called.

Note: If your **somStream::StreamIO** is the top of a "diamond inheritance" structure, then you must ensure that the **reset** method is done prior to using the

already_streamed method. This **reset** is implicitly done with the **externalize** and **internalize** methods. It must be explicitly coded in other cases.

Data Formats and Code Pages

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.

Many formats exist for storing float and double data types. The Externalization Service uses the IEEE-754 floating point format and IBM-390 floating point formats.

A code page is a mapping of numeric values to characters. Different computers, and different processes in the same computer may use different code pages. The implementation of some streams puts a code page identifier in the header of the stream. This allows the character data that is written by one process to be converted when it is read by a different process. The code page identifier is 5 characters that are ASCII digits, 0x30 ('0') through 0x39 ('9'). The following table shows some example code page identifiers. Currently, not all code page conversions are supported by the Externalization Service.

Table 1-1. Code page identifiers

Code page id	Description
00850	ASCII Latin 1, multilingual
00437	ASCII USA
00852	ASCII Latin 2
00863	ASCII Canadian French
00037	EBCDIC USA (OS/400)
01047	EBCDIC USA (OS/390)
00284	EBCDIC Spain, Latin America

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.
- No implementation of **CosExternalization::FileStreamFactory** is provided, because no file stream is provided.
- The following comments apply to the **StandardStreamIO**.
- The externalized object data format (tag 1) stores both the id and kind values of the key. They are stored id1, kind1, id2, kind2, etc. and each string is null-terminated. The key length is a one byte prefix that indicates the total number of strings. Since the id and kind strings are paired, the length is always even, and at most 127 pairs can be written.
- The externalized repeated reference data format stores an object number, four bytes, in big endian format. The object number indicates the order in the buffer

of the object. The first object in the buffer is number zero, the second is 1, etc. Only the objects stored with externalized object data format (tag 1) are counted as objects in the buffer. The object numbers do not start again at zero after the **end_context** method is called, because there is no provision for storing begin and end context markers in the buffer.

- The externalized NIL data tag is written to the stream when **write_object**, **write_object_value**, or **write_string** is passed a NULL pointer. When the externalized NIL data tag is read by **read_object** or **read_string** it returns NULL.
- If a new tag, such as tag 4, is passed to the **write_object_tag** method, then the tag is stored in the buffer using the **write_octet** method.

Chapter 2. 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 most 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. If both objects are identifiable but have different CONSTANT_RANDOM_ID values, then they are not the same object and there is no need to consult the server (presumably across the network). The operation can reject the current object in the local collection and proceed to the next.
6. 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.
7. 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

Mixin the **somOS::ServiceBase** class when a consistent interface and implementation for establishing identity of object instances for both local and distributed SOM objects is desired.

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 2-1 on page 2-4 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.

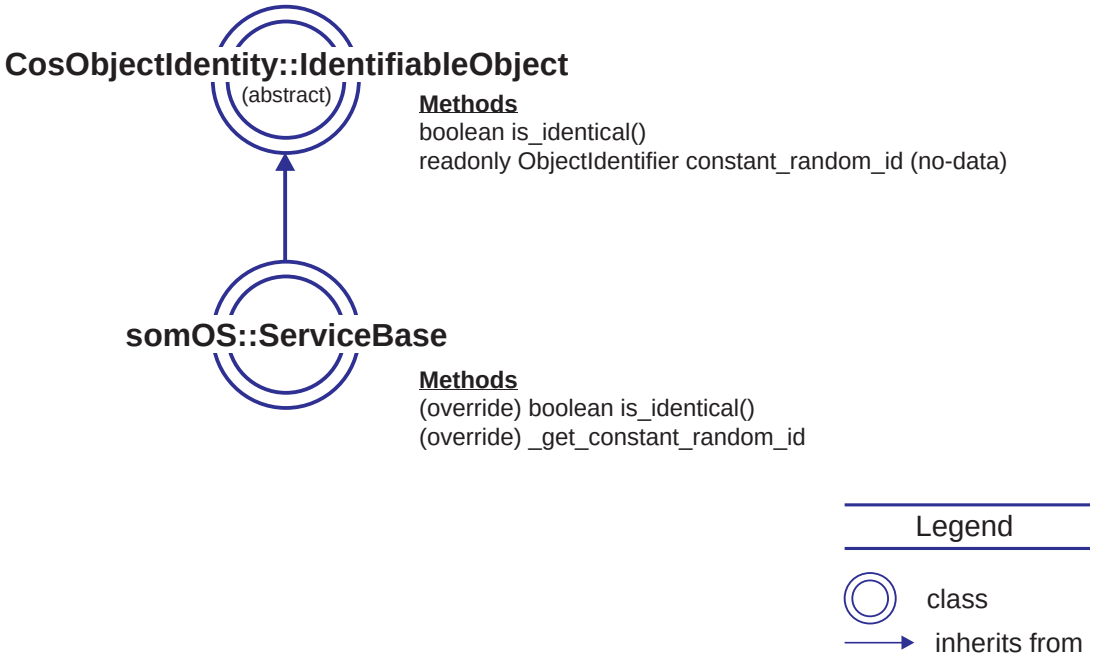

Figure 2-1. somOS::ServiceBase Class Diagram

Pros and Cons

Some benefits and liabilities 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 3. Life Cycle Service

The Life Cycle Service enables clients to create and remove local or distributed objects without having to be aware of the location of the objects. The Life Cycle Service also provides a mechanism that can be used to customize your development environment and manage the creation of objects on your servers.

The Life Cycle interfaces inherit from those defined by the Object Management Group (OMG), which allows users to meet OMG compliance requirements.

The rest of this chapter contains the following topics:

Before You Start

The Life Cycle Service can be very complex since it interacts with several other services and components. It will be very helpful to review the following suggested readings before reading this chapter:

- Common Object Services Specification Volume 1 (OMG Document Number 94-1-1) which contains the standard definition of the CosLife Cycle module.
- "Distributed SOM" in *OS/390 SOMobjects Programmer's Guide*
- Chapter 4, "Naming Service" on page 4-1
- Chapter 5, "Object Services Server" on page 5-1

The following documents will be very handy references when using the Life Cycle Service. You will want to familiarize yourself with them also.

- Chapter 3, "Life Cycle Service"
- "Error Reporting and Troubleshooting Hints" in *OS/390 SOMobjects Messages, Codes, and Diagnosis*
- Error Codes in *OS/390 SOMobjects Messages, Codes, and Diagnosis*.

Setting Up and Using Life Cycle with a Simple Example

There are administrative tasks to setting up Life Cycle as well as client application tasks. The following will describe a scenario of using the Life Cycle Service that describes both sets of tasks.

Scenario:

- A third party vendor builds two classes called "Cat" and "Dog".
- A system administrator decides that these classes should be put on separate servers because they will be highly accessed.

- A client wants to use the “Cat” class and create a cat named “9Lives” and wants to use the “Dog” class and create a dog named “Lucky”.

The following are the tasks that will be described to enable this scenario.

1. Building a “Cat” and “Dog” class library (done by the class library builder).
2. Configure system to put “Cat” and “Dog” on two different servers (done by the system administrator).
3. Customize Life Cycle to only look at these two servers.
4. Accessing “Cat” and “Dog” as a client, giving them names (done by the client application builder).

Building a “Cat” and “Dog” Class Object

The following steps need to be done by the class library builder of “Cat” and “Dog”:

- Create the IDL
- Take the SOM compiled output and create the implementation code
- Create the DLL and definition side deck.

Create the IDL

The following is the IDL for “Cat”:

```
//   Description
//   -----------
//   Provides a simple SOMObject (Cat). Cat inherites from lifecycleobject.
//


#ifndef SOM_Cat_idl
#define SOM_Cat_idl

#include <somlc.idl>


interface Cat : somLifeCycle::LifeCycleObject
{
  attribute string name;
  // Name of Cat.

 #ifdef __SOMIDL__
  implementation {
    releaseorder: _get_name, _set_name;

    override: somDefaultInit;
    override: somDestruct;
    name: noset;
    name: noget;

    //# Class Modifiers
    dllname = "pet.dll";
    //# Need to tell the proxy class to inherit from
    baseproxyclass = "somLifeCycle::LifeCycleObject__Proxy";


    //# Parameter memory management is per CORBA except as inidcated otherwise
    memory_management = corba;
    };
#endif /* __SOMIDL__ */
};


#endif  /* SOM_Cat_idl */
```

The following is the IDL for “Dog”:

```
//   Description
//   -----------
//   Provides a simple SOMObject (Dog). Dog inherites from lifecycleobject.
//


#ifndef SOM_Dog_idl
#define SOM_Dog_idl

#include <somlc.idl>


interface Dog : somLifeCycle::LifeCycleObject
{
  attribute string name;
  // Name of Dog.

 #ifdef __SOMIDL__
  implementation {
    releaseorder: _get_name, _set_name;

    override: somDefaultInit;
    override: somDestruct;
    name: noset;
    name: noget;

    //# Class Modifiers
    dllname = "pet.dll";
    //# Need to tell the proxy class to inherit from
    baseproxyclass = "somLifeCycle::LifeCycleObject__Proxy";


    //# Parameter memory management is per CORBA except as inidcated otherwise
    memory_management = corba;
    };
#endif /* __SOMIDL__ */
};


#endif  /* SOM_Dog_idl */
```

Take the SOM Compiled Output and Create the Implementation Code

The following will be the C implementation code for “Cat”, “Dog”, and “INITPET”. INITPET is an initialization function for the “Pet DLL”. The initialization function is called by SOMobjects whenever it loads a class library.

The implementation code for “Cat” is:

```
#ifndef SOM_Module_cat_Source
#define SOM_Module_cat_Source
#endif
#define Cat_Class_Source
#include "DD:IH(cat)"
/*
 * Name of Cat.
 */
SOM_Scope string  SOMLINK _get_name(Cat *somSelf,  Environment *ev)
{
    CatData *somThis = CatGetData(somSelf);
    CatMethodDebug("Cat","_get_name");

    /* Return statement to be customized: */
    return(_name);
}
/*
 * Name of Cat.
 */
SOM_Scope void  SOMLINK _set_name(Cat *somSelf,  Environment *ev,
                                  string name)
{
    CatData *somThis = CatGetData(somSelf);
    CatMethodDebug("Cat","_set_name");
    if (_name != NULL)
      SOMFree(_name);
    _name= (string) SOMMalloc(strlen(name)+1);
    memcpy(_name, name, (strlen(name) + 1));
    return;

}

SOM_Scope void SOMLINK somDefaultInit(Cat *somSelf, som3InitCtrl* ctrl)
{
    CatData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CatMethodDebug("Cat","somDefaultInit");
    Cat_BeginInitializer_somDefaultInit;

    Cat_Init_somLifeCycle_LifeCycleObject_somDefaultInit(somSelf, ctrl);
    /*
     * local Cat initialization code added by programmer
     */
    _name= NULL;
    return;
}

SOM_Scope void SOMLINK somDestruct(Cat *somSelf, octet doFree,
                                   som3DestructCtrl* ctrl)
{
    CatData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    CatMethodDebug("Cat","somDestruct");
    Cat_BeginDestructor;
    /*
     * local Cat deinitialization code added by programmer
     */
    if (_name != NULL)
      SOMFree(_name);

    Cat_EndDestructor;
}
```

The implementation code for “Dog” is:

```
#ifndef SOM_Module_dog_Source
#define SOM_Module_dog_Source
#endif
#define Dog_Class_Source
#include "DD:IH(dog)"
/*
 * Name of Dog.
 */
SOM_Scope string  SOMLINK _get_name(Dog *somSelf,  Environment *ev)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog","_get_name");

    /* Return statement to be customized: */
    return(_name);
}
/*
 * Name of Dog.
 */
SOM_Scope void  SOMLINK _set_name(Dog *somSelf,  Environment *ev,
                                  string name)
{
    DogData *somThis = DogGetData(somSelf);
    DogMethodDebug("Dog","_set_name");
    if (_name != NULL)
      SOMFree(_name);
    _name= (string) SOMMalloc(strlen(name)+1);
    memcpy(_name, name, (strlen(name) + 1));
    return;
}
SOM_Scope void SOMLINK somDefaultInit(Dog *somSelf, som3InitCtrl* ctrl)
{
    DogData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    DogMethodDebug("Dog","somDefaultInit");
    Dog_BeginInitializer_somDefaultInit;

    Dog_Init_somLifeCycle_LifeCycleObject_somDefaultInit(somSelf, ctrl);

    /*
     * local Dog initialization code added by programmer
     */
    _name= NULL;
    return;
}

SOM_Scope void SOMLINK somDestruct(Dog *somSelf, octet doFree,
                                   som3DestructCtrl* ctrl)
{
    DogData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    DogMethodDebug("Dog","somDestruct");
    Dog_BeginDestructor;
    /*
     * local Dog deinitialization code added by programmer
     */
    if (_name != NULL)
     SOMFree(_name);

    Dog_EndDestructor;
}
```

The implementation code for “INITPET” follows:

```
#include "dog.h"
#include "cat.h"
#ifdef __IBMC__
     #pragma linkage (SOMInitModule,system)
#endif
SOMEXTERN void SOMLINK SOMInitModule(long majorVersion,
                                                long minorVersion,
                                                string className)
{
            SOM_IgnoreWarning(majorVersion);
            SOM_IgnoreWarning(minorVersion);
            SOM_IgnoreWarning(className);

            DogNewClass(Dog_MajorVersion, Dog_MinorVersion);
            CatNewClass(Cat_MajorVersion, Cat_MinorVersion);
}
```

Create the DLL and Definition Side Deck.

The next step is to C compile the implementation code for “Cat”, “Dog” and “INITPET”. For more information on how to do this see *OS/390 SOMobjects Programmer's Guide*.

Configure the System to Put “Cat” and “Dog” on Two Different Servers

The following steps need to be taken by the system administrator to put “Cat” and “Dog” on two different servers.

1. Register the classes and servers using REGIMPL and Workload Manager (WLM).
2. Customize the GenericFactory and FactoryFinder objects
   a. “Get GenericFactory Object from the Global Root Naming Context:”
   b. “Instantiate a Location object:” on page 3-9
   c. “Set the Location Object up with a List of Servers” on page 3-10. This will restrict the search for factories to these servers.
   d. “Obtain the FactoryFinder Object from the GenericFactory:” on page 3-10
   e. “Register this Location Object with the Factory Finder Object” on page 3-10

Register the Classes and Servers Using REGIMPL and Workload Manager (WLM)

The first task the Life Cycle system administrator has is to register the classes and servers with REGIMPL and WLM. The server names will be called CATSVR and DOGSVR. For detailed information on how to do this, see *OS/390 SOMobjects Configuration and Administration Guide*.

Customize the GenericFactory and FactoryFinder Objects

The next task is to customize the GenericFactory and FactoryFinder objects. The following steps demonstrate this.

Get GenericFactory Object from the Global Root Naming Context: Figure 3-1 on page 3-8 shows how to get the global root naming context.

```
/************************************************************/
/* Administrative setup program:                            */
/*                                                          */
/* This program will:                                       */
/* 1)Customize the GenericFactory and FactoryFinder objects*/
/*   instantiated by the SOM@CFG program.                   */
/*   These objects reside on the namingServer. SOM@CFG      */
/*   binds the GenericFactory object to the Global Node     */
/*   Context with the name of "somLifeCycleGenericFactory".*/
/*   Client Programs understand this and will look here to  */
/*   get the GenericFactory object reference. SOM@CFG reg-  */
/*   isters the FactoryFinder object it instantiates on the*/
/*   namingServer with the GenericFactory object it instan-*/
/*   tiates on the namingServer.                            */
/*                                                          */
/************************************************************/
#include <stdio.h>
#include <somd.h>
#include <implrep.h>
#include <impldef.h>
#include <omglc.h>
#include <somlc.h>

void main(int argc, char **argv)
{
    Environment *ev;
    ImplRepository *ir;
    ImplementationDef *impldef;
    string hostname;
    boolean addingServer;
    ExtendedNaming_ExtendedNamingContext    *localrootNC;
    ExtendedNaming_ExtendedNamingContext    *globalrootNC;
    CosNaming_Name  name;
    SOMDServer      *nameServer;
    somLifeCycle_GenericFactory  *nameServerGF;
    somLifeCycle_FactoryFinder   *nameServerFF;
    somLifeCycle_ServerSetLocation  *serverloc;
    ServerId Dogservid;
    string dogservid = "87654321-87654321-7f-00-1234567890aa";
    ServerId Catservid;
    string catservid = "87654321-87654321-7f-00-1234567890ab";

    ev = somGetGlobalEnvironment();
    SOMD_Init(ev);
    if (ev->_major != NO_EXCEPTION){
       somExceptionFree(ev);
       printf("Error: Unable to update the regimpl!\n");
       return;
    }
```

Figure 3-1 (Part 1 of 2). Getting the global root naming context.

```
    /*********************************************************/
    /* Start: Customize GenericFactory and FactoryFinder    */
    /*        Objects on namingServer.                       */
/*********************************************************************/
/* 2a. Get the Global Root Naming Context.                           */
/*  This is where we can find                                        */
/*  GenericFactory and FactoryFinder objects. These                  */
/*  objects were instantiated by SOM@CFG program during SOM          */
/*  configuration. The GenericFactory object was bound to the        */
/*  Global Root Naming Context under the "well known" name of        */
/*  LifeCycleGenericFactory. The FactoryFinder object was            */
/*  registered in the GenericFactory object.                         */
/*********************************************************************/
localrootNC= (ExtendedNaming_ExtendedNamingContext *)ORB_resolve_initial_references(
                                SOMD_ORBObject, ev, "NameService");
name._buffer = SOMMalloc(sizeof(CosNaming_NameComponent));
name._length = name._maximum = 1;
name._buffer->id = ".:";
name._buffer->kind = "";
globalrootNC= (ExtendedNaming_ExtendedNamingContext *)_resolve(localrootNC, ev, &name);
/*********************************************************************/
/* Set up the name to retrieve the GenericFactory object          */
/*  reference out of the namingServer. This is a "well known"     */
/*  name. All client programs will know this name and use it to   */
/*  get to  the GenericFactory object.                            */
/*********************************************************************/
name._buffer->id = "LifeCycleGenericFactory";
name._buffer->kind = "";

/*********************************************************************/
/* Get the GenericFactory object reference out of the namingServer*/
/*********************************************************************/
nameServerGF= (somLifeCycle_GenericFactory *) _resolve(globalrootNC, ev, &name);
```

Figure 3-1 (Part 2 of 2). Getting the global root naming context.

Instantiate a Location object: Figure 3-2 shows how to instantiate a location object.

```
/*********************************************************************/
/* 2b. Instantiate a Location object.                             */
/*     a list of servers. This will restrict the search for        */
/*     factories to these servers.                                */
/* Now we want to do some customization. We want to restrict      */
/* our GenericFactory to search for factories only on the CATSVR   */
/* and the DOGSVR. We will do this by building a Location Object  */
/* on the namingServer and adding server ids to it to limit the   */
/* search to these two servers.                                   */
/*                                                                */
/*  1) Get a somdserver object that is aimed at the namingServer. */
/*  2) Create a ServerSetLocation object on the namingServer.     */
/*********************************************************************/
nameServer= _somdFindServerByName(SOMD_ObjectMgr, ev, "NAMINGSERVER");
serverloc= (somLifeCycle_GenericFactory *) _somdCreateObj (nameServer, ev,
                                        "somLifeCycle::ServerSetLocation","");
serverloc=(somLifeCycle_ServerSetLocation *)((void *)_init_for_object_creation(serverloc, ev));
```

Figure 3-2. Instantiating a location object.

Set the Location Object up with a List of Servers: This will restrict the search for factories to these servers. Figure 3-3 on page 3-10 shows how to set up the location object up with a list of servers.

```
/*****************************************************************/
/* 2c. Set the location object up with a list of servers. This will*/
/*     restrict the search for factories to these servers.         */
/* Allocate storage for the individual ServerIds to be added to the*/
/* ServerSetLocation object.                                       */
/*****************************************************************/
Dogservid = (ServerId) SOMMalloc(strlen(dogservid) + 1);
strcpy(Dogservid, dogservid);
Catservid = (ServerId) SOMMalloc(strlen(catservid) + 1);
strcpy(Catservid, catservid);

/*****************************************************************/
/* Add server ids to the Location object. These are the servers    */
/*  we want the search for factories to include.                    */
/*****************************************************************/
_add_server_id(serverloc, ev, Dogservid);
_add_server_id(serverloc, ev, Catservid);
```

Figure 3-3. Setting the location object up with a list of servers.

Obtain the FactoryFinder Object from the GenericFactory: Figure 3-4 shows how to obtain the FactoryFinder Object from the GenericFactory.

```
/*****************************************************************/
/* 2d. Obtain the FactoryFinder object from the GenericFactory     */
/*  object obtained above.                                         */
/*****************************************************************/
nameServerFF= _get_factory_finder(nameServerGF, ev);
/*****************************************************************/
```

Figure 3-4. Obtaining the FactoryFinder object from the GenericFactory.

Register this Location Object with the Factory Finder Object Figure 3-5 shows how to register this location object with the Factory Finder object.

```
/* 2e. Register this location object with the Factory Finder object.*/
/*  The Factory Finder object                                      */
/*  is registered in the Generic Factory object. The FactoryFinder  */
/*  will only search the servers contained in the location object   */
/*  that is registered with the factoryfinder.                     */
/*****************************************************************/
_set_location_scope(nameServerFF, ev, serverloc);
    /*********************************************************/
    /* End:   Customize GenericFactory and FactoryFinder    */
    /*        Objects on namingServer.                       */
    /*********************************************************/
}
```

Figure 3-5. Registering this location object with the Factory Finder object.

Accessing “Cat” and “Dog” as a Client, Giving Them Names

A client wants to use the “Cat” class and create a cat named “9Lives” and wants to use the “Dog” class and create a dog named “Lucky”. This is done with two basic tasks:

1. Obtain a GenericFactory object from the global root naming context.
2. Create a cat and dog object using the GenericFactory.
3. Destroy the cat and dog objects.

Obtain a GenericFactory Object from the Global Root Naming Context

Figure 3-6 on page 3-12 shows how to obtain a GenericFactory object from the global root naming context.

```
#include "dog.h"
#include "cat.h"
#include <somd.h>
#include <stdio.h>
#include <somlc.h>
#include <somlcdef.h>
#include <xnaming.h>
#include <naming.h>

main ()
{
    ExtendedNaming_ExtendedNamingContext    *globalrootNC;
    ExtendedNaming_ExtendedNamingContext    *localrootNC;
    CosNaming_Name  name;
    somLifeCycle_GenericFactory  *nameServerGF;
save    CosNaming_NameComponent  elementsff1“;
    CosLifeCycle_Key   myKey;
    Environment *ev;
    string anmlname;
    Dog  *mydog;
    Cat   *mycat;

    myKey._maximum= 1;
    myKey._length= 0;
    myKey._buffer= elements;
    ev= SOM_CreateLocalEnvironment();
    SOMD_Init(ev);
    /*****************************************************************/
    /*  1. Obtain a GenericFactory object from the Global Root Naming */
    /*     Context.                                                   */
    /*  The GenericFactory object was                                 */
    /*  instantiated by SOM@CFG program during SOM                    */
    /*  configuration. The GenericFactory object was bound to the     */
    /*  Global Root Naming Context under the "well known" name of     */
    /*  LifeCycleGenericFactory. The FactoryFinder object was         */
    /*  registered in the GenericFactory object.                      */
    /*****************************************************************/
    localrootNC=  (ExtendedNaming_ExtendedNamingContext *)              ORB_resolve_initial_references(
                                    SOMD_ORBObject, ev, "NameService");
    name._buffer = SOMMalloc(sizeof(CosNaming_NameComponent));
    name._length = name._maximum = 1;
    name._buffer->id = ".:";
    name._buffer->kind = "";
    globalrootNC= (ExtendedNaming_ExtendedNamingContext *) _resolve(localrootNC, ev, &name);

   /*****************************************************************/
   /* Set up the name to retrieve the GenericFactory object          */
   /*  reference out of the namingServer. This is a "well known"     */
   /*  name. All client programs will know this name and use it to   */
   /*  get to  the GenericFactory object.                            */
   /*****************************************************************/
   name._buffer->id = "LifeCycleGenericFactory";
   name._buffer->kind = "";

   /*****************************************************************/
   /* Get the GenericFactory object reference out of the namingServer*/
   /*****************************************************************/
   nameServerGF= (somLifeCycle_GenericFactory *) _resolve(globalrootNC, ev, &name);
```

Figure 3-6. Obtaining a GenericFactory object from the global root naming context.

Create a Cat and Dog Object Using the GenericFactory

Figure 3-7 shows how to create a cat and dog object using the GenericFactory. It also shows how to name the cat object “9Lives” and the dog object “Lucky”.

```
   /*****************************************************************/
   /*  2. Create a dog and cat object using the Generic Factory.    */
   /*****************************************************************/
   myKey._buffer[0].kind= KIND_OBJ_INF;
   myKey._buffer[0].id= "Dog";
   mydog= _create_object(nameServerGF, ev, &myKey, NULL);
   myKey._buffer[0].kind= KIND_OBJ_INF;
   myKey._buffer[0].id= "Cat";
   mycat= _create_object(nameServerGF, ev, &myKey, NULL);

   /*****************************************************************/
   /* Give the dog and cat a name, then read out the name and print  */
   /*****************************************************************/
    Dog__set_name(mydog,ev,"lucky");
    anmlname= Dog__get_name(mydog,ev);
    printf("name is %s\n", anmlname);
    Cat__set_name(mycat,ev,"9lives");
    anmlname= Cat__get_name(mycat,ev);
    printf("name is %s\n", anmlname);

   /*****************************************************************/
   /* Now we are done with the cat and dog objects. Destroy the     */
   /* Cat and Dog objects.                                          */
   /*****************************************************************/
    if (mydog != NULL)
      Dog_remove(mydog, ev);
    if (mycat != NULL)
      Cat_remove(mycat, ev);

    SOMD_Uninit(ev);
    SOM_DestroyLocalEnvironment(ev);
}
```

Figure 3-7. Creating a cat and dog object using the GenericFactory.

The next section will describe in more detail the Life Cycle interfaces.

Life Cycle Service Interfaces

The Life Cycle Service contains eight interfaces:

- **somLifeCycle::GenericFactory**
- **somLifeCycle::FactoryFinder**
- **somLifeCycle::LifeCycleObject**
- **somLifeCycle::Location**
- **somLifeCycle::ServerSetLocation**
- **somLifeCycle::ConstraintBuilder**
- **somLifeCycle::FactoryFilter**
- **somLifeCycle::GenericFactoryAbstraction**

somLifeCycle::GenericFactory, **somLifeCycle::FactoryFinder** and **somLifeCycle::LifeCycleObject** all descend from interfaces defined by the **CosLifeCycle** module as specified by OMG. The other five interfaces are defined by the **somLifeCycle** module to assist with object creation, locating Factory objects

and environment configuration. They do not descend from the CosLifeCycle module.

The complete interface definitions may be found in the **somlc.idl** and the **omglc.**idl files for the **somLifeCycle** and **CosLifeCycle** modules, respectively.

For ease of reading, the remainder of this chapter will not qualify the interface names with the module name. The reader may assume that the interfaces discussed are the **somLifeCycle** interfaces unless specifically noted as otherwise.

GenericFactory

GenericFactory provides an interface that clients can use to create objects. It can create objects of many different interfaces on many different servers. Generally, the objects created support the **LifeCycleObject** interface, but this is not a requirement. Associated with a **GenericFactory** is a **FactoryFinder**, which is used to locate a more specific factory that can support the creation request.

The **GenericFactory** interface provides an implementation for the following methods:

- **create_object**
- **supports**
- **set_factory_finder**
- **get_factory_finder**

FactoryFinder

The **FactoryFinder** interface allows users to locate specific factories to invoke methods on. The **create_object** method on the **GenericFactory** uses the **FactoryFinder** registered with the **GenericFactory** to locate more specific factories to use for object creation. The **FactoryFinder** can also be used directly by clients who desire to interface with the more specific factories directly rather than making calls to a **GenericFactory**. The DSOM Factory Service provides the underlying mechanism that **FactoryFinder** uses to locate more specific factories.

Several Life Cycle objects may be registered with the **FactoryFinder** that will control which factories are located and returned to the caller. The **Location** object (actually always a subclass of **Location**, since **Location** is an abstract interface) registered with the **FactoryFinder** is used to specify a set of servers that make up the scope for object creation. Only factories that are able to operate on the designated servers will be returned. Any number of **ConstraintBuilder** and **FactoryFilter** objects (actually always a subclass of **ConstraintBuilder** or **FactoryFilter** since they are abstract) may also be associated with the **FactoryFinder**. The **FactoryFinder** can use them to limit the set of factories identified, based on the policy encapsulated within these objects.

The **FactoryFinder** interface provides an implementation for the following methods:

- **find_factories**
- **find_factory**
- **set_location_scope**
- **get_location_scope**
- **add_constraint_builder**
- **remove_constraint_builder**
- **list_constraint_builders**

- **add_factory_filter**
- **remove_factory_filter**
- **list_factory_filters**

Location

Location provides an interface that **FactoryFinder** uses to identify the scope of the operation. This interface is abstract and must be subclassed. The Life Cycle Service provides a subclass implementation of **Location** called **ServerSetLocation**. If the **ServerSetLocation** interface doesn't meet the needs of the user, the user may create other subclass implementations of **Location**.

Should it be desirable to create your own subclass for **Location**, the subclass should provide an implementation of the **list_server_ids** method, since the **FactoryFinder** invokes this method to retrieve the servers associated with the **Location** object registered with the **FactoryFinder**. This interface provides other methods and is potentially useful for clients to interface with directly. However, its primary reason for existence is to be used by the **FactoryFinder**.

The **Location** interface defines the following methods that may be overridden and implemented by subclasses:

- **list_server_ids**
- **is_server_in_location**
- **is_object_in_location**
- **location_union**
- **location_intersection**

The Life Cycle Service does provide an implementation for the **is_server_in_location** and **is_object_in_location** methods, so there is no need to override those methods unless the user wishes to change the implementation details of the methods.

ServerSetLocation

ServerSetLocation provides a concrete implementation of the **Location** interface. It maintains a list of servers that comprise the location scope for operations. **ServerSetLocation** overrides and implements the **list_server_ids**, **location_union** and **location_intersection** methods introduced by the **Location** interface. It also defines the following methods and provides an implementation for each:

- **add_server_id**
- **add_server_ids**
- **remove_server_id**
- **remove_server_ids**

ConstraintBuilder

ConstraintBuilder provides an abstract way to define additional constraints to be considered by the DSOM Factory Service during factory searches. The **ConstraintBuilder** is a powerful way to build *static* controls into object creation since they will impact the number of factories identified by DSOM Factory Service. It also provides performance controls since the user may specify conditions that will decrease the number of factories found and returned by the DSOM Factory Service. This is important because the DSOM Factory Service will create an instance of every factory found (if it does not already exist) and return actual object

references to the client so that the client is able to invoke methods on the factories immediately.

The **ConstraintBuilder** interface is designed to be subclassed to meet the specific environment requirements of the user. The user may subclass **ConstraintBuilder** and implement the **build_constraint** method to return a valid **ExtendedNaming::ExtendedNamingContext::Constraint** that defines a specific policy to consider in factory searches. (See Chapter 4, “Naming Service” on page 4-1 of Programmer' s Guide for Object Services for more information on the definition of a valid **ExtendedNaming::ExtendedNamingContext::Constraint**.)

If a **ConstraintBuilder** is registered with a **FactoryFinder** object, the **FactoryFinder** object will automatically build the **ExtendedNaming::ExtendedNamingContext::Constraint** returned from invocations of the **build_constraint** method for the registered **ConstraintBuilder** into the final **ExtendedNaming::ExtendedNamingContext::Constraint** built by the **FactoryFinder** and passed to the DSOM Factory Service in factory searches. This means that the static policies will automatically be included for you in **find_factories** and **find_factory** requests, as well as **create_object** and **supports** since the **GenericFactory** will have a **FactoryFinder** registered with it. The **ConstraintBuilder** subclasses may also be useful in client situations for building an **ExtendedNaming::ExtendedNamingContext::Constraint** for direct client interaction with the DSOM Factory Service.

An existing **ConstraintBuilder** may be registered with any number of **FactoryFinder** objects, and there is no limit to the number of subclasses that may be defined. The **ConstraintBuilder** introduces the following method, but no meaningful default implementation is provided since it should be specific to the user environment:

- **build_constraint**

FactoryFilter

FactoryFilter provides an abstract way to define additional filtering capabilities that will be considered during factory searches. The **FactoryFilter** is a powerful way to build *dynamic* controls into object creation since the number of factories originally identified by the DSOM Factory Service may be reduced based on the policy of the **FactoryFilter**.

The **FactoryFilter** interface is designed to be subclassed to meet the specific environment requirements of the user. The user may subclass **FactoryFilter** and implement the **filter_factories** method to purge any factories that do not meet the policies enforced by the **FactoryFilter**. If the **FactoryFilter** is registered with any **FactoryFinder** objects, the **FactoryFinder** objects will automatically invoke the **filter_factories** method for the **FactoryFilter**, passing in the sequence of factories that were found by the DSOM Factory Service. The **FactoryFilter** will then evaluate the factories passed in and eliminate any that don't meet the requirements of the **FactoryFilter**. This means that the dynamic policies will automatically be included for you in **find_factories** and **find_factory** requests, as well as **create_object** and **supports** since the **GenericFactory** will have a **FactoryFinder** registered with it. The **FactoryFilter** subclasses may also be useful from a client perspective after interfacing directly with the DSOM Factory Service to reduce the number of factories returned.

An existing **FactoryFilter** may be registered with any number of **FactoryFinder** objects, and there is no limit to the number of subclasses that may be defined. The **FactoryFilter** introduces the following method, but no default implementation is provided since it should be specific to the user environment:

- **filter_factories**

GenericFactoryAbstraction

GenericFactoryAbstraction provides an abstract interface used by the **GenericFactory** interface to create and initialize objects based on the Factory object. It provides two methods that are called from within the **create_object** implementation; one that performs the object creation (**real_object_creation**) and one that initializes the object based on specifications from the user (**initialize_object**). These are separated out from the **create_object** code to allow these functions to be easily overridden and specialized by subclasses. The methods provided by this interface are never intended to be called directly by clients. Only the **create_object** method should invoke these methods.

The following methods are introduced by the **GenericFactoryAbstraction** interface:

- **real_object_creation**
- **initialize_object**

The **initialize_object** method is abstract, and no default implementation is provided. A default implementation is provided for the **real_object_creation** method that will handle creation of objects with a factory that is a **SOMClass** or another **GenericFactory**. The user may want to subclass the **GenericFactory** interface and override the **real_object_creation** and **initialize_object** methods (since **GenericFactory** is a subclass of **GenericFactoryAbstraction**) to incorporate handling of specific factories and initializers. See “Subclassing the GenericFactory Interface” on page 3-35 for more details on this process.

LifeCycleObject

LifeCycleObject provides an interface that objects may want to inherit to comply with OMG standards. It allows the objects to be removed, moved, or copied through methods directly on the object itself. Move and copy are not implemented in this release of **somLifeCycle**. It is possible for the user to subclass the interface and override and implement the methods, but it is not a trivial undertaking.

It is important to mention that any interfaces inheriting the **LifeCycleObject** interface must list the **LifeCycleObject__Proxy** interface as the baseproxyclass in the IDL. The **LifeCycleObject__Proxy** interface is not designed to be instantiated and no methods should be invoked directly on the interface. It is used solely by the **LifeCycleObject** interface to ensure that remote requests are processed correctly.

Interfaces that inherit from the **LifeCycleObject** interface must also list the **LifeCycleObject** interface first in the list of inherited interfaces in the IDL. This will ensure that the **LifeCycleObject** interface methods will control the destruction of the object for the C++ **delete** operation and **somFree** or **remove** method calls.

The following methods are introduced by the **LifeCycleObject** interface:

- **remove**
- **copy**
- **move**

Important Life Cycle Relationships

The Life Cycle Service relies on relationships both internally and externally. The **somLifeCycle** interfaces have many relationships, and the Life Cycle Service itself has relationships with other components and services. This section will describe these relationships to help you better understand the Life Cycle Service.

Relationship of somLifeCycle to CosLifeCycle Module

The CosLifeCycle module represents the definition of the Life Cycle Object Service as specified by OMG. The interfaces, exceptions, types and methods defined in the module match those specified by OMG. The interfaces and the methods defined by OMG are as follows:

- **CosLifeCycle::GenericFactory**
  -create_object
  -supports
- **CosLifeCycle::FactoryFinder**
  -find_factories
- **CosLifeCycle::LifeCycleObject**
  -remove
  -copy
  -move

The **CosLifeCycle** module introduces the interfaces and methods, but does not provide any implementation. The interfaces introduced by the **CosLifeCycle** module are inherited by interfaces in the **somLifeCycle** module, which extends those interfaces with additional operations and also introduces several new interfaces. The **somLifeCycle** module provides implementation for most of the operations defined in the **CosLifeCycle** and **somLifeCycle** modules. The only **CosLifeCycle** methods that are not implemented in this release of the Life Cycle Service are **copy** and **move** from the **CosLifeCycle::LifeCycleObject** interface. Please review the omglc.idl file for details on the IDL definitions of the **CosLifeCycle** interfaces.

All interfaces and operations of **CosLifeCycle** and **somLifeCycle** are described as if they were part of **somLifeCycle**, regardless of which module they were defined.

Relationships among somLifeCycle Interfaces

To fully understand these interfaces, it is important to understand the relationships among them. In understanding the relationships, you need to understand the principal purpose for life cycle interfaces, which is to create, remove, move, and copy objects. All the support for removing, moving, and copying is encapsulated within the **LifeCycleObject** interface. Therefore, all the other interfaces within Life Cycle are there essentially to support object creation. The **Location** and **ServerSetLocation** interfaces also offer the capabilities to control where the objects are created. This is very useful for environment customization.

Figure 3-8 on page 3-19 shows the relationships among several of the Life Cycle Service interfaces. The **GenericFactory** may have a reference to a **FactoryFinder**, which may contain a **Location**, any number of **ConstraintBuilder** objects, and any number of **FactoryFilter** objects.

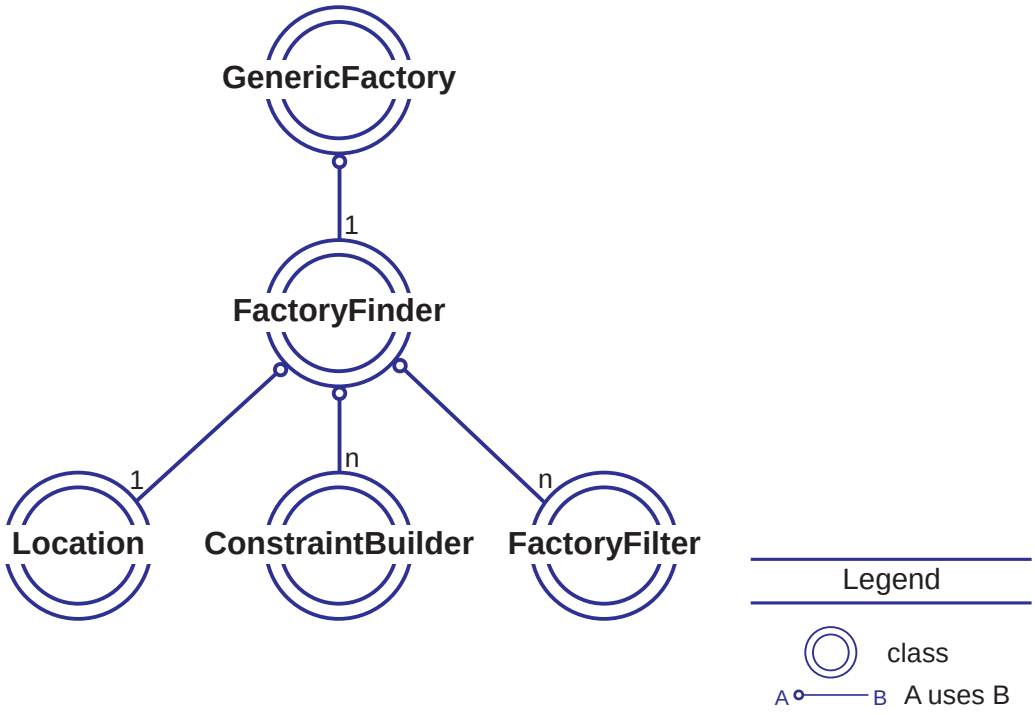

Figure 3-8. Life Cycle Service Interface Relationships

It is possible to have a **GenericFactory** that does not have a **FactoryFinder** registered with it since the **set_factory_finder** method will accept a NULL reference, but the **create_object** and **supports** methods require that one be registered to search for factories. A **GenericFactory** may have at most one (1) **FactoryFinder** registered with it, but a **FactoryFinder** may be registered with any number of **GenericFactory** objects.

It is also possible to have a **FactoryFinder** that does not have a **Location** registered with it. In this case, the location scope for operations on the **FactoryFinder** will consist of all servers in the environment since no specific location is indicated. A **FactoryFinder** may have at most one (1) **Location** registered with it, but a **Location** may be registered with any number of **FactoryFinder** objects.

Any number of **ConstraintBuilder** and **FactoryFilter** objects may be registered with a given **FactoryFinder**. When **ConstraintBuilder** or **FactoryFilter** objects are registered with a **FactoryFinder**, they are assigned a specific name for identification purposes. The only restriction is that a specific **ConstraintBuilder** or **FactoryFilter** may only be registered once under the assigned name per **FactoryFinder** object. The same **ConstraintBuilder** or **FactoryFilter** may be registered numerous times with a **FactoryFinder** as long as a unique identifying name is used each time. This feature is very helpful in the case where you have a **ConstraintBuilder** or **FactoryFilter** that behaves differently based on parameters specified in the **build_constraint** or **filter_factories** methods, and you would like to be able to reuse a single object and simply modify the parameter value (perhaps in the **CosLifeCycle::Key** parameter on several of the **FactoryFinder** and **GenericFactory** methods) rather than creating several different subclasses. The **ConstraintBuilder** and **FactoryFilter** objects may be registered with any number of **FactoryFinder** objects.

Life Cycle Relationship with the 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, which includes the Life Cycle Object Service. The server is a specialization of the DSOM framework that supports the specific needs of the SOMobjects object services. The majority of this responsibility is transparent to client applications or class programmers. However, since several of the Life Cycle Service interfaces are dependent on the Object Services Server for persistent mechanisms, it is important to understand the relationship and subsequent requirements.

Persistent Life Cycle Service Interfaces

The **GenericFactory**, **FactoryFinder** and **ServerSetLocation** all need to maintain a persistent state. As a result, instances of those interfaces must be created and reside on an Object Services Server. Additional initialization and uninitialization requirements are introduced to maintain the persistent references. As a result, the creation and destruction of these objects may differ from the non-persistent Life Cycle Service objects that are not required to reside on the Object Services Server. The non-persistent objects may also be created and Life Cycle reside on an Object Services Server, but are not required to do so.

Object Creation

Due to the special initialization requirements, the creation of instances of the Life Cycle Service interfaces mentioned above is slightly more complex. See “Creating ServerSetLocation Objects” on page 3-38, “Creating FactoryFinder Objects” on page 3-40 and “Creating GenericFactory Objects” on page 3-42 for more details on creating instances of the persistent interfaces.

Object Destruction

All of the Life Cycle Service interfaces inherit from the **LifeCycleObject** interface. As a result, it is a fairly trivial matter to destroy the Life Cycle Service object by simply invoking the **remove** method on the object.

See Chapter 5, “Object Services Server” on page 5-1 for additional information on the Object Services Server.

Life Cycle Relationship with the DSOM Factory Service

The DSOM Factory Service provides a mechanism for locating factory objects based on properties that are bound into a NamingContext. Factories are typically registered with the DSOM Factory Service through the use of the **regimpl** utility provided by DSOM but can also be registered into the name space through use of the Naming Service.

The somLifeCycle support uses the DSOM Factory Service as its mechanism for locating specific factories. It is the underlying mechanism used by the **FactoryFinder** interface. This section describes what a factory is, how these factories are registered with the Naming Server, identifies Life Cycle Service methods that interact with the DSOM Factory Service and differences between the two services.

What is a Factory?

A factory is any object that can be used to create and return a reference to a new object. There are many different objects that fit this definition. This section identifies those that are relevant to the discussion of somLifeCycle services.

Factories as Defined by DSOM Factory Services

All factory objects are bound (explicitly or implicitly) into the DSOM Factory Service. These factory objects are obtained through the **resolve** or **find_any** methods of the DSOM Factory Service. There are basically three different kinds of factories:

1. *SOMClass objects*

   When a binding in the DSOM Factory Service contains a NULL object, and no 'factory=' modifier has been specified in the IDL for the interface of the object for which a factory is desired, the Factory Service returns a reference to a **SOMClass** object. The DSOM Factory Service obtains the **SOMClass** object through a request to the **SOMDServer** object, which dynamically creates it if needed.

2. *objects specified with the 'factory=' modifier*

   When a binding in the DSOM Factory Service contains a NULL object, and the 'factory=' modifier has been specified in the IDL for the interface of the object for which the factory is desired, the Factory Service returns a reference to an instance of that type. The DSOM Factory Service obtains the factory object through a request to the **SOMDServer** object which dynamically creates it. The class specified in the 'factory=' modifier can be of any class and does not have to support any particular interface (i.e. DSOM Factory Service does not provide a base class for factory objects).

3. *objects bound into the DSOM Factory Service*

   When a binding in the DSOM Factory Service contains an object rather than a NULL reference, the Factory Service returns that object. This only occurs when someone specifically binds an object into the DSOM Factory Service and can not occur through the **regimpl** interface.

Factories as Defined by somLifeCycle Services

All factories that are a part of the somLifeCycle service support the **CosLifeCycle::GenericFactory** interface. The **GenericFactoryAbstraction** class is intended to be the base class for all factories defined by the Life Cycle Service or provided through user defined subclasses.

At the present time, somLifeCycle only provides one concrete implementation of a factory, that being the **GenericFactory** class. The **GenericFactory** provides the **create_object** method as a means for object instantiation.

somLifeCycle Factories and Their Relationship to Binding in the DSOM Factory Service

Some factories may be bound into the DSOM Factory Service, and others should not be. There is no specific code in the DSOM Factory Service that enforces the guidelines provided here. However, the guidelines should be followed for the Life Cycle Service and DSOM Factory Service to interact with each other correctly and to provide clients with reasonable and expected results. The following list describes

characteristics of factory interfaces, how clients might be expected to use them and their relationship to factories in terms of the DSOM Factory Service.

- *GenericFactory*

  This is the concrete factory class provided by the Life Cycle Service. It is used by clients to create objects of various interfaces in various servers. Through the associated **FactoryFinder** object, the servers available to the **GenericFactory** object are defined. **GenericFactory** objects are persistent. The most likely scenario for the use of a **GenericFactory** is that it was created and configured as an administrative task and serves the client as the factory for objects to be created within a particular domain and/or under a certain set of policies. The **GenericFactory** is most likely registered with a well known name in the Naming Service, allowing the client to get a reference to it through the Naming Service. For an example of getting a GenericFactory object reference out of the Naming Server, see “Setting Up and Using Life Cycle with a Simple Example” on page 3-1.

- *Non-Persistent Subclasses of GenericFactoryAbstraction*

  These would be user provided factory classes that do not maintain persistent state. These factories normally provide some specialized creation function. If bound into the DSOM Factory Service, these factories would be bound implicitly through use of the 'factory='modifier in IDL. They can be returned by direct calls to the DSOM Factory Service and **FactoryFinder** objects. If not bound into the DSOM Factory Service, the most likely client scenario would be that these factories be created as needed and destroyed when no longer needed by that client.

- *Persistent Subclasses of GenericFactory or GenericFactoryAbstraction*

  These again would be user provided subclasses that provide some specialized creation function. If bound into the DSOM Factory Service, a particular instance would have to be explicitly bound. They can be returned by calls to the DSOM Factory Service and **FactoryFinder** objects. If not bound into the Factory Service, the most likely client scenario would be that these factories be registered with the Naming Service under some well known name so that they can be accessed for use by clients when needed.

Usage of the Term Factory Within this Document

Both factory and generic factory are used quite often within this document. In the absence of further qualification, use of the term 'factory' will generally mean any factory that can be returned from the DSOM Factory Service. The term '**GenericFactory**' will refer to a **somLifeCycle::GenericFactory**.

Registering Items with the DSOM Factory Service

There are several options available that will register a factory with the DSOM Factory Service. The **regimpl** tool is provided that allows the user to identify servers and add classes to those servers. The **regimpl** tool is menu-driven. All classes added through **regimpl** are automatically registered with the DSOM Factory Service.

The recommended way to programmatically register factories in the DSOM Factory Service is through the ImplRepository API using **add_class_to_impldef**.

The ImplRepository API (also called the programmatic interface to the Implementation Repository) is described in “Programmatic Interface to the Implementation

Repository" in "Distributed SOM" in *OS/390 SOMobjects Programmer's Guide*. "The regimpl Registration Utility" in *OS/390 SOMobjects Configuration and Administration Guide* offers a more detailed explanation of how to use **regimpl**.

Life Cycle Service methods that interact with the DSOM Factory Service

The following methods interact directly or indirectly with the DSOM Factory Service:

- **find_factory**
- **find_factories**
- **create_object**
- **supports**

Differences between Life Cycle and the DSOM Factory Service

There are several options available for locating factories and using the factories to create objects. The user may choose to interact directly with the DSOM Factory Service to locate objects and invoke creation requests, or use the Life Cycle Service to locate the factories and make the creation requests for the user. The DSOM Factory Service and the Life Cycle Service both offer certain levels of local/remote transparency, so the user need not be concerned with the location specifics of the object. Due to the fact that some of the Life Cycle Service objects must reside on an Object Services Server, some restrictions exist when creating local objects in pure client processes. The Life Cycle Service will not be able to retrieve local **SOMClass** factories since a **FactoryFinder** must reside on an Object Services Server which is remote to the client process. The DSOM Factory Service is able to avoid this restriction since there is no relationship with the Object Services Server. The DSOM Factory Service requires less configuration and administration, but Life Cycle offers a layer of abstraction that allows the system administrator to control the development environment. The Life Cycle Service provides several methods that will interact with the DSOM Factory Service for the user, also offering additional policy control, filtering, creation and initialization capabilities that are not available if the user interacts directly with the DSOM Factory Service. The Life Cycle Service also provides OMG compliance for users that are interested in adhering to OMG standards.

The following discussion walks through similar scenarios for each, and may be helpful to identify the relationship between Life Cycle and the DSOM Factory Service.

The DSOM Factory Service offers the capability of locating a set of factories that are able to create an object of a desired class. The user may build an **ExtendedNaming::ExtendedNamingContext::Constraint** specifying which factories to find, and invoke the DSOM Factory Service **find_all** or **find_any** methods. If the **find_all** method is used, the user will then need to invoke the **resolve** method on each binding to retrieve references to the factory objects. The user is then responsible for identifying the type of each factory (**SOMClass**, a **GenericFactory**, or a user-defined factory) and invoking the correct creation and initialization requests for that particular factory object. The DSOM Factory Service requires very little environment configuration and administration.

The Life Cycle Service is also able to locate factories and create objects for the user. If the user is interested in obtaining a reference to a factory (or factories) the **find_factory** (or **find_factories**) method may be invoked on a **FactoryFinder**. The **FactoryFinder** will receive a **CosLifeCycle::Key** parameter that identifies the type

of factory to find, additional constraint information to be considered, and filtering mechanisms. The **FactoryFinder** will build the **ExtendedNaming::ExtendedNamingContext::Constraint** for the user based on information in the **CosLifeCycle::Key**, and will interact with the DSOM Factory Service for the user. The **FactoryFinder** will resolve the bindings returned from the Factory Service if necessary and perform further filtering on the factories located based on additional filtering information specified in the **CosLifeCycle::Key**.

If the user is interested in creating an object, the Life Cycle Service makes the operation simple with the **create_object** method defined by the **GenericFactory**. The **create_object** method will first interact with a **FactoryFinder** to locate factories that are capable of creating the desired object, then will traverse the sequence of factories found, identify the types of factories, and optimally invoke creation requests based on the type of factory.

The Life Cycle Service requires more initial configuration, but is easy to maintain once established. It provides more policy control and simplifies the object creation process.

Building blocks of the Life Cycle Service

To use the Life Cycle Service to its full potential, it is important to understand the basic building blocks necessary for object creation. In several of the methods, a **CosLifeCycle::Key** or **CosLifeCycle::Criteria** are passed as parameters. These structures provide valuable information to the methods for factory searches and object creation and initialization. Once the user understands these complex base structures and how they are used by the Life Cycle Service, the user will be able to use all of the special options available that allow for environment control and efficient application development.

This section will describe the components of the **CosLifeCycle::Key** and **CosLifeCycle::Criteria**, describe how an **ExtendedNaming::ExtendedNamingContext::Constraint** is constructed for use in factory searches, and provides an example illustrating how factories are found using the **find_factories** or **find_factory** methods. It also describes exception monitoring and error logging capabilities.

Building the CosLifeCycle::Key Parameter

The key parameter is a sequence of structures that each contain two strings, a "kind" string and an "id" string. These structures identify the properties to use when searching for a factory to perform the desired function. Table 3-1 on page 3-25 illustrates acceptable key values.

Table 3-1. Key Definitions

"kind" string	**"id" string**
object interface	A fully qualified IDL interface name. Factories returned must support the creation of objects that support this interface as their principal interface. The interface should be specified in standard CORBA IDL. If the interface is part of a module, colon-colon (::) should be used to delimit the module and interface name.
factory interface	A fully qualified IDL interface name. Factories returned must support this interface as their principal interface. The interface should be specified in standard CORBA IDL. If the interface is part of a module, colon-colon (::) should be used to delimit the module and interface name.
constraint	An ExtendedNaming::ExtendedNamingContext::Constraint that is included in the constraint passed to the DSOM Factory Service for the **find_all** method. The constraint will be included in the following manner: ((other constraint items) and (constraint)).
cb:<nameOfTarget **ConstraintBuilder** >	A string in a syntax defined by the **ConstraintBuilder**. The **build_constraint** method is called with this string as the input argument, and the result is included in the constraint passed to **find_all** in the following manner: ((other constraint items) and (constraint returned by the **ConstraintBuilder**)).
ff:<nameOfTarget**FactoryFilter**>	A string in a syntax defined by the **FactoryFilter**. The filter_factories method is invoked with this string as an input parameter to further reduce the size of the sequence of factories returned from a factory search.

The only required item in the **CosLifeCycle::Key** parameter is object interface. The others are optional.

The **FactoryFinder** uses object interface, constraint, and any **ConstraintBuilder** objects associated with it to prepare an **ExtendedNaming::ExtendedNamingContext::Constraint** to pass to the DSOM Factory Service. Those three items are in effect the search arguments used to find factories that can create the desired object as specified in the *key*.

After the DSOM Factory Service locates all factory object references that match the requirements, the **FactoryFinder** uses the factory interface portion of the **CosLifeCycle::Key** and **FactoryFilter** objects associated with the **FactoryFinder** to further reduce the factories.

The following code snippet illustrates how to build the **CosLifeCycle::Key** parameter. The Life Cycle Service provides a file called 'somlcdef.h' that contains several constant definitions that will be helpful in this process and are used in the following example. This example makes the assumptions that the cb: and ff: portions of the following **CosLifeCycle::Key** reference **ConstraintBuilder** and **FactoryFilter** subclasses that have been defined and implemented, and have been created and registered with the **FactoryFinder**.

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

```
/* include the necessary header files for LifeCycle */
#include <somlcdef.h> /* defines helpful constants */
#include <somlc.xh>
     /* the LifeCycle Service header */
```

The next 2 lines define the actual **CosLifeCycle::Key** structure. the Key is a sequence of **CosNaming::NameComponent** elements. Each **CosNaming::NameComponent** consists of a 'kind' string and an 'id' string. In this case, the size of the buffer is set to contain 5 elements. (The C++ array index starts at 0.)

```
CosNaming_NameComponent
    elements[4];
CosLifeCycle_Key
    myKey = {5,0,elements};
```

The following portion builds the **CosLifeCycle::Key** by assigning 'kind' and 'id' values for each of the elements in the sequence buffer. The following illustrates the object interface element of the Key. This indicates to locate all factories that can create Movie objects.

```
myKey._buffer[0].kind = KIND_OBJ_INF;
myKey._buffer[0].id = "MovieRental::Movie";
```

The following illustrates the factory interface element of the Key. It will be used after the factories are retrieved by a factory search to eliminate any factories that do not support the MovieRental::MovieFactory interface.

```
myKey._buffer[1].kind = KIND_FAC_INF;
myKey._buffer[1].id = "MovieRental::MovieFactory";
```

The following illustrates the constraint element of the Key. It indicates not to retrieve factories located on that specific server.

```
myKey._buffer[2].kind = KIND_CONSTRAINT;
myKey._buffer[2].id = "serverId != '2d74625f-9824cff0-80-04- \n
100051234560'";
```

The following illustrates the cb: element of the Key. The name of a **ConstraintBuilder** registered with the **FactoryFinder** is **OperSysBuilder**, and the value we want to have passed in on invocations of the **build_constraint** method is OS/390. This will ensure that the DSOM Factory Service only retrieve factories that use the specified operating system.

Note: This assumes that a **ConstraintBuilder** subclass is provided by the user that is able to build a meaningful constraint based on the parameter value provided.

See "Subclassing the ConstraintBuilder Interface" on page 3-43 for the logic of the subclass implementation.

```
myKey._buffer[3].kind = "cb:OperSysBuilder";
myKey._buffer[3].id = "OS/390";
```

Five elements have been added to the **CosLifeCycle::Key** structure, so the size should be incremented to match.

```
myKey._length = 5;
```

The elements in the **CosLifeCycle::Key** may be specified in any order. The Life Cycle Service implementation is designed to traverse the **CosLifeCycle::Key** and extract all recognized elements. The user is free to add elements other than the five outlined above, but they will not be recognized by the Life Cycle Service implementation.

It may be useful for the administrator to create a file consisting of constants that may be used often in the **CosLifeCycle::Key** structure. All of the string values specified for the 'id' portions in the example above could easily be defined once in a common utility file and reused by developers throughout the environment.

How Constraint Expressions Passed to the DSOM Factory Service by a FactoryFinder are Built

The **ExtendedNaming::ExtendedNamingContext::Constraint** passed to the **find_all** operation on the DSOM Factory Services consists of several constraint strings joined by conditional **and** and **or** operators. In this example lets assume that two properties have been added to the Naming Server; one called OpSys that is able to identify the Operating System of a particular server, and another named Shared, which indicates if the server is part of a pool of resources or independent. See Chapter 4, "Naming Service" on page 4-1 for more details on how to create additional properties.

A **ConstraintBuilder** named OperSysBuilder is registered with the **FactoryFinder** and is able to generate a constraint indicating which operating system is desirable. A **ConstraintBuilder** named *SharedResource* is also registered with the **FactoryFinder** and will build a default constraint indicating that factories that have a value of TRUE for the Shared property should be located. The **ConstraintBuilder** objects referenced are really subclasses of the Life Cycle Service

ConstraintBuilder interface, since the default interface provided is abstract. The **FactoryFinder** does have a **Location** registered with it that consists of the three servers available to the Movie Rental Application Development department. The **FactoryFinder** resides on a server that is included in the **Location**, and the *factory_key* consists of the following information:

- object interface = "MovieRental::Movie"
- -The desired class of object to create is a MovieRental::Movie.
- constraint = "serverId != '2f4e51f3-1f31f3eda80-7f-00-10005ab1b55c'"
- -This server is very heavily used, so the developer would like to avoid it if possible.
- cb:<OperSysBuilder> = "OpSys == 'OS/390'"
- -The developer requires that factories found must support the OS/2 operating system.

The *factory_key* specified by the developer doesn't specify a value to use for the SharedResource **ConstraintBuilder**, so the default value of TRUE will be used.The steps the **FactoryFinder** will take to build the **ExtendedNaming::ExtendedNamingContext::Constraint** used in the factory search are as follows:

1. All ServerIds are retrieved from the **Location** object registered with the **FactoryFinder** and put into an **or**ed expression. In addition, if the ServerId of the process the **FactoryFinder** is running in exists in the **Location**, the keyword '_LOCAL' concatenated with the hostname is added. The hostname is retrieved from an environment variable specification if one exists, or the somd stanza of the gosenv.ini file. If no **Location** object is associated with the **FactoryFinder**, nothing is built for server selection into the constraint. After this first step is completed, the constraint will look something like the following:

   ```
   ((serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1b24a' ) or
    (serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1b55c' ) or
    (serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1c12d' ) or
    (serverId == '_LOCALhostname'))
   ```

2. The object interface as specified in the key is built into the constraint and is **and**ed with what has already been built. After this step, the constraint will change to look like the following:

   ```
   ((serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1b24a' ) or
    (serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1b55c' ) or
    (serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1c12d' ) or
    (serverId == '_LOCALhostname')and
    (class == 'MovieRental::Movie'))
   ```

3. The value in the "constraint" portion of the key is added, using the **and** condition. The constraint will now look like the following:

   ```
   ((serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1b24a' ) or
    (serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1b55c' ) or
    (serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1c12d' ) or
    (serverId == '_LOCALhostname') and
    (class == 'MovieRental::Movie') and
    (serverId != '2f4e51f3-1f3eda80-7f-00-10005ab1b55c'))
   ```

4. The **build_constraint** method is invoked for each **ConstraintBuilder** registered with the **FactoryFinder**. The *key_kind* specified in the *factory_key* for the OperSysBuilder will be passed to the **build_constraint** method instead of the default value. The value returned from the invocation of **build_constraint** for

the registered **ConstraintBuilder** objects is added using the **and** condition. After invoking **build_constraint** for the two registered **ConstraintBuilders**, the constraint will look like the following:

```
((serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1b24a' ) or
 (serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1b55c' ) or
 (serverId == '2f4e51f3-1f3eda80-7f-00-10005ab1c12d' ) or
 (serverId == '_LOCALhostname') and
 (class == 'MovieRental::Movie') and
 (serverId != '2f4e51f3-1f3eda80-7f-00-10005ab1b55c') and
 (OpSys == 'OS/390' ) and
 (Shared == 'TRUE'))
```

The resulting constraint, assuming all the above, is passed by the **FactoryFinder** to the DSOM Factory Service's **find_all** method, and all factories identified that meet the requirements will be returned.

In the simplest case, if the **FactoryFinder** has no **Location** object or registered **ConstraintBuilder**s and if a constraint is not specified in the *factory_key*, the constraint string built would look like this:

```
(class == 'MovieRental::Movie')
```

See Chapter 4, “Naming Service” on page 4-1 for more information on the constraint syntax, and the DSOM Factory Service discussion for additional information on the process involved in evaluating the constraint to locate factories.

Filtering Capabilities of the Life Cycle Service

Interaction with the DSOM Factory Service, either directly or through the Life Cycle Service, allows the user to retrieve a set of factories that meet the static constraints specified for the search. The Life Cycle Services takes this a step further and provides two mechanisms that allow the user to dynamically modify the number of factories that will be returned to the caller. After the **find_factories** method has been invoked on a **FactoryFinder** and a set of factories has been returned, the Life Cycle service will filter the set of factories further based on additional policies specified in the factory interface portion of the **CosLifeCycle::Key** or any **FactoryFilter** objects registered with the **FactoryFinder**.

The factory interface

The factory interface portion of the **CosLifeCycle::Key** offers the user the opportunity to specify the name of a particular interface that is preferred over all other types of factories. The **FactoryFinder** will traverse the sequence of factories and remove any instances that do not support the interface specified in the **CosLifeCycle::Key**. In our movie rental scenario, the user may want to specify the MovieRental::MovieFactory interface as the factory interface in the **CosLifeCycle::Key**. As a result, only factories that are MovieRental::MovieFactory references will be returned. Table 3-1 on page 3-25 provides more information on the factory interface and how to specify one.

FactoryFilter objects

The Life Cycle Service provides the **FactoryFilter** interface which may be used to further eliminate factories based on some dynamic policies the user would like to enforce. The **FactoryFinder** will first interact with the DSOM Factory Service to retrieve a set of factories that match the **CosLifeCycle::Key**. The **FactoryFinder** will then eliminate any factories that do not support the factory interface specified in the **CosLifeCycle::Key**, if one was specified. The **filter_factories** method on all **FactoryFilter** objects that are registered with the **FactoryFinder** will then be invoked to further reduce the number of factories returned to the caller. Any number of **FactoryFilter** objects may be registered with a **FactoryFinder** and any number of dynamic policies may be enforced.

See “Subclassing the FactoryFilter Interface” on page 3-43 for more information on subclassing the **FactoryFilter** interface and building dynamic policies.

How Factories Are Found

The process of finding factories can become complex, depending on the items specified in the *factory_key* passed in on the **find_factories** request and the number of **ConstraintBuilders** and **FactoryFilters** registered with a **FactoryFinder**. Figure 3-9 on page 3-31 illustrates the process flow involved when the client wants to locate a factory that it can use to create a MovieRental::CustomerAccount object.

In the example, the client sends a message to the **FactoryFinder**, requesting any factories that are able to create a MovieRental::CustomerAccount object. The client sends this message via the **find_factories** method. If the client wants only one object reference returned and does not care which one it receives, the client could use the **find_factory** method.

The **FactoryFinder** starts by building a constraint string that is used to interface with DSOM Factory Service. (The first four boxes of the **FactoryFinder** processing illustrates that various pieces are pulled together to build the constraint used on factory searches.) The constraint string consists of the object interface portion of the *factory_key* parameter along with the constraint portion of the *factory_key* if specified and the constraints built by **build_constraint** method of the registered **ConstraintBuilder** objects. See “How Constraint Expressions Passed to the DSOM Factory Service by a FactoryFinder are Built” on page 3-27 for specific details on what the constraint string will look like as the various pieces are combined to form the final constraint string. It may also be helpful to see “Building the CosLifeCycle::Key Parameter” on page 3-24 for more information on the **CosLifeCycle::Key** parameter.

After a valid constraint string is built, the **FactoryFinder** invokes the DSOM Factory Service **find_all** method. DSOM Factory Service searches through all registered factories and returns bindings to all that match the specified constraint string.

The **find_factories** method then invokes the DSOM Factory Service **resolve** method on each of the bindings returned. DSOM will instantiate any factory objects that are not already in existence and returns the object reference to the **FactoryFinder**. The **FactoryFinder** stores all of the factory object references that matched in a sequence. When all the bindings are resolved, each factory is compared with the factory interface that was passed in the key. All factories that do not support the specified interface are removed from the sequence.

Finally, the sequence is filtered further through the execution of the **filter_factories**method on all the **FactoryFilter** objects associated with this **FactoryFinder**. The **filter_factories** method examines every object reference and eliminates any that do not meet the requirements of the **FactoryFilter**. The potential result of the operation is a reduced sequence of object references.

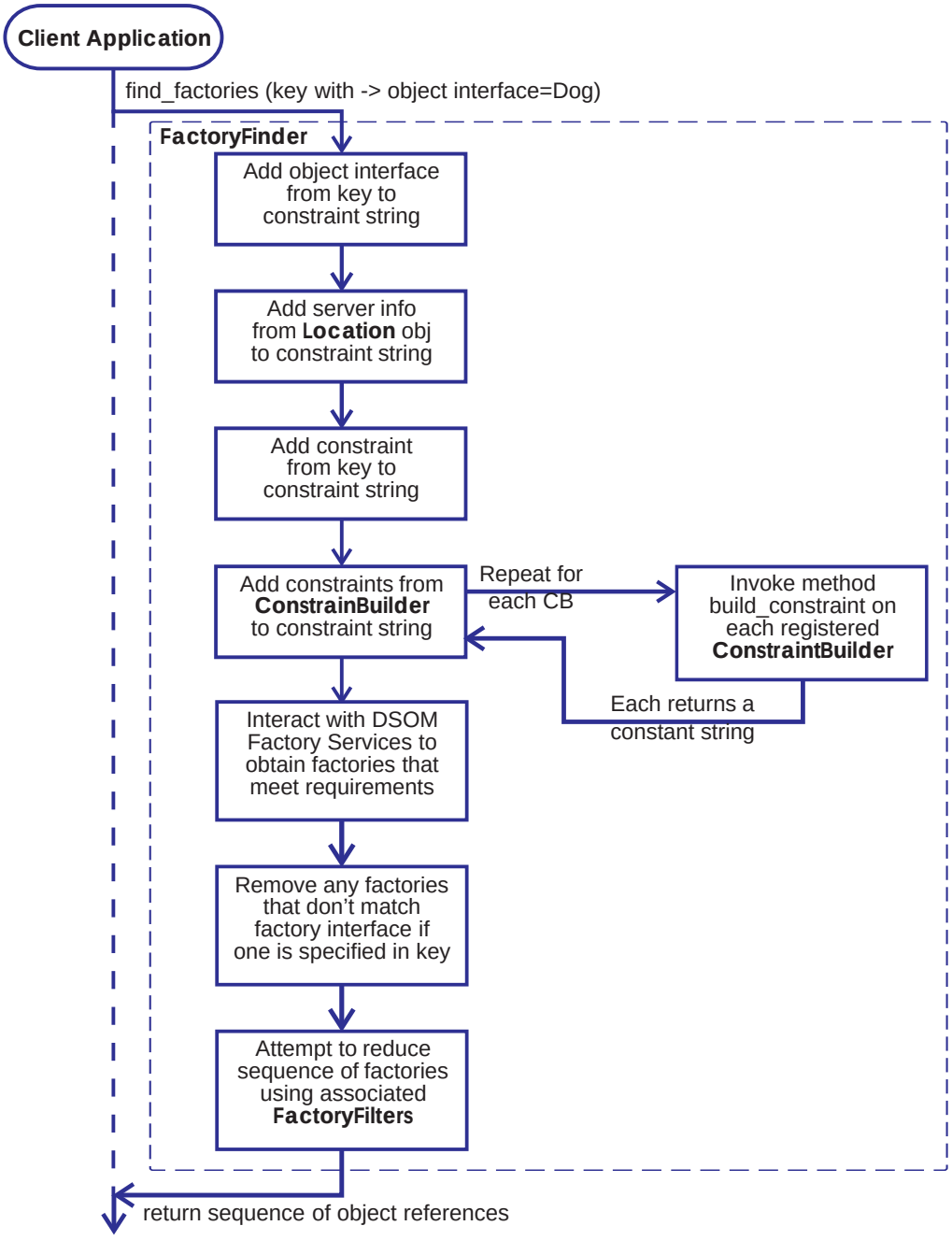

Figure 3-9. Process flow for find_factories method of FactoryFinder

The sequence of object references that are capable of creating instances of the MovieRental::CustomerAccount class is returned to the client from the **find_factories** method. If the sequence is not empty, the client evaluates whether a factory exists that the client knows how to interface with. If a factory exists, the client is responsible for knowing the interface of the factory to invoke the method that allows the object creation. For example, if the object reference is to a class

object, the client can then invoke methods such as **new**, **somNew** or **somNewNoInit**. If the factory object is a user- defined factory, such as a MovieRental::CustAcctFactory with a create_cust_account method, the client must understand the interface and know to invoke the create_cust_acct method passing the required method parameters.

Once the client has an object reference to a factory, the client can submit multiple create requests in succession without having to invoke the **find_factories** method each time. If the client has no need to create multiple instances of an object, it may be beneficial to use the **create_object** method on a **GenericFactory** to create the object instead. The **create_object** method is able to evaluate the factories and make creation requests for the user.

Building the CosLifeCycle::Criteria Parameter

The **CosLifeCycle::Criteria** is in essence a sequence of name/value pairs, where each name/value pair is composed of a string in the name field and a CORBA 'any' type in the value field. Table 3-2 provides additional information on criteria.

Table 3-2. Criteria parameter

Name	Value	Interpretation
SOM initialization	TRUE or FALSE	a boolean set to TRUE to create the object using somNew, FALSE to create with **somNewNoinit**. If not specified, the default is TRUE.
<anyName>	Anything this subclass understands	The value can be anything that this particular subclass of GenericFactory understands. It is likely to contain initialization information to be used within **initialize_object** to either invoke initializers or other methods.

The **CosLifeCycle::Criteria** parameter is optional for this release of the Life Cycle Service. In other words, it is possible to simply pass in a NULL reference in it's place. The **GenericFactory** and **GenericFactoryAbstraction** interfaces provided are able to use the default value of TRUE for the SOM initialization name, so the only situations that would warrant a specific **CosLifeCycle::Criteria** structure would be if the user wanted to specify a value other than the default, or specify other name/value pairs that have meaning to a subclass.

Overriding the Default for SOM initialization

The default value for the SOM initialization name is TRUE, which means that the **GenericFactory** will use the default initializers for object creation. If the user would like to initialize the object in a manner other than via default initializers (i.e. possibly with specialized initializers), the value of FALSE should be associated with the SOM initialization name. The **GenericFactory** will not invoke the default initialization and the user will be able to invoke different initialization mechanisms.

The following code example illustrates how to change the value of the SOM initialization portion of the **CosLifeCycle::Criteria**. The Life Cycle Service provides a

file called **somlcdef.h** that contains several constant definitions that will be helpful in this process and are used in this example.

```
/* include the necessary header files for LifeCycle */
#include <somlcdef.h> /* defines helpful constants */
#include <somlc.xh>
     /* the LifeCycle Service header */
```

The next 2 lines define the actual **CosLifeCycle::Criteria** structure. the Criteria is a sequence of **CosLifeCycle::NameValuePair** elements. Each **CosLifeCycle::NameValuePair** consists of a **CosNaming::Istring** 'name' field and an 'any' type 'value' structure, which consists of a _type and _value. In this case, the size of the buffer is set to contain 3 elements. (The C++ array index starts at 0.)

```
CosLifeCycle_NameValuePair
    nvPair[2];
CosLifeCycle_Criteria
    criteria1 = {3,0,nvPair};
boolean
    InitVal = FALSE;
```

The following portion builds the **CosLifeCycle::Criteria** by assigning values for each of the elements in the sequence buffer.

```
criteria1._buffer[0].name = SOM_INIT;
/* indicates we are working with the SOM initialization 'name' of \n
the
   NameValuePair */
criteria1._buffer[0].value._type = TypeCodeNew(tk_boolean);
/* this defines that the 'value' of the NameValuePair is of type
   boolean. Please refer to
OS/390 SOMobjects Programmer's Guide.
   for more information on TypeCode definitions. */
criteria1._buffer[0].value._value = &Init1;
/* a _value portion of the 'value' of the NameValuePair is really \n
a
   pointer to the value. As a result, we pass in a pointer to the
   boolean value FALSE we defined above */
criteria1._length = criteria1._length + 1;
/* since we have added an element to the Criteria, we need to \n
increment
   the length to match */
```

Including User-Defined Criteria Requirements

In many cases it may be desirable for the user to use the **CosLifeCycle::Criteria** parameter as a means for specifying other creation or initialization requirements. The Life Cycle Service only provides the SOM initialization option, but the user is free to provide other options and requirements that are meaningful to user-defined factories and subclasses.

For example, let's assume we are developing a movie rental application, and every time the Movie Rental Company purchases a movie for the rental business, a new

MovieRental::Movie object is created that stores information pertinent to the purchase. A MovieRental::MovieFactory interface has been defined that is responsible for creating instances of MovieRental::Movie objects. The MovieFactory has a method called 'make_movie' that takes in a string parameter indicating the name of the movie purchased. The following steps must be performed:

1. The title of the movie must be stored in the Movie object upon creation
2. The movie must be logged in the MovieRental::MovieLog, which lists all of the movies owned by the Movie Rental Company
3. The name of the purchaser must be stored in the Movie object

It may be desirable to specify additional portions in the **CosLifeCycle::Criteria** passed in that would contain the title of the movie, a reference to the MovieLog, and the name of the buyer. This information could be extracted from the **CosLifeCycle::Criteria** and applied in the creation process. The **CosLifeCycle::Criteria** parameter would be built in the same manner as illustrated in the example in “Overriding the Default for SOM initialization” on page 3-32.

Of course other methods will need to be overridden, such as **real_object_creation** or **initialize_object**, that will be able to recognize and process the additional user-defined information in the **CosLifeCycle::Criteria**.

How Objects Are Created using the create_object Method

The **GenericFactory** uses the services provided by the **FactoryFinder** interface to locate factories to use for object creation. Because of the interaction of the **create_object** method with **FactoryFinder** objects (which also interact with DSOM Factory Service), the use of the **real_object_creation** and **initialize_object** methods, and the various kinds of factories that can be interacted with, the processing done by the **create_object** method can be complex. At a high level, the logic used by this method is that it tries to use all the factories that meet the constraints for factory selection until an object is successfully created. Because all object creation in SOM eventually requires interaction with a **SOMClass** object, priority is given to **SOMClass** objects in the list of potential factories. In addition, a local factory is given priority over a remote factory.

Process Flow for create_object

Without regard to how error conditions are handled, the following is a high-level outline of the logic used in the **create_object** method.

```
Get list of potential factories from factory finder
Do while (an object has not been created
     and there are more factories left to try)
     Select a factory that hasn't been tried, using these priorities:
     1. a SOMClass object that is local to the FactoryFinder
     2. a SOMClass object that is remote to the FactoryFinder
     3. an object that doesn't support
     CosLifeCycle::GenericFactory that is local
     4. an object that doesn't support
     CosLifeCycle::GenericFactory that is remote
     5. an object that supports CosLifeCycle::GenericFactory
     that is local
     6. an object that supports CosLifeCycle::GenericFactory
     that is remote
     If the selected factory supports CosLifeCycle::GenericFactory,
     invoke the create_object method on itself, passing the
     same key and criteria passed to the current call
     else
     call real_object_creation on itself
End (Do While)
If an object was successfully created
     call initialize_object
     return object to caller
else
     raise NoFactory exception
```

The **real_object_creation** method is invoked from within the **create_object** method if the factory is a **SOMClass** or a factory that is not a **GenericFactory**, such as a user- defined factory. The real_object_creation method is capable of creating objects in the case of **SOMClass** factories, but will perform no operation in the case of user-defined factories. For instance, in our Movie Rental example, a MovieFactory interface may exist to create and initialize MovieRental::Movie objects using the **make_movie** method. The **make_movie** method requires a parameter that specifies the name of the Movie. Obviously the Life Cycle Service implementation will not recognize the MovieFactory or know how to invoke **make_movie**. Therefore, it would be desirable to subclass the **GenericFactory** and override the **real_object_creation** method to handle specific user- defined factory objects.

Subclassing the GenericFactory Interface

Subclassing the **GenericFactory** and overriding the methods is desirable if the user environment invokes methods on factory objects other than **SOMClass** references for object instantiation and initialization. The **GenericFactory** inherits from the **GenericFactoryAbstraction** interface and uses the **real_object_creation** and **initialize_object** methods provided by the **GenericFactoryAbstraction** during a **create_object** request to create an instance of the desired object and initialize it. As illustrated previously, the **real_object_creation** method is equipped to create objects from a **SOMClass**, but will not be able to recognize user-defined factories. As a result, if user-defined factories exist in the development environment, it will be necessary to override the **real_object_creation** method.

Overriding the real_object_creation method

Following on with the MovieFactory example introduced in the previous section, let's walk through a possible scenario that would identify a useful case for overriding the **real_object_creation** method.

If a user makes a **create_object** request passing in the correct **CosLifeCycle::Key** necessary to locate the MovieFactory, and the MovieFactory has been registered with the DSOM Factory Service, the MovieFactory will be returned as one of the factories found in the search. The **real_object_creation** method would be overridden to provide implementation that recognizes MovieFactory references and understands how to invoke the instantiation request. The logic of the implementation may be as follows:

```
If (the factory passed in is a MovieFactory)
     extract the name of the movie from the name portion of Criteria
     invoke make_movie method on the MovieFactory passing in name
else
     invoke parent real_object_creation method to handle other cases
return object to caller (create_object method)
```

The implementation would, of course, have to monitor for and handle exception cases to provide a valid object or OBJECT_NIL to the caller.

Overriding the initialize_object method

The **create_object** method also makes a call to the **initialize_object** method if an object was successfully created. The **initialize_object** method is designed to perform any additional initialization on the newly created object that was not completed during object creation. This method is abstract and no default implementation is provided. It is also a good method to override and implement to perform special initialization requirements. Since the **initialize_object** method is always called if the **GenericFactory** is able to create an object, this is an ideal place to put instructions that should always be carried out no matter what type of factory is used to create the object. For example, every time a new movie is purchased for the Movie Rental Shop, a new Movie object is created and a unique code number is assigned during the creation process. The **initialize_object** method could perform some automated processes based on the information in the **CosLifeCycle::Criteria** and the object. It is possible to add any information into the **CosLifeCycle::Criteria**, so let's assume the cases identified in the previous "Building the Criteria Parameter" section where the names of the additional sections of the **CosLifeCycle::Criteria** are "log" and "buyer", and the values are an object reference to a MovieRental::MovieLog and a string indicating the name of the purchasing agent, respectively. The **initialize_object** method could extract the log from the **CosLifeCycle::Criteria** and modify it to add the code number from the new movie to the log. It could then initialize the buyer_name attribute of the Movie object with the name of the purchaser specified in the buyer portion of the **CosLifeCycle::Criteria**. The logic for the implementation may be as follows:

```
If (the object passed in is a Movie)
     extract MovieLog reference from the "log" portion of Criteria
     invoke the get_code_number method on the Movie object
     invoke the add_movie method on the MovieLog
     extract identity of buyer from the "purchaser" portion of Criteria
     invoke the add_buyer_name method on the Movie object passing in
     name extracted
return object to caller (create_object method)
```

The subclass capabilities provide a lot of flexibility in the creation and initialization of objects. The **CosLifeCycle::Criteria** is a very powerful structure and can be tailored to meet specific needs. For additional information on the

CosLifeCycle::Criteria parameter, see “Building the CosLifeCycle::Criteria Parameter” on page 3-32.

If the user does choose to subclass the **GenericFactory** interface and overriding the above methods, the user is responsible for verifying parameters and raising any exceptions necessary. Also, since the **GenericFactory** interface inherits from the **LifeCycleObject** interface the user must list the **LifeCycleObject__Proxy** interface as the base proxy class (baseproxyclass modifier) in the implementation section of the IDL.

Exception Monitoring/Error Logging

It is always a good programming practice to monitor for exceptions after all method invocations. The Life Cycle Service has defined several exceptions that may be raised in various situations to help the user identify problem areas. These exceptions are defined in the somlcdef.h file, and may be monitored for by name or by error code.

The Life Cycle Service will also monitor for exceptions raised by underlying methods, and will map any exceptions raised for you. An error log is also provided, which will provide additional information when an exception is raised. The error log facility is described in more detail in “The Error Log Facility” in *OS/390 SOMobjects Messages, Codes, and Diagnosis*.

See Life Cycle Service Error Codes in *OS/390 SOMobjects Messages, Codes, and Diagnosis* for a complete list of the Life Cycle Service exceptions, as well as an explanation of each error and possible steps to take to recover.

Environment Configuration using the Life Cycle Service

Tailoring your development environment to meet your business policies is one of the key features offered by the Life Cycle Service. This section will help you identify items to consider during the configuration process. It will also show how to create Life Cycle Service objects and subclass many of the Life Cycle interfaces to suit your needs.

Getting Started

This section outlines some important environment configuration issues that must be decided upon to successfully use the Life Cycle Service. The decisions will be finalized based on your specific environmental needs.

The Role of an Administrator

The systems administrator in a typical enterprise is responsible for setting up and maintaining the computer environment for all employees to use. For example, the administrator might configure a computer network, define naming conventions, or assign user IDs.

For Life Cycle Service, the system administrator's responsibilities include creating the Life Cycle Service objects for clients to use and tailoring the environment to meet specific policies and requirements. The Life Cycle interfaces have been introduced in previous sections of this chapter, and any of them may be created or subclassed for environment control capabilities. This will require that the administrator has solid programming skills or has programming resources available for use. Since

the administrator will want to provide a development environment that is meaningful and efficient for the programming teams, it is also important that the administrator understand how developers will ideally use the Life Cycle Service. See “Using the Life Cycle Service in Application Development” on page 3-48.

When to Use the Provided Life Cycle Service Objects

The Life Cycle Service has provided several interfaces that may be used to tailor the application development environment. Default Life Cycle Service objects are also created during the configuration of SOM (when **som@cfg** is invoked) that are available for use in the case where it is not beneficial to tailor the application development environment. A **FactoryFinder** object is created and bound into the Naming Service with the name "LifeCycleFactoryFinder." No **Location** object, **ConstraintBuilder** objects or **FactoryFilter** objects are registered with the LifeCycleFactoryFinder. A **GenericFactory** object is also created and bound into the Naming Service with the name "LifeCycleGenericFactory." The LifeCycleFactoryFinder is then registered with the LifeCycleGenericFactory. This will provide the basic Life Cycle Service functionality (**create_object, find_factories, find_factory, supports**) without requiring the creation of additional Life Cycle Service objects.

If your application development environment consists of a small number of servers, simple creation and initialization mechanisms exist, and there is no need for additional filtering capabilities, it may not be useful to use the Life Cycle Service **Location** objects, **ConstraintBuilder** objects and **FactoryFilter** objects to customize your environment. In this case you may just want to use the **GenericFactory** (LifeCycleGenericFactory) and **FactoryFinder** (LifeCycleFactoryFinder) default objects provided by the Life Cycle Service.

A very large and complex environment is an ideal candidate for customization using the Life Cycle Service objects. A complex environment will have specific policies to enforce, whether it be restricting certain departments to specific servers, using specific operating systems or filtering based on other properties or policies. In this case it is very beneficial and efficient to tailor the development environment to meet the policies while offering efficient programming opportunities to application developers. With a well-tailored environment, the application developer need only invoke the **create_object** or **find_factories** methods to instantiate an object or locate a factory. The programmer need not be concerned with the environment policies or restrictions that the administrator needs to enforce.

Subclassing and Creating Life Cycle Service Objects

To tailor the development environment to suit your needs, several Life Cycle Service objects will need to be created so they are available for use. This section will discuss the various Life Cycle Service objects you may want to create and provide examples indicating the method calls necessary to accomplish this.

Creating ServerSetLocation Objects

The **ServerSetLocation** objects must be created and reside on an Object Services Server. This requirement adds a special twist to the creation of a **ServerSetLocation** object since special initialization must be done to maintain the persistent state of the object. See “Life Cycle Relationship with the Object Services Server” on page 3-20 and Chapter 5, “Object Services Server” on page 5-1 of Programmer' s Guide for Object Services for more information on the Object Services Server.

Several options exist for the creation of a **ServerSetLocation** object, depending on where the object is being created (local or remote), if the Life Cycle Service objects are used, or if other methods are invoked that return an instance of **ServerSetLocation**.

Local Object Creation: If the application is running on an Object Services Server, it will be possible to create a **ServerSetLocation** object locally by invoking **new** (for C++) or **somNewNoInit** (for C) followed by **init_for_object_creation**. The following code example illustrates the steps involved in the process.

```
Environment *ev; /* environment to monitor for exceptions */
/* the following variable will store the value returned from the \n
new call. This will be released after the object is initialized \n
via the init_for_object_creation method */
somLifeCycle_ServerSetLocation  *pre_initSSL;
/* this will be our fully initialized object reference */
somLifeCycle_ServerSetLocation  *mySSL;
/* create the ServerSetLocation object */
pre_initSSL = new somLifeCycle_ServerSetLocation;
mySSL = (somLifeCycle_ServerSetLocation *)
    ((void *) pre_initSSL->init_for_object_creation(ev));
/* release the reference returned from the new call since the \n
object reference we'll use is the fully initialized mySSL */
((SOMDObject *)pre_initSSL)->release(ev);
```

Remote Object Creation: If the application is not running on an Object Services Server, it will be necessary to locate an Object Services Server and create the **ServerSetLocation** object remotely by invoking **somdCreate** with FALSE as the initialization parameter followed by **init_for_object_creation**. The following code example illustrates the steps involved in the process.

```
Environment *ev; /* environment to monitor for exceptions */
/* the following variable will store the value returned from the \n
somdCreate call. This will be released after the object is \n
initialized via the init_for_object_creation method */
somLifeCycle_ServerSetLocation  *pre_initSSL;
/* this will be our fully initialized object reference */
somLifeCycle_ServerSetLocation  *mySSL;
/* create the ServerSetLocation object */
pre_initSSL = (somLifeCycle_ServerSetLocation *) ((void *)
    somdCreate(ev, "somLifeCycle::ServerSetLocation", FALSE));
mySSL = (somLifeCycle_ServerSetLocation *)
    ((void *) pre_initSSL->init_for_object_creation(ev));
/* release the reference returned from the new call since the \n
object reference we'll use is the fully initialized mySSL */
((SOMDObject *)pre_initSSL)->release(ev);
```

Combining Location Objects to Create New Ones: It is possible to combine one or two existing **ServerSetLocation** objects via the **location_union** or **location_intersection** methods to create a new **ServerSetLocation** consisting of the result of the union or intersection. For instance, if the application development environment consists of many servers that are shared by various departments, these methods provide a simple way to create and configure location scope without having to specifically create a **ServerSetLocation** and invoke the **add_server_id** or **add_server_ids** methods for setup.

Using location_union

It may be useful to obtain the union of two existing **ServerSetLocation** objects and create a third **ServerSetLocation** containing the result. Consider the situation where Department 100 uses "D100Servers" which is a **ServerSetLocation** with two servers registered with it and Department 200 uses "D200Servers" which is a **ServerSetLocation** with three different servers registered with it. The company decides to form Department 300, and the servers assigned to the new department are all of the servers available to Department 100 and Department 200. The administrator simply has to invoke **location_union** on D100Servers passing in the D200Servers object on the call. The result will be a **ServerSetLocation** object containing all five of the servers. The administrator can then bind the new **ServerSetLocation** object with the Naming Service with the name "D300Servers". See Chapter 3, “Life Cycle Service” on page 3-1 for the syntax of the **location_union** method.

Using location_intersection

It may also be useful to obtain the intersection of two existing **ServerSetLocation** objects and create a third **ServerSetLocation** containing the result. Consider the situation where Department 100 uses "D100Servers" which is a **ServerSetLocation** with two servers registered with it and Department 200 uses "D200Servers" which is a **ServerSetLocation** with three servers registered with it. Two of the servers in D200Servers are also registered in D100Servers. The company decides to form Department 300, and the servers assigned to the new department are the servers shared by Department 100 and Department 200. The administrator simply has to invoke **location_intersection** on D100Servers passing in the D200Servers object on the call. The result will be a **ServerSetLocation** object containing the two common servers. The administrator can then bind the new **ServerSetLocation** object with the Naming Service with the name "D300Servers." See Chapter 3, “Life Cycle Service” on page 3-1 for the syntax of the **location_intersection** method.

Using Life Cycle Service Objects: It is also possible to use the Life Cycle Service objects to create instances of other Life Cycle Service objects. For instance, if a **GenericFactory** exists in the environment and is able to create instances of **ServerSetLocation** objects, the administrator could simply invoke the **create_object** method on the **GenericFactory** passing in a **CosLifeCycle::Key** specifying an object interface of **ServerSetLocation**. An obvious advantage to this approach is that the user need not know if the object is created locally or remotely.

The default LifeCycleGenericFactory provided may also be used in the same manner to create a **ServerSetLocation**.

Creating FactoryFinder Objects

The **FactoryFinder** objects must also be created and reside on an Object Services Server. Several options also exist for the creation of a **FactoryFinder** object, depending on where the object is being created or if the Life Cycle Service objects are used.

Local Object Creation

If the application is running on an Object Services Server, it will be possible to create a **FactoryFinder** object locally by invoking **new** (for C++) or **somNewNoInit**

(for C) followed by **init_for_object_creation**. The following code example illustrates the steps involved in the process.

```
Environment *ev; /* environment to monitor for exceptions */
/* the following variable will store the value returned from the \n
new call. This will be released after the object is initialized \n
via the init_for_object_creation method */
somLifeCycle_FactoryFinder
    *pre_initFF;
/* this will be our fully initialized object reference */
somLifeCycle_FactoryFinder
     *myFF;
/* create the FactoryFinder object */
pre_initFF = new somLifeCycle_FactoryFinder;
myFF = (somLifeCycle_FactoryFinder *)
    ((void *) pre_initFF->init_for_object_creation(ev));
/* release the reference returned from the new call since the \n
object reference we'll use is the fully initialized myFF */
((SOMDObject *)pre_initFF)->release(ev);
```

Remote Object Creation

If the application is not running on an Object Services Server, it will be necessary to locate an Object Services Server and create the **FactoryFinder** object remotely by invoking **somdCreate** with FALSE as the *init* parameter followed by **init_for_object_creation**. The following code example illustrates the steps involved in the process.

```
Environment *ev; /* environment to monitor for exceptions */
/* the following variable will store the value returned from the \n
somdCreate call. This will be released after the object is \n
initialized via the init_for_object_creation method */
somLifeCycle_FactoryFinder
     *pre_initFF;
/* this will be our fully initialized object reference */
somLifeCycle_FactoryFinder *myFF;
/* create the FactoryFinder object */
pre_initFF = (somLifeCycle_FactoryFinder *) ((void *)
    somdCreate(ev, "somLifeCycle::FactoryFinder", FALSE));
myFF = (somLifeCycle_FactoryFinder *)
    ((void *) pre_initFF->init_for_object_creation(ev));
/* release the reference returned from the new call since the \n
object reference we'll use is the fully initialized myFF */
((SOMDObject *)pre_initFF)->release(ev);
```

Using Life Cycle Service Objects

It is also possible to use Life Cycle Service objects to create instances of **FactoryFinder** objects, as described in “Creating ServerSetLocation Objects” on page 3-38.

Creating GenericFactory Objects

The same options are available for creating **GenericFactory** objects as those used to create **FactoryFinder** objects. The **GenericFactory** objects must also be created and reside on an Object Services Server.

Local Object Creation

If the application is running on an Object Services Server, a **GenericFactory** object may be created locally by invoking **new** (for C++) or **somNewNoInit** (for C) followed by **init_for_object_creation**. The following code example illustrates the steps involved in the process.

```
Environment *ev; /* environment to monitor for exceptions */
/* the following variable will store the value returned from the \n
new call. This will be released after the object is initialized \n
via the init_for_object_creation method */
somLifeCycle_GenericFactory
    *pre_initGF;
/* this will be our fully initialized object reference */
somLifeCycle_GenericFactory
     *myGF;
/* create the GenericFactory object */
pre_initGF = new somLifeCycle_GenericFactory;
myGF = (somLifeCycle_GenericFactory *)
    ((void *) pre_initGF->init_for_object_creation(ev));
/* release the reference returned from the new call since the \n
object reference we'll use is the fully initialized myGF */
((SOMDObject *)pre_initGF)->release(ev);
```

Remote Object Creation

If the application is not running on an Object Services Server, it will be necessary to locate an Object Services Server and create the **GenericFactory** object remotely by invoking **somdCreate** with FALSE as the *init* parameter followed by **init_for_object_creation**.

```
Environment *ev; /* environment to monitor for exceptions */
/* the following variable will store the value returned from the \n
somdCreate call. This will be released after the object is \n
initialized via the init_for_object_creation method */
somLifeCycle_GenericFactory
     *pre_initGF;
/* this will be our fully initialized object reference */
somLifeCycle_GenericFactory *myGF;
/* create the GenericFactory object */
pre_initGF = (somLifeCycle_GenericFactory *) ((void *)
    somdCreate(ev, "somLifeCycle::GenericFactory", FALSE));
myGF = (somLifeCycle_GenericFactory *)
    ((void *) pre_initGF->init_for_object_creation(ev));
/* release the reference returned from the new call since the \n
object reference we'll use is the fully initialized myGF */
((SOMDObject *)pre_initGF)->release(ev);
```

Using Life Cycle Service Objects

It is also possible to use Life Cycle Service objects to create instances of **GenericFactory** objects, as described in “Creating ServerSetLocation Objects” on page 3-38.

Subclassing the ConstraintBuilder Interface

It is possible to create instances of the **ConstraintBuilder** interface provided, but since it is abstract, designed to be subclassed and no meaningful implementation is provided for the methods, it is not recommended. Instead, it may be desirable to subclass the **ConstraintBuilder** interface and implement the **build_constraint** method to suit your application environment policies and requirements. This concept is very useful in cases where static policies need to be enforced.

Let's continue with our Movie Rental scenario. We previously identified a need for a subclass of **ConstraintBuilder** that is able to build an **ExtendedNaming::ExtendedNamingContext::Constraint** that consists of the operating system preference of the user. (See “How Constraint Expressions Passed to the DSOM Factory Service by a FactoryFinder are Built” on page 3-27 for more information on the **ConstraintBuilder** objects needed for the Movie Rental scenario.) The administrator creates an IDL file that defines the OperSysCB interface, which inherits from the **somLifeCycle::ConstraintBuilder** interface. The administrator may also inherit from other interfaces that provide other requirements for the environment, such as persistence. The OperSysCB interface will override the **build_constraint** method and provide an implementation that will evaluate the parameter passed in on the **build_constraint** invocation. The OperSysCB interface also produces a constraint specifying that the OpSys property matches the operating system specified in the parameter. The logic for the implementation may be similar to the following:

```
if (parameter is recognized as a valid operating system)
     return "OpSys == '<parameter>'"
else raise an exception
```

The return value must be a valid **ExtendedNaming::ExtendedNamingContext::Constraint** as defined by the Naming Service. See Chapter 4, “Naming Service” on page 4-1 of Programmer' s Guide for Object Services for more information on creating a valid **ExtendedNaming::ExtendedNamingContext::Constraint** based on the Naming Constraint Language. The implementation is also responsible for raising any exceptions necessary. Also, since the **ConstraintBuilder** interface inherits from the **LifeCycleObject** interface, any subclasses of **ConstraintBuilder** should list the **LifeCycleObject__Proxy** interface as the base proxy class (baseproxyclass modifier) in the implementation section of the IDL.

Once the OperSysCB subclass has been defined and implementation provided for the **build_constraint** method, instances of the interface may be registered with various **FactoryFinder** objects or the **build_constraint** method may be invoked directly.

Subclassing the FactoryFilter Interface

It is possible to create instances of the Life Cycle Service **FactoryFilter**, but since no implementation is provided it is not recommended. The **FactoryFilter** interface is abstract and designed to be subclassed to provide dynamic filtering capabilities that are meaningful to the application development environment. The purpose of the **FactoryFilter** is to filter out factory objects from the list of matches returned from the DSOM Factory Service.

For example, in the movie rental scenario the administrator may wish to control where an object (such as a MovieRental::Movie) is created based on CPU usage on the server. In this example, it could be an OS/2 server (since OS/390 manages its own resources based on a set of performance goals, etc.). The administrator creates an IDL file that defines the CPUFilter, which inherits from the **somLifeCycle::FactoryFilter** interface. The administrator may also inherit from other interfaces that provide other requirements for the environment, such as persistence. The CPUFilter interface will override the **filter_factories** method and provide an implementation that will eliminate any factories that reside on servers that have a CPU usage percentage higher than the value passed to the **filter_factories** method upon invocation. The logic for the implementation may be similar to the following:

```
if (key_id parameter is recognized as valid)
     while (there are still factories to examine)
     obtain server of factory in question
     obtain CPU usage percentage of that server
     if (server CPU usage > key_id parameter )
     remove factory from the sequence
     free the memory allocated for the removed factory
     return the sequence of remaining factories to the user
else raise an exception
```

The return value must be a valid **somLifeCycle::Factories** sequence. The implementation is responsible for raising any exceptions necessary. Also, since the **FactoryFilter** interface inherits from the **LifeCycleObject** interface, any subclasses of **FactoryFilter** should list the **LifeCycleObject__Proxy** interface as the base proxy class (baseproxyclass modifier) in the implementation section of the IDL.

Once the CPUFilter subclass has been defined and implementation provided for the **filter_factories** method, instances of the interface may be registered with various **FactoryFinder** objects or the **filter_factories** method may be invoked directly.

Destroying Life Cycle Service Objects

All of the Life Cycle Service interfaces inherit from the **LifeCycleObject** interface. Simply invoke the **remove** method on the Life Cycle Service object to destroy. The object and corresponding proxy, if one existed, are destroyed.

This feature provides local/remote transparency since the user need not know where the object resides to request the destruction of the object.

Configuration Operations on Life Cycle Service Objects

The Life Cycle Service objects provide methods that allow you to modify the state of the objects. Each interface may be modified to meet the specific requirements of your environment. This section outlines many useful methods that may be used to tailor the application development environment. See Chapter 3, “Life Cycle Service” on page 3-1 for code examples illustrating the usage of the methods outlined in this section.

Registering Servers with a ServerSetLocation object

Depending on the size of the development environment and the number of servers available, it may be desirable to define location scopes using **ServerSetLocation** objects or other user-defined location objects. It is possible to register any number of servers with a particular **ServerSetLocation** object. The **add_server_id** and **add_server_ids** methods are provided to handle the registration process for you. For example, if you have a **ServerSetLocation** and would like to register two servers with the object, the **add_server_id** and **add_server_ids** methods could accomplish the task for you.

Removing Servers from a ServerSetLocation Object

A situation may arise where an administrator needs to remove registered servers from a **ServerSetLocation** object, possibly if the server is out of commission for a while, or the administrator needs to reconfigure portions of the development environment. The **remove_server_id** and **remove_server_ids** methods are provided to handle the registration process for you.

Listing Servers Associated with a Location

In a large environment with many servers and **Location** objects, it is possible to forget which servers are registered with which **Location** objects. The **list_server_ids** method is implemented by the **ServerSetLocation** interface and will return a list of the servers registered with the **ServerSetLocation**.

This method is defined by the **Location** interface, but is abstract with no provided implementation. This allows subclasses to override and provide their own implementation if desired. The Life Cycle Service provides an implementation for **list_server_ids** on a **ServerSetLocation**, which is a subclass of **Location**.

Checking to See if a Server is Registered with a Location

There will be occasions when the administrator or application developer may just be interested in knowing if a specific server is registered with a **Location**. Rather than use the **list_server_ids** method which lists every server currently registered with the **Location**, the **is_server_in_location** method can be used to quickly identify if a particular server is registered with a specific **Location** object. A value of TRUE will be returned if the server is registered with the **Location** or FALSE if it is not registered.

Checking to See if an Object is within a Location

There will be occasions when the administrator or application developer may be interested in knowing if a particular object may be instantiated on one of the servers registered with a **Location**. Rather than invoke the **list_server_ids** method and compare each with the ImplId of the object, the **is_object_in_location** method can be used to quickly identify if a particular object can be instantiated on one of the registered servers. A value of TRUE will be returned if a server exists that will allow creation of the desired object. If no server exists with the capability desired, FALSE will be returned.

Registering a Location with a FactoryFinder Object

A **FactoryFinder** object may have one **Location** object registered with it that will define the location scope for the **find_factories** and **find_factory** operations. The Life Cycle Service provides the **set_location_scope** method which will perform this operation. The administrator can set the scope by invoking the **set_location_scope** method and passing in a valid **Location** object. All future **find_factories** and **find_factory** method invocations will consider the location information currently registered with the **FactoryFinder** in factory searches.

It is possible to specify a NULL reference as the parameter on the **set_location_scope** method. If the location scope is set to NULL, server information will not be considered in the factory searches performed by the **FactoryFinder**.

Removing a Location from a FactoryFinder Object

If you currently have a **Location** registered with a **FactoryFinder** and would like to replace it with another **Location**, simply invoke the **set_location_scope** method, passing in the preferred **Location**. This will replace the current **Location** registered with the **FactoryFinder** with the preferred one. If you have a **Location** registered, and decide you do not want server information to be considered during searches for factories, invoke the **set_location_scope** method passing in a NULL reference for the location parameter.

When a **Location** registered with a **FactoryFinder** is replaced by another one, the previous reference is not deleted. The **FactoryFinder** no longer has a reference to the object.

Retrieving a Location from a FactoryFinder Object

If it is desirable to obtain a reference to the **Location** registered with a **FactoryFinder**, it is possible to obtain one by invoking the **get_location_scope** method on the **FactoryFinder**. If a **Location** is registered with the **FactoryFinder**, a valid reference to the **Location** object will be returned. The user is then able to invoke methods on the location returned. If there is no **Location** registered with the **FactoryFinder**, OBJECT_NIL will be returned.

Registering a ConstraintBuilder with a FactoryFinder Object

A **FactoryFinder** object may have any number of **ConstraintBuilder** objects registered with it that will define additional constraints to be considered during the **find_factories** and **find_factory** operations. The Life Cycle Service provides the **add_constraint_builder** method which will perform this operation. The administrator can register a **ConstraintBuilder** with a **FactoryFinder** by invoking the **add_constraint_builder** method and passing in a unique identifying name (*key_kind* parameter), a reference to the **ConstraintBuilder** object (*builder* parameter) and the default parameter value to be used on **build_constraint** method invocations (*default_key_id* parameter). All future **find_factories** and **find_factory** method invocations will consider the constraint information provided by the **ConstraintBuilder** objects currently registered with the **FactoryFinder** during factory searches.

It is possible to register the same **ConstraintBuilder** object reference with a single **FactoryFinder** several times using different identifying names (and perhaps different default values) each time. The main purpose for using this feature would be if the **build_constraint** method is capable of creating different constraints depending on the values specified in the **CosLifeCycle::Key**.

Remember that each **ConstraintBuilder** must be registered with a unique identifying name. If the user intends to modify the default parameter value of a **ConstraintBuilder** already registered with the **FactoryFinder**, the **ConstraintBuilder** must first be removed from the **FactoryFinder** so it no longer shows as registered. The user may then invoke the **add_constraint_builder** method passing in the *key_kind* identifying name (it may be the same as the one removed, since it is no longer registered and won't be considered a duplicate), the **ConstraintBuilder** object reference and the preferred *default_key_id*.

Removing a ConstraintBuilder from a FactoryFinder Object

The Life Cycle Service provides the **remove_constraint_builder** method that may be invoked on the **FactoryFinder** to remove a **ConstraintBuilder** currently registered with the **FactoryFinder**. Simply invoke the method passing in the identifying name of the **ConstraintBuilder** to remove, and the **ConstraintBuilder** will no longer be considered during operations performed by the **FactoryFinder**.

When it is removed from the **FactoryFinder**, the **ConstraintBuilder** object is not deleted; the object just is not registered with the **FactoryFinder** anymore under that identifying name. It may still be registered with the **FactoryFinder** under different identifying names, or registered with other **FactoryFinder** objects.

Retrieving the ConstraintBuilders from a FactoryFinder Object

The Life Cycle Service provides the **list_constraint_builders** method which will provide a listing of all **ConstraintBuilder** objects currently registered with the **FactoryFinder** along with the identifying names for each and the associated default parameter values. The user may then evaluate the list of **ConstraintBuilder** objects in any way that is meaningful. This feature is very helpful for finding the identifying names or default values associated with specific **ConstraintBuilder** objects to determine if modifications to the **FactoryFinder** are desired.

Registering a FactoryFilter with a FactoryFinder Object

A **FactoryFinder** object may have any number of **FactoryFilter** objects registered with it that will define additional filtering policies to consider after the factory searches have been performed. The Life Cycle Service provides the **add_factory_filter** method which will perform this operation. The administrator can register a **FactoryFilter** with a **FactoryFinder** by invoking the **add_factory_filter** method and passing in a unique identifying name (*key_kind* parameter), a reference to the **FactoryFilter** object (*filter* parameter) and the default parameter value to be used on **filter_factories** method invocations (*default_key_id* parameter).

It is possible to register the same **FactoryFilter** object reference with a single **FactoryFinder** several times using different identifying names (and perhaps different default values) each time. The main purpose for using this feature would be if the **filter_factories** method is capable of performing different filtering functions based on the values specified in the **CosLifeCycle::Key**.

Remember that each **FactoryFilter** must be registered with a unique identifying name. If the user intends to modify the default parameter value of a **FactoryFilter** already registered with the **FactoryFinder**, the **FactoryFilter** must first be removed from the **FactoryFinder** so it no longer shows as registered. The user may then invoke the **add_factory_filter** method passing in the *key_kind* identifying name (it may be the same as the one removed, since it is no longer registered and won't be considered a duplicate), the **FactoryFilter** object reference and the preferred *default_key_id*.

Removing a FactoryFilter from a FactoryFinder Object

The Life Cycle Service provides the **remove_factory_filter** method that may be invoked on the **FactoryFinder** to remove a **FactoryFilter** currently registered with the **FactoryFinder**. Simply invoke the method passing in the identifying name of the **FactoryFilter** to remove, and the **FactoryFilter** will no longer be considered during operations performed by the **FactoryFinder**.

When it is removed from the **FactoryFinder**, the **FactoryFilter** object is not deleted; the object just is not registered with the **FactoryFinder** anymore under that identifying name. It may still be registered with the **FactoryFinder** under different identifying names, or registered with other **FactoryFinder** objects.

Retrieving the LifeCycleFactoryFilters from a FactoryFinder Object

The Life Cycle Service provides the **list_factory_filters** method which will provide a listing of all **FactoryFilter** objects currently registered with the **FactoryFinder** along with the identifying names for each and the associated default parameter values. The user may then evaluate the list of **FactoryFilter** objects in any way that is meaningful. This feature is very helpful for finding the identifying names or default values associated with specific **FactoryFilter** objects to determine if modifications to the **FactoryFinder** are desired.

When the Configuration has been Completed

The administrator may want to create a list of the tailored **GenericFactory**, **FactoryFinder** and (depending on the policies of the development environment) **ServerSetLocation, ConstraintBuilder** and **FactoryFilter** objects available. For the process to work smoothly, the administrator must inform the developers as to which objects they should use. The administrator is also responsible for registering the tailored objects with the Naming Service, using easily identifiable names so developers are able to find them quickly. See Chapter 4, “Naming Service” on page 4-1 of Programmer' s Guide for Object Services for the steps necessary for binding objects into the Naming Service.

Ideally, all developers will have to do is create **CosLifeCycle::Key** and **CosLifeCycle::Criteria** structures specifying what they want to create, retrieve references to the existing Life Cycle Service objects created by the administrator, and invoke **create_object** or **find_factories** requests. The administrator could create a file of **CosLifeCycle::Key** and **CosLifeCycle::Criteria** definitions also, which would make development more efficient in the case where certain objects are created often. Ideally, developers will not modify the state of the tailored Life Cycle Service objects, only the administrator. This is especially important in large development environments. In smaller environments it may be more desirable to have a team of developers that are able to modify Life Cycle Service attributes to meet individual needs.

Using the Life Cycle Service in Application Development

The Life Cycle Service provides many benefits to the application developer. If the development environment has been tailored, the typical developer does not have to know how Life Cycle processes requests. Simple method invocations will perform complex requests without intervention and monitoring by the user. This section will cover some of the potential uses of the Life Cycle Service in application develop-

ment. This includes building objects that inherit from the **LifeCycleObject** interface, different ways to create objects, and helpful programming tips.

Even though the environment configuration was most likely performed by an administrative team, it would be helpful for the programmer to understand the relationships of the Life Cycle Service objects and the configuration mechanisms available. See the previous portions of this chapter for detailed information on the Life Cycle Server interfaces, relationships with other services, key building blocks, and configuration mechanisms that may be helpful.

To ensure application portability across platforms, it may be desirable to conform to industry standards. The Life Cycle Service meets the standards specified by OMG for the Life Cycle Object Service. The **LifeCycleObject** interface provides the definitions of the **copy, move** and **remove** methods, and the **GenericFactory** provides a generic means of object creation.

Application developers can build their interfaces to inherit from the **LifeCycleObject** interface to use the OMG compliance features. This will make porting of the applications much easier since platforms offering an implementation of the Life Cycle Service will recognize the method invocations.

Since the **LifeCycleObject** interface offers local/remote transparency with regard to the destruction of objects, all interfaces that inherit from the **LifeCycleObject** interface must list the **LifeCycleObject__Proxy** interface as the base proxy class (baseproxyclass modifier) in the implementation portion of the IDL definition. The **LifeCycleObject__Proxy** interface performs the steps necessary to destroy the object and the associated proxy object when a **remove** request is made on a remote **LifeCycleObject**. The client is then able to simply invoke **remove** (and eventually **copy** and **move**) without knowing where the object resides.

Creating objects

The Life Cycle Service provides the user with several options for object instantiation. One alternative is to submit a **create_object** request to **GenericFactory** and wait for the object to be returned. Alternatively, the user can submit a **find_factories** request to the **FactoryFinder**, select a factory from the sequence returned, and use it to create the object.

Deciding which Life Cycle Service Objects to Use

If the development environment has been tailored by the administrator, there will be a variety of Life Cycle Service objects available for use. Ideally the programmer will have received a listing of the easily-identifiable names of the Life Cycle Service objects to use from the administrator. The client can then retrieve a reference to the specific Life Cycle Service object from the Naming Service.

If no specific tailoring was done for the environment, the default **GenericFactory** (LifeCycleGenericFactory) and default **FactoryFinder** (LifeCycleFactoryFinder) may be used.

The client may also want to use the information retrieval methods provided by the Life Cycle Service to obtain specific information about a Life Cycle Service object. For instance, if the client would like to perform a **find_factories** request on a **FactoryFinder** but is interested in reviewing the listing of the servers registered with the **FactoryFinder** first, the programmer could first retrieve the **Location** registered with the **FactoryFinder** by invoking **get_location_scope**, and invoke the

list_server_ids method to list the servers registered with the **Location** object. The list functions provide a simple way to obtain the registered items from a Life Cycle Service object.

The **supports** method is also very useful when trying to decide if a particular **GenericFactory** is capable of creating an instance of a desired object. The client can invoke the **supports** method on the **GenericFactory** object in question, passing in a **CosLifeCycle::Key** specifying the object interface and any additional constraint or filtering information. The **supports** method will return the boolean value TRUE if a factory is found that is able to create an instance of the object specified. The value of FALSE will be returned if the **GenericFactory** does not possess the capability to create the object.

Using the create_object method

Once the client knows the name of the **GenericFactory** to use for the creation request, it is possible to obtain a reference to that **GenericFactory** object from the Naming Service. The client must first invoke the **resolve_initial_references** method to obtain a reference to a naming context object. It is then possible to use any of the resolve methods provided by the Naming Service to retrieve a reference to an object bound into the naming context. See Chapter 4, “Naming Service” on page 4-1 for more information on using the Naming Service.

Once the client has a reference to the **GenericFactory** to invoke the **create_object** request on, the next step is to actually invoke the method passing in the **CosLifeCycle::Key** and **CosLifeCycle::Criteria** parameters. See “Building the CosLifeCycle::Key Parameter” on page 3-24 for more information on the **CosLifeCycle::Key**, “Building the CosLifeCycle::Criteria Parameter” on page 3-32 for information on the **CosLifeCycle::Criteria** and “How Objects Are Created using the create_object Method” on page 3-34 for more information on the process flow of the **create_object** method.

The Life Cycle chapter of the Programmer's Reference, Volume II: Object Services provides a detailed description of the syntax of the **create_object** method and a description of what exceptions may be raised by the method.

Using the find_factories (or find_factory) method

Once the client has identified a **FactoryFinder** object to use for the request, it is possible to obtain a reference to that **FactoryFinder** from the Naming Service in the same manner as described above with regard to **GenericFactory**. When the client has the object reference, the **find_factories** or **find_factory** method may be invoked. The **find_factories** method will return all factories that match the specified requirements, while the **find_factory** method will return the first factory found that matches the requirements. The **find_factories** method is useful if a variety of factories is desired and the client wishes to choose from the sequence of factories returned. The **find_factory** method is useful if only one factory reference is needed, and the client does not care which type of reference is returned (**SOMClass**, user-defined factory, or **GenericFactory**).

The client could use either of the methods (**find_factory** or **find_factories**) to obtain a factory reference that is capable of creating an instance of the desired object. The client may then invoke the proper creation request on the factory object. This differs from the **create_object** method in that the user is responsible for identifying the type of factory object returned and deciding what creation method to

invoke on the factory. The **create_object** method will evaluate the factory references and make the creation request for the user.

See “Building the CosLifeCycle::Key Parameter” on page 3-24 for more information on the **CosLifeCycle::Key** and “How Factories Are Found” on page 3-30 for more information on the process flow of the **find_factories** and **find_factory** methods.

The Life Cycle chapter of the Programmer's Reference, Volume II: Object Services provides a detailed description of the syntax of the **find_factory** and **find_factories** methods and a description of what exceptions may be raised by the methods.

Removing objects

The destruction of an object is fairly simple if that object descends from the **LifeCycleObject** interface. Objects that support the **LifeCycleObject** interface are able to use the **remove** method, which will destroy the object and the corresponding proxy if a proxy exists. The client can simply invoke the **remove** method passing in the object to be destroyed. The Life Cycle Service's implementation of **remove** will deal with the local/ remote issues and clean up allocated memory. The client need not be concerned with where the object resides or the processing involved with the actual destruction of the object.

As mentioned previously, to use the **remove** method correctly, any interface that inherits from the **LifeCycleObject** interface must list the **LifeCycleObject__Proxy** interface as the base proxy class (baseproxyclass modifier) in the IDL for the subclass. If the **LifeCycleObject__Proxy** class is not listed as the base proxy class, the proxy will not be destroyed correctly.

If the object to destroy does not support the **LifeCycleObject** interface, the client should follow standard SOM/DSOM practices for object destruction.

Exception Monitoring

The client should always monitor the environment structure for any exceptions that may be raised by the Life Cycle Service or any underlying services used by Life Cycle. The Life Cycle Service has defined several exceptions that may be raised in various situations to help the user identify problem areas. These exceptions are defined in the somlcdef.h file, and may be monitored for by name or by error code.

See Life Cycle Service Error Codes in *OS/390 SOMobjects Messages, Codes, and Diagnosis* for a complete list of the Life Cycle Service exceptions, as well as an explanation of each error and possible steps to take to recover.

The Life Cycle Service will also log the exception in the Error Logging Facility provided this release. See “The Error Log Facility” in *OS/390 SOMobjects Messages, Codes, and Diagnosis*.

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 using DSOM.

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.

Abstract and Concrete Implementations

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 4-1 on page 4-2 and Figure 4-2 on page 4-3.

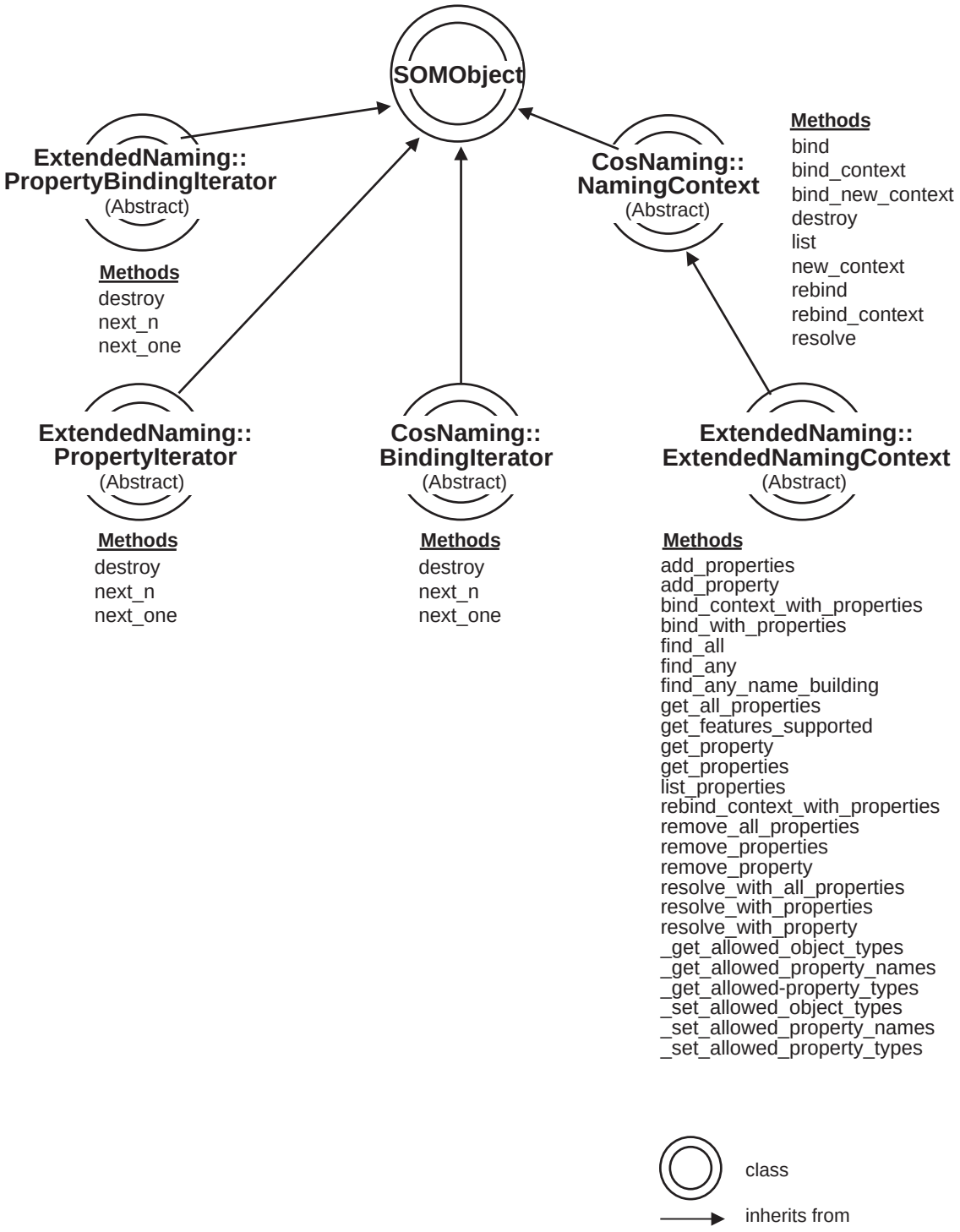

Figure 4-1. Derivation for SOMobjects 3.0 Naming Service, Part One

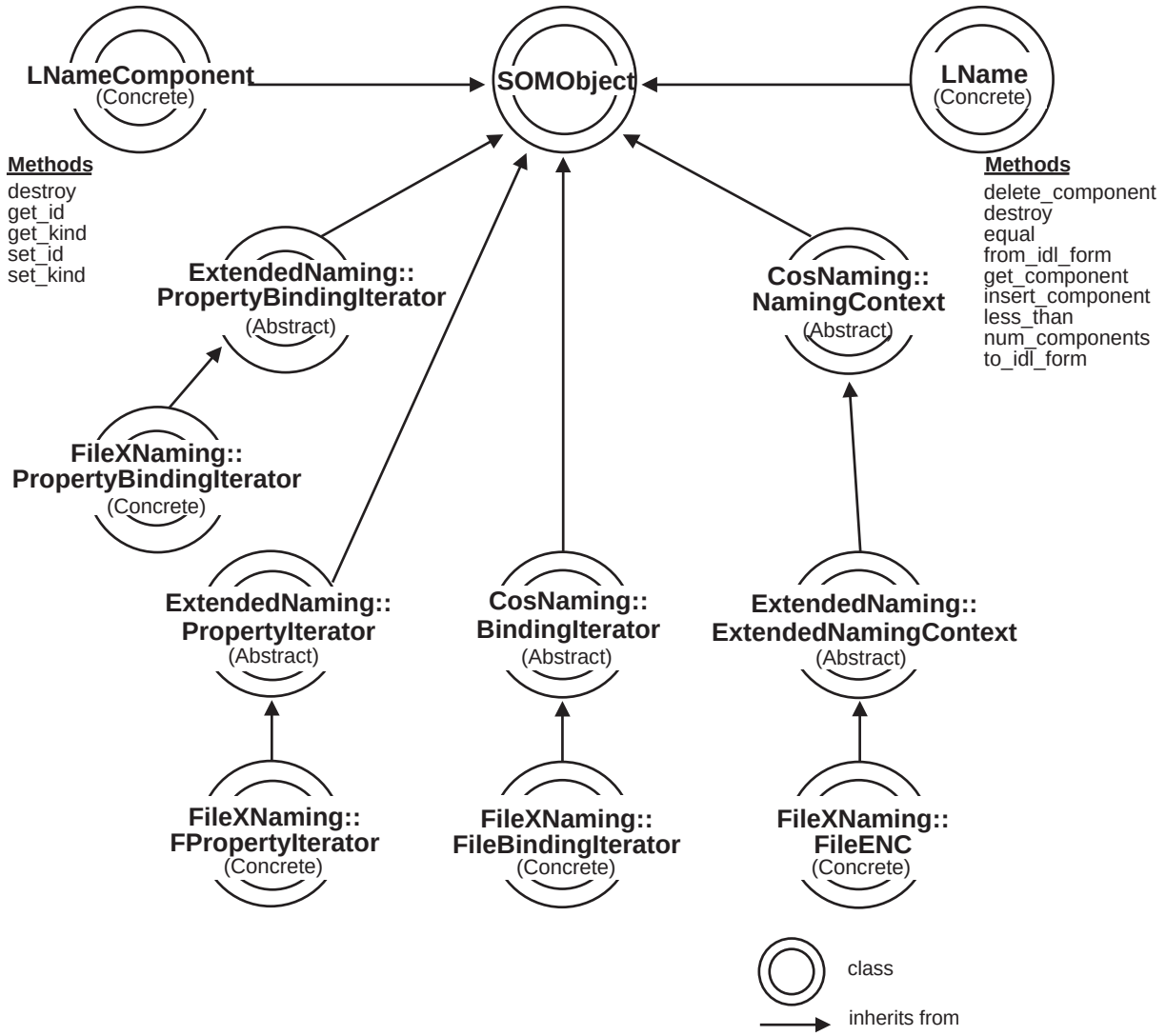

Figure 4-2. Derivation for SOMobjects 3.0 Naming Service, Part Two

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) 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*. 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 4-3 on page 4-5 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

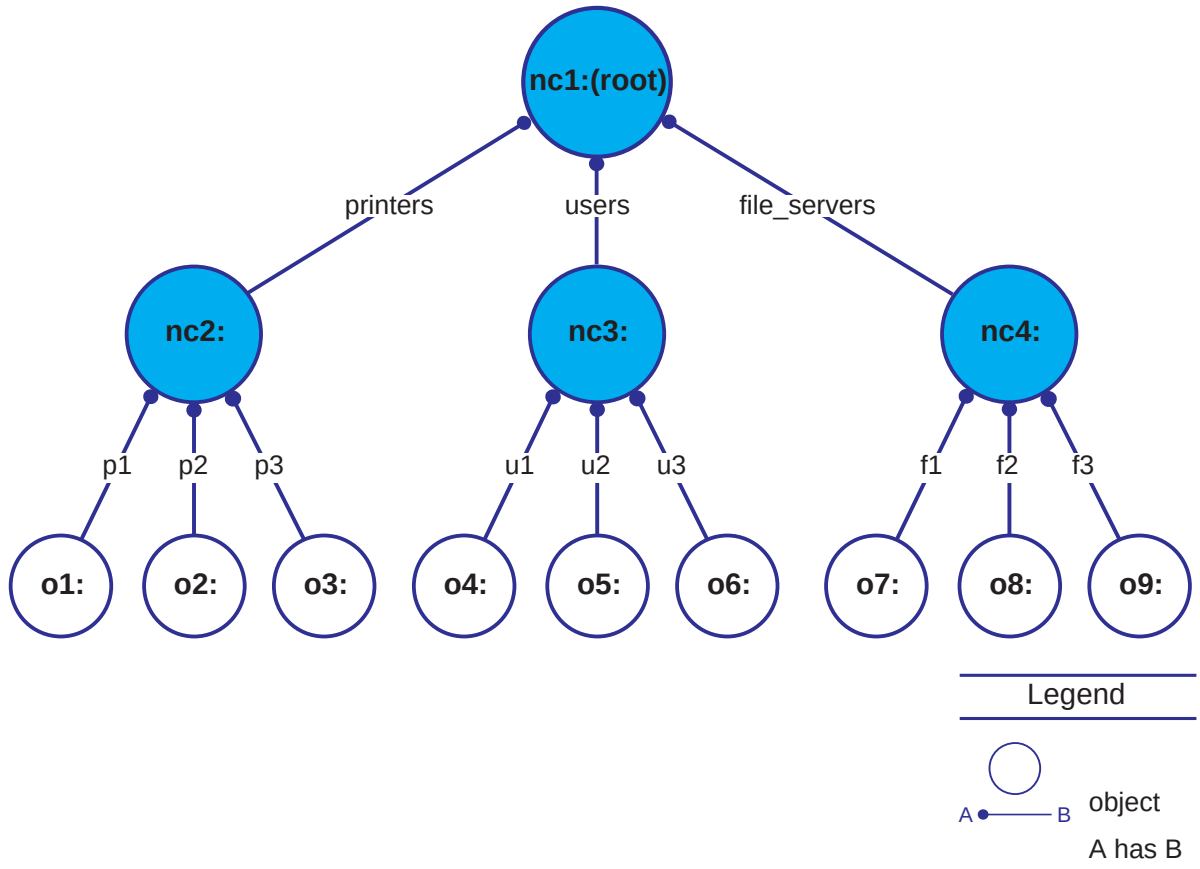

Figure 4-3. Example Name Graph

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 4-4 on page 4-6, 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 rela-

tive 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*.

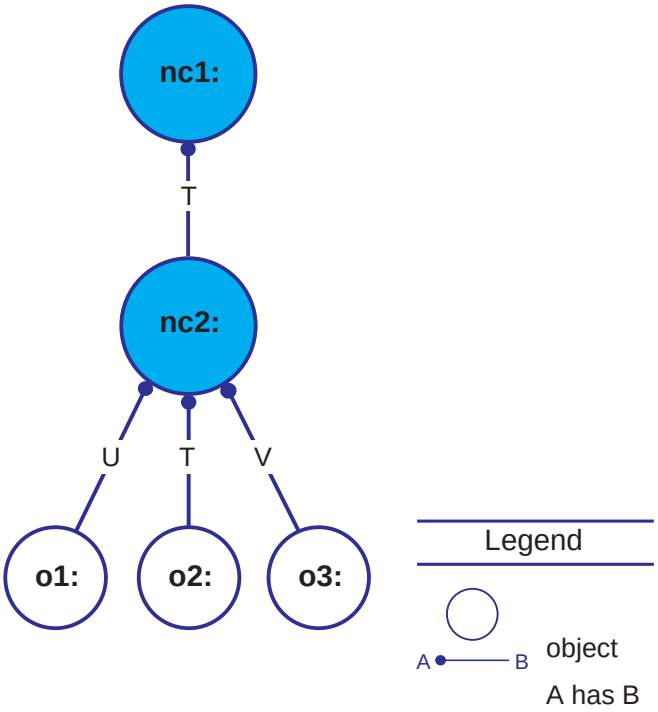

Figure 4-4. Component and Compound Name Examples

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 4-5 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.

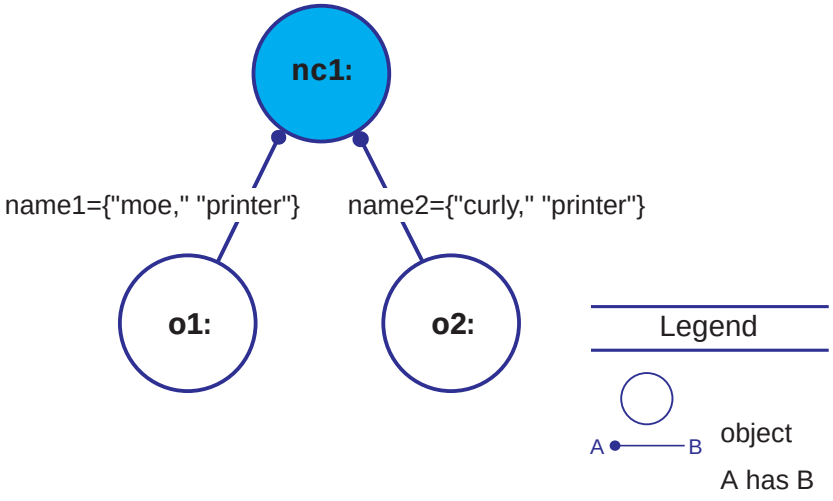

Figure 4-5. A Name Uniquely Identified by id and kind Fields

Properties

A property is a double: **property_name**, and **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;
```

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 4-6 on page 4-8. 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.

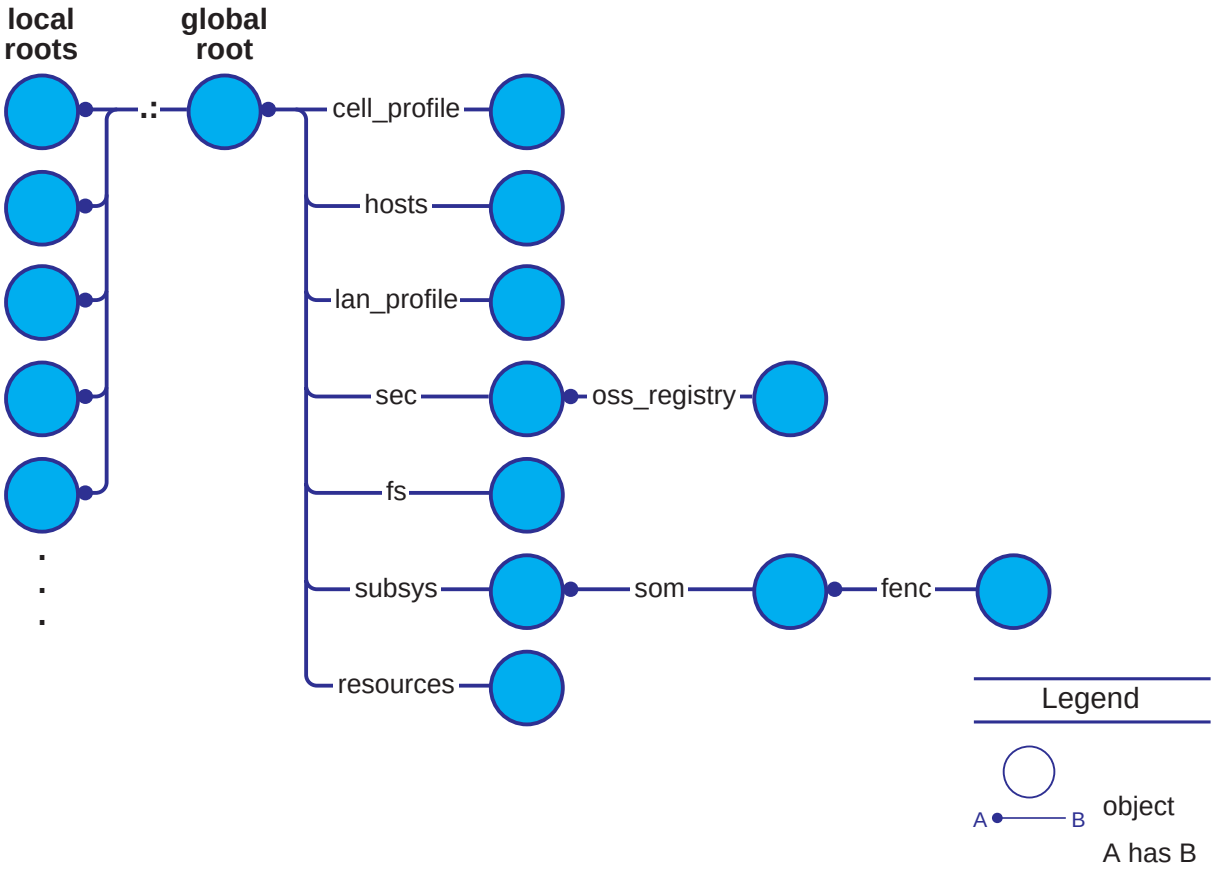

Figure 4-6. Name Space Structure

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, \n
"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 4-4 on page 4-6, 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;
```

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 4-7 on page 4-14.

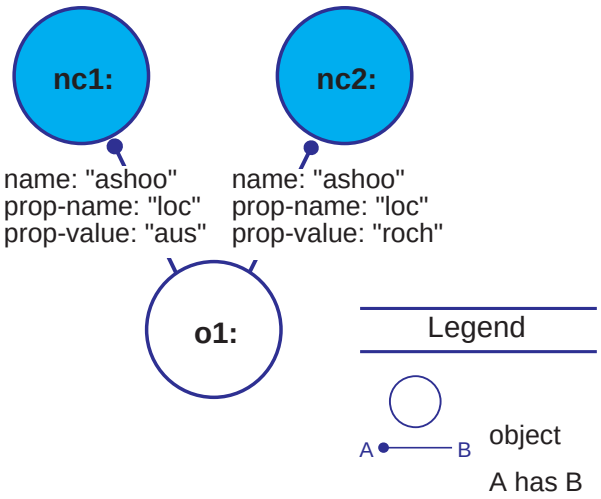

Figure 4-7. An Object Bound with the Same Name in Different Contexts

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], pbl._buffer[i].property_name);
    /* step through the iterator */
if (!_is_nil(pbIterator, &ev)) {
    while (ExtendedNaming_PropertyBindingIterator_next_one
    (pbIterator, &ev, &propBind)){
    somPrintf ("[%s], 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 4-18.

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 alphanumeric 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 adhoc 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 ==
     'file"
```

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 do 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 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** or by using the Persistent Object Service. For more information about the Persistent Object Service, refer to Chapter 6, “Persistent Object Service” on page 6-1. 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. For more information, see *OS/390 SOMobjects Programmer's Guide*.

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 5-1 on page 5-2. 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).

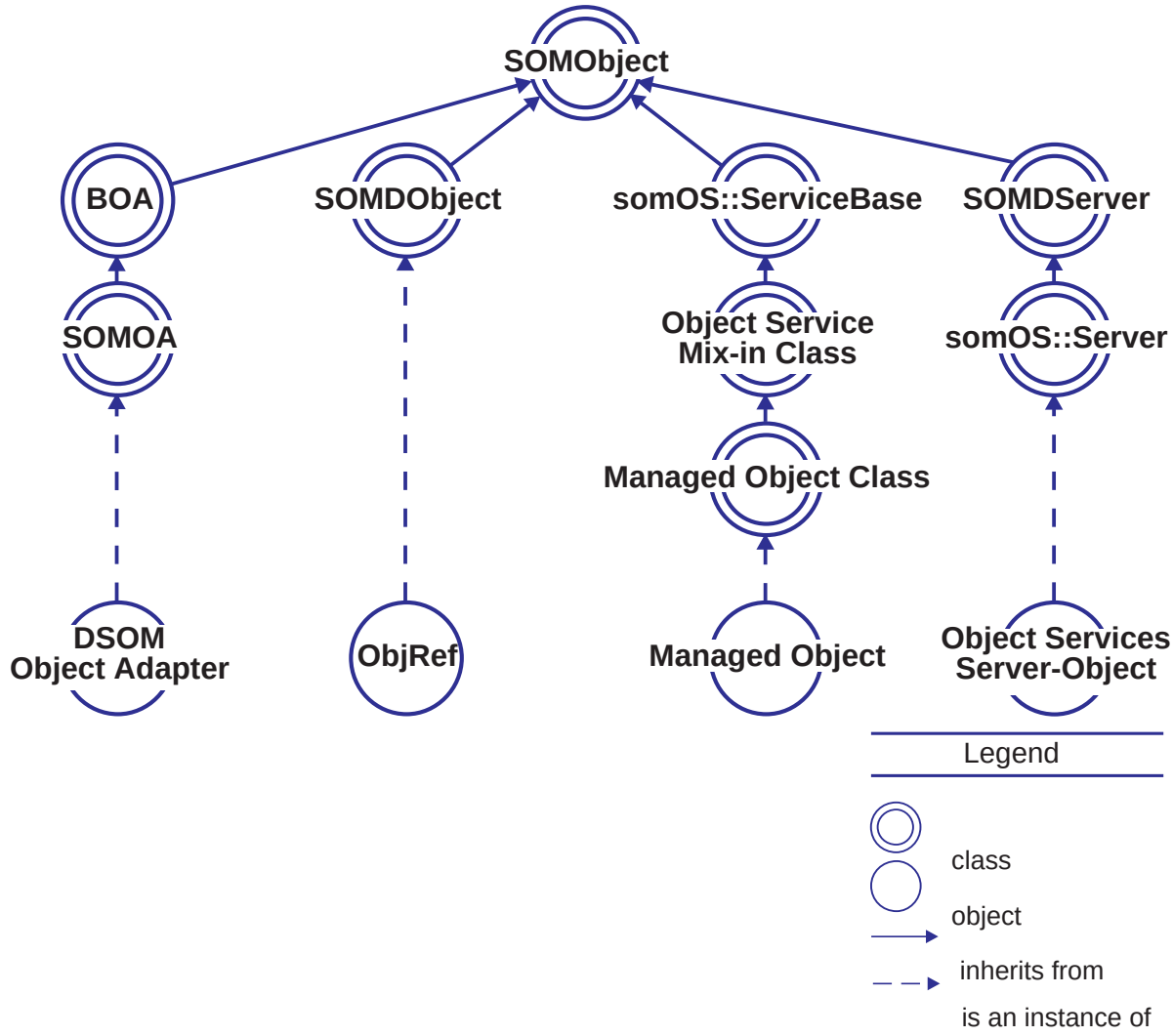

Figure 5-1. Object Services Server is a Specialization of the DSOM Framework

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 5-2 on page 5-3. 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.

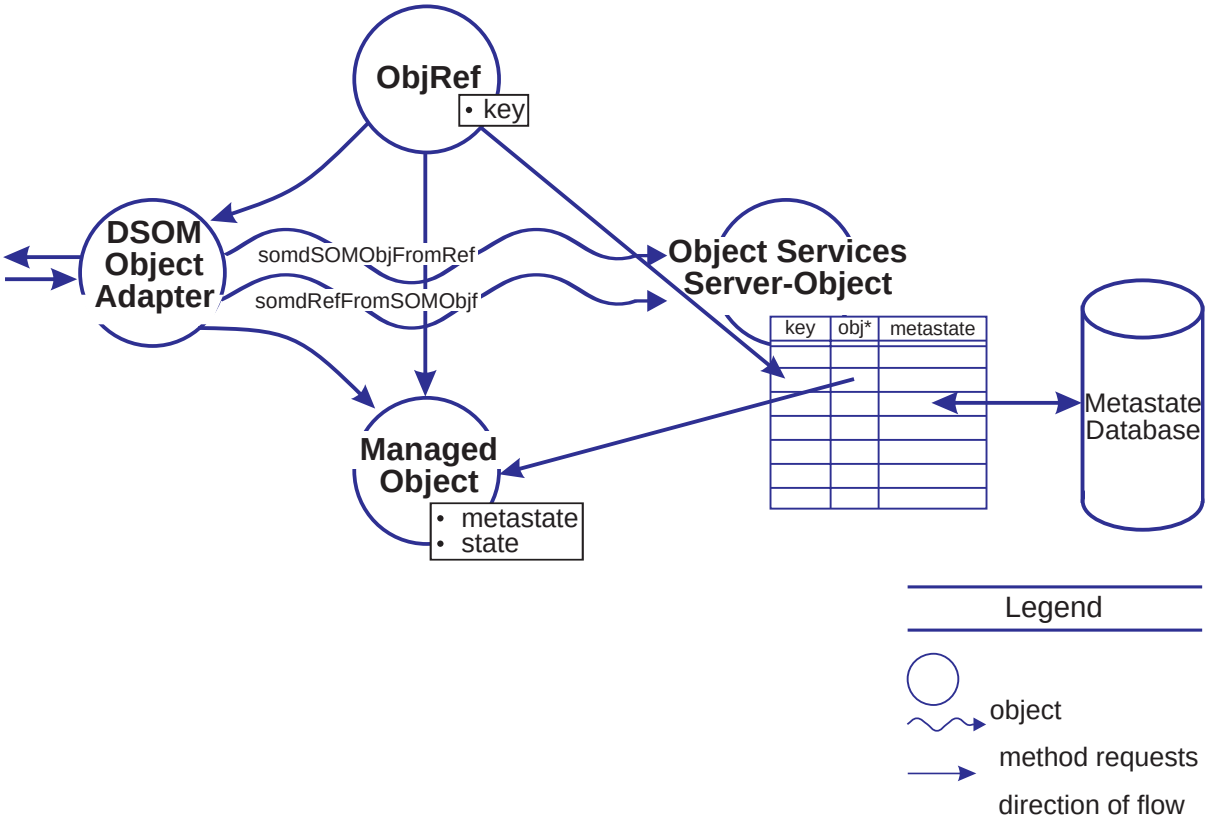

Figure 5-2. Object Services Server Participates in the Exporting and Importing of Object References

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 5-7.

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 5-7.

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 2, “Object Identity Service” on page 2-1.

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 5-3 on page 5-6, 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.

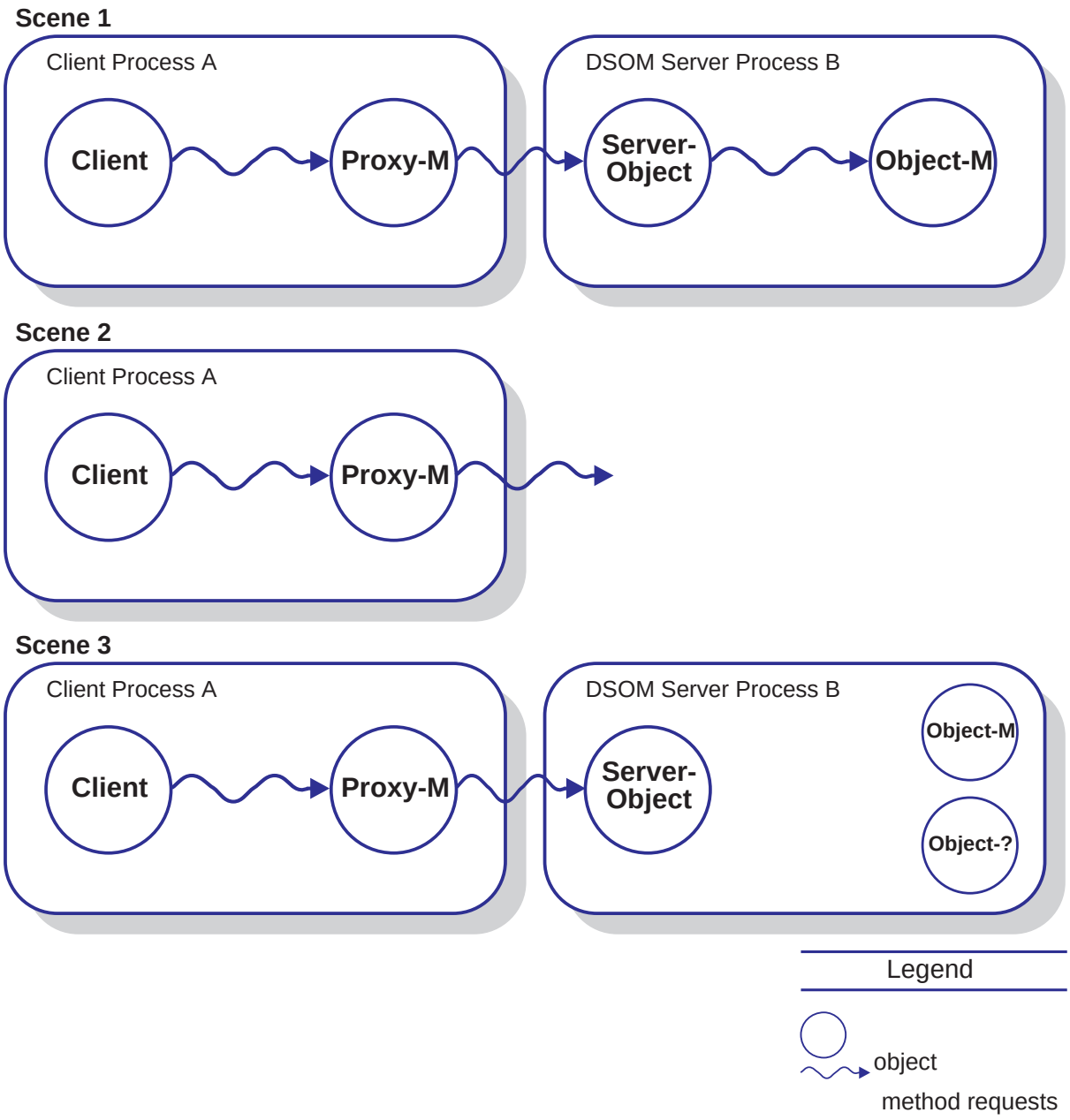

Figure 5-3. Consequences of Transient Object References

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 is used by the Persistence Service to restore the persistent state of the object. See “Overview of the Object Life Cycle Model” on page 5-7 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” 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 5-3 on page 5-6, 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. If you are using the object Persistence Service, that Service handles allocation of persistent storage for you through its stream and PDS support. For more information on the object persistence service refer to Chapter 6, “Persistent Object Service” on page 6-1.

The introduction of specialized initializers deprecated the use of **somDefaultInit**. Likewise, the use of **somDestruct** is limited to removing the in-memory object (not unintializing 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.

Attention: 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 5-3, 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 5-4.

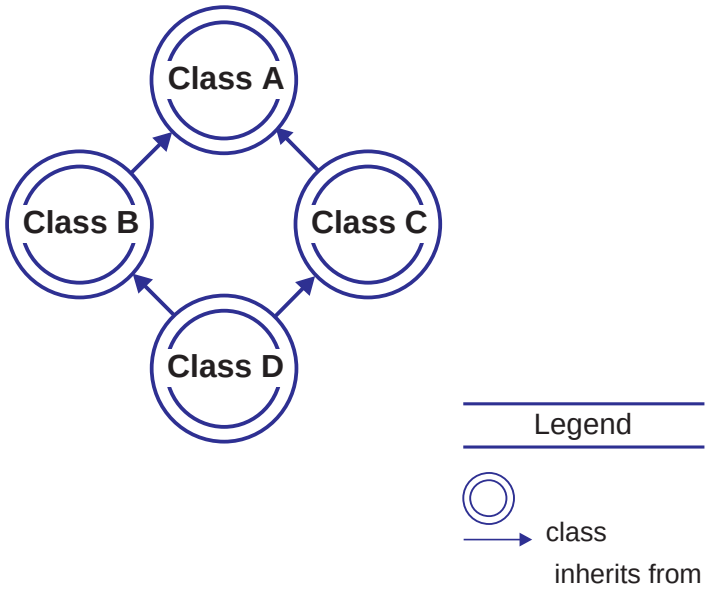

Figure 5-4. Multiple Inheritance and Diamonds

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.

  Attention: 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 unique object services server maintains two binary databases in VSAM datasets, one for metastate data (<somddir>.DBxxxxxx.MDB), and one for persistent attributes (<somddir>.DBxxxxxx.ADB).

<somddir> is the high-level qualifier specified by the SOMDDIR variable in the configuration file under the [somd] stanza heading.

"xxxxxx" is a system-generated unique string.

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 database is a VSAM dataset named <somddir>.SOMOSDB.DAT, also located under the SOMDDIR high-level qualifier. 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 5-12.) For each server, there are two entries in the master database, one for each of the binary databases.

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 5-12 and “Initializing the Server Manually” on page 5-12.

Initializing the Server

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

Initializing the Server Manually

The *-i* 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.

The server must be invoked once with the "-i" option so that the aforementioned database files will be properly initialized. For more information on the somOS::Server, see the *OS/390 SOMobjects Configuration and Administration Guide*.

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;
  /* ... set impl_alias to implementation alias string ... */
  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 in 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** 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. Figure 5-5 on page 5-14 shows the general flow of the Object Services Server program.

```
/*
*  myossvr.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,
           "myossvr [-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;
```

Figure 5-5 (Part 1 of 3). General flow of the Object Services Server program.

```
    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 );
  }
```

Figure 5-5 (Part 2 of 3). General flow of the Object Services Server program.

```
  /* Initialize any required object services -
   * after impl_is_ready */
  somos_init_services_afterimpl(initialize_mode);
  printf( "%s%s%s\n","myOS::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);
}
```

Figure 5-5 (Part 3 of 3). General flow of the Object Services Server program.

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 **somosutil.h** file. Prior to calling any exit callback procedure for an object service, the server passivates all objects.

The preceding program must be compiled on the IBM C/C++ for MVS/ESA compiler that SOMobjects supports.

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.

Configuring Your Own Server Program

In order to use the Object Services Server, you must register 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 5-12.

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

```
Implementation alias:     MYSERVER
   ImplDef class name:    ImplementationDef
   Server secure:         No
   Server class:          somOS::Server
   Protocol information:  SOMD_TCPIP
```

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.

After the implementation definition of your new server program is complete, the server must be initialized (see “Initializing the Server” on page 5-12). If you then wish the server to be automatically started whenever the class it supports is utilized, it must be defined to the Workload Manager (WLM). See the section on “Defining Servers to Workload Manager (WLM)”" in the *OS/390 SOMobjects Configuration and Administration Guide*.

Chapter 6. Persistent Object Service

Persistent objects are objects that live beyond the execution time of an application in which they are referenced. Persistence is usually achieved by storing the object, or at least the pertinent information about the object, in some sort of persistent store (relational database, flat file and so forth).

This chapter and the examples it includes demonstrate the use of the Persistent Object Service (also known as POSSOM) for SOMobjects. POSSOM is the IBM implementation of the CosPersistence interface defined by OMG, along with some extensions.

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

This chapter has two purposes:

- For a description of the classes to inherit from or methods to override to make your SOM objects persistent, see “Explicit Persistence Programming Model” on page 6-11 and “PDS Design Considerations” on page 6-19.
- For a description of how to extend the implementation to either use a new datastore or to perform some other behavior, see “POSSOM Architecture” on page 6-5.

The purpose of the SOMobjects Persistent Object Service is to allow an application to use persistent objects (PO) without having to program to the underlying datastore APIs. The framework is designed so that replacement objects can be registered with the framework to store the data in multiple different datastores. The framework component that takes care of interfacing to the underlying datastore interface is called the Persistent Data Service (PDS).

Two PDSs are supplied with the framework, one that stores the object state data in a VSAM keyed file (B-Tree), and another that stores each object in a POSIX flat file (a standard file that is created by **fopen** and written to by **fwrite** and referred to by its path name). Because of the extensible datastore interface, additional datastores can be plugged into the POSSOM Framework any time. POSSOM provides the ability and flexibility to enable different datastores as required.

POSSOM and OMG

This section describes OMG compliance as related to POSSOM, and how the POSSOM implementation differs from the OMG persistence architecture.

SOMobjects introduces the concept of OMG-compliant Object Services. SOMobjects provides interfaces for many of the OMG object services such as lifecycle, persistence, and so forth.

POSSOM is the SOMobjects OMG-compliant Persistent Object Service. POSSOM complies with "CORBAservices: Common Object Services Specification, OMG Document Number 95-3-31, Chapter 5, Persistent Object Service Specification". For

more information, contact the Object Management Group (OMG) at their home page:

`http://www.omg.org`

POSSOM provides implementations for all of the OMG-defined persistence interfaces. Any client code written to the OMG persistence interfaces works with SOMobjects and the POSSOM service. This means that any client code you write using POSSOM's OMG implementations is interoperable with any platform (IBM or non-IBM) that is compliant with the OMG persistence architecture.

Although client code is portable between platforms, the underlying datastore used to store persistent objects is not necessarily portable to other OMG-compliant platforms. The object can be stored on another platform if a Persistence Data Service (PDS) for the targeted persistent store is available on that other platform. In effect, the object ports to another platform if a PDS is available on the other platform. The POSSOM components that interface with the underlying datastore are the PDS and the protocol.

POSSOM provides support for persistent objects (objects with both persistent object references and persistent state data) in conjunction with other related Object Services. In particular, POSSOM uses the services of Externalization, Life Cycle, and the OS Server. Persistence uses Life Cycle to locate a factory (via the find_factory method) which can be used to create PDS objects.

The Externalization Service describes the state data of an object. When using one of the IBM-supplied PDSs, the object developer provides the methods that will put the state data of the object into a stream and retrieve the state data of an object from a stream. This Streamable protocol is part of the OMG Externalization Service (see “Streamable Protocol” on page 6-13). The Externalization Service that is provided in SOMobjects 3.0 is a subset of the full OMG service. See Chapter 1, “Externalization Service” on page 1-1 for more details on the Externalization Service.

The OS Server is responsible for instituting persistent object references and managing object metastate on behalf of the SOMobjects Object Services. POSSOM creates persistent object references for you by sub-classing from **somOS::ServiceBasePRef**. This means that all persistent objects sub-classed from the POSSOM framework are OS Server managed objects and have persistent object references. See “Persistent Object References” on page 6-12 in this chapter for more information. See also Chapter 5, “Object Services Server” on page 5-1 for details on the OS Server.

POSSOM Implementation of OMG Interfaces

POSSOM provides some implementation of the OMG interfaces that are unique. In some cases, the OMG specification leaves the details to the implementor, so the SOMobjects implementation is an interpretation of how the interface should operate. The following sections describe any SOMobjects-unique implementation of the OMG persistence interfaces.

OMG Connect and Disconnect Semantics

OMG defines a connection as the mechanism that "establishes a close relationship between the Persistent Object and its datastore where the two data representations can be viewed as one for the duration of the connection." The connect semantic was defined to couple memory to disk for single-level store types of datastores (datastores that have a tight coupling of memory to disk so that memory and disk are virtually one and the same). One of the problems in trying to implement **connect** as a mapping between memory and disk is that the **connect** is part of the Persistent Object interface, so it is invoked on already instantiated objects. Single-level store datastores usually do the memory to disk mapping at the time of instantiation, thus making the implementation of the **connect** semantic to mirror single-level store semantics very difficult.

Because of the previously mentioned implementation problems, POSSOM provides a default implementation of the OMG **connect** interface that returns a **NO_IMPLEMENT** exception. Client code could choose to override the **connect** interface as a passthrough to the **restore** method or any other desired implementation.

Similarly, OMG describes the disconnect semantic as follows: "When the connection is ended, the data is the same in the PO and the datastore, and the relationship between them no longer exists." For POSSOM a default implementation of the OMG **disconnect** interface is provided that returns a **NO_IMPLEMENT** exception. Client code could choose to override the **disconnect** interface as a passthrough to the **store** method or any other desired implementation.

OMG delete interface

Because C++ uses "delete" as a keyword, the SOMobjects implementation of the OMG delete interface uses the capital 'D' (**Delete**) as the method name.

Contrasting COS and SOM interfaces

There are interfaces provided by POSSOM that are prefaced by **Cos** and others by **som** (such as **CosPersistencePO::PO** and **somPersistencePO::PO**). The **Cos** modules are abstract base classes corresponding to the OMG specification names. The **som** modules inherit from the **Cos** modules and are the IBM-supplied implementations of, and in some cases extensions to, the OMG specification.

General Concepts

POSSOM can be used by three distinct types of developers:

- Application Developers who want to use persistent objects in their SOM application.

  Application developers should focus on sections “POSSOM Architecture” on page 6-5, “Explicit Persistence Programming Model” on page 6-11, and “Scenarios for Creating and Using Persistent Objects” on page 6-19.

- Object Implementers who design and develop persistent objects.

  Object implementers should focus on the same sections as the application developers, listed above.

- Datastore providers who want to extend POSSOM to interface to their particular datastore.

Datastore providers should focus on sections “POSSOM Architecture” on page 6-5, “IBM-Supplied Datastores” on page 6-12, and “Providing a new Datastore (PDS)” on page 6-16.

Datastore Complexity

Before POSSOM, the developer had to deal directly with the complexity of the underlying datastore as depicted in Figure 6-1.

The purpose of POSSOM is to eliminate datastore-dependent code. as shown in Figure 6-2 on page 6-5. POSSOM shelters the object developer from programming to datastore specific APIs. Objects can be stored persistently in a variety of datastores as long as the object is designed to the POSSOM interfaces. The POSSOM framework relieves the object developer from having to deal directly with the complexities of the datastore and provides the flexibility of interfacing simultaneously to multiple underlying datastores through the PDS. Client code does contain datastore dependent information such as the Persistence Identifier (PID). However, the datastore specific code is encapsulated in the PID and PDS and only dealt with minimally by the client code.

In addition, POSSOM is an open architecture that provides you with the ability to "plug" additional datastores of your choosing into the framework. Once a new datastore has been added, existing and new SOM objects can be stored persistently to that datastore without the need to change the object code. POSSOM can provide transparent access to an unlimited number of underlying datastores simultaneously. Details regarding the IBM implementation of B-Tree, and POSIX datastores are given in “IBM-Supplied Datastores” on page 6-12.

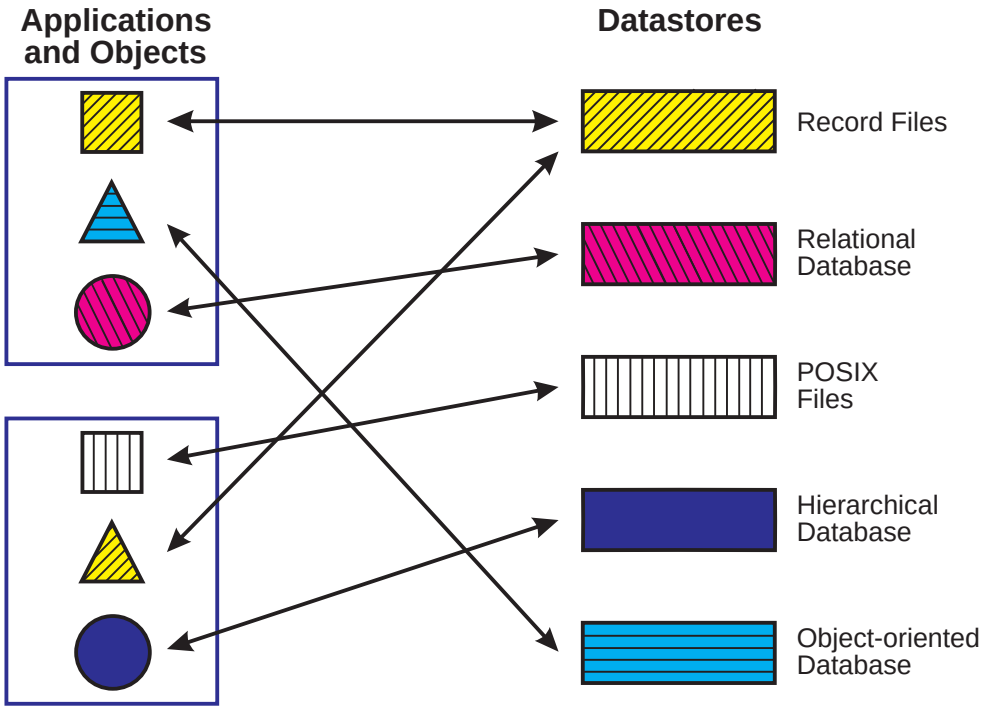

Figure 6-1. The Complex World Faced by Object Application Developer (before POSSOM)

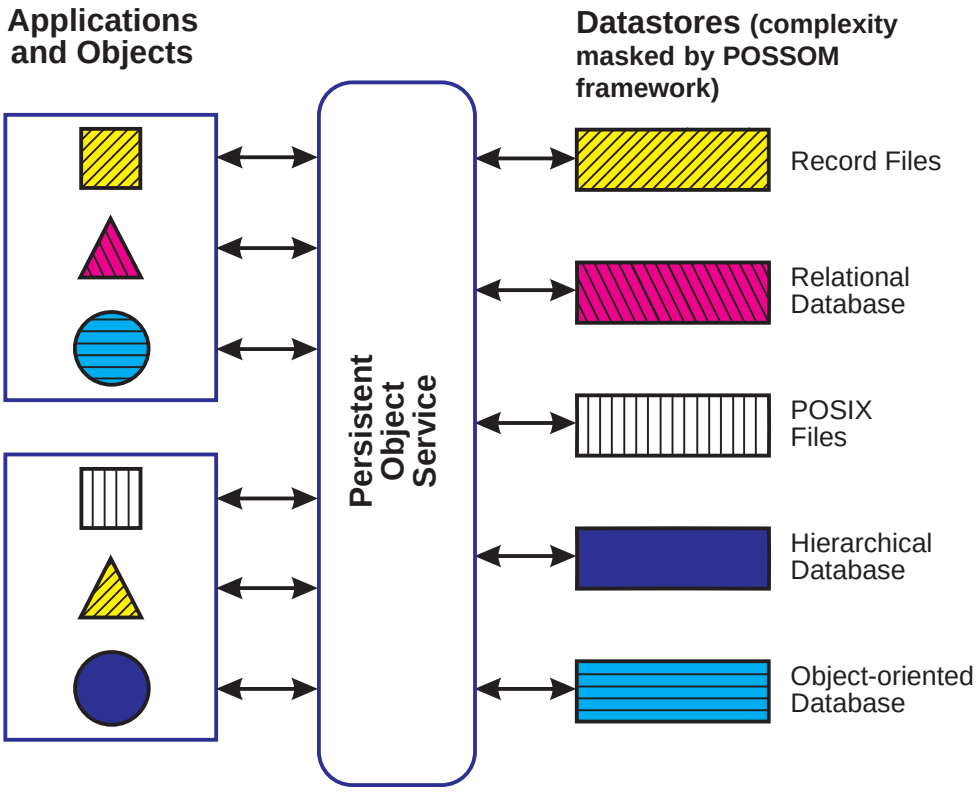

Figure 6-2. The Persistent Object Service (POSSOM) Vision

POSSOM Architecture

This section gives you an overview of the pieces of the POSSOM framework that you need to become familiar with to make your SOM objects persistent.

Conceptual View

Figure 6-3 on page 6-6 shows the POSSOM architecture from a conceptual view, with the POSSOM framework itself (labeled **Persistent Object Framework** in the figure) and the two pluggable interfaces:

1. The client-level interface describes how clients request persistent operations, such as **store**, **restore** or **Delete**, on their SOM objects (regardless of the underlying datastore). This interface is labeled in the figure as **Client Code.**
2. The datastore interface describes how a datastore provider plugs a new datastore into the POSSOM framework. This interface is labeled in the figure as the **PersistentDataService (PDS)**.

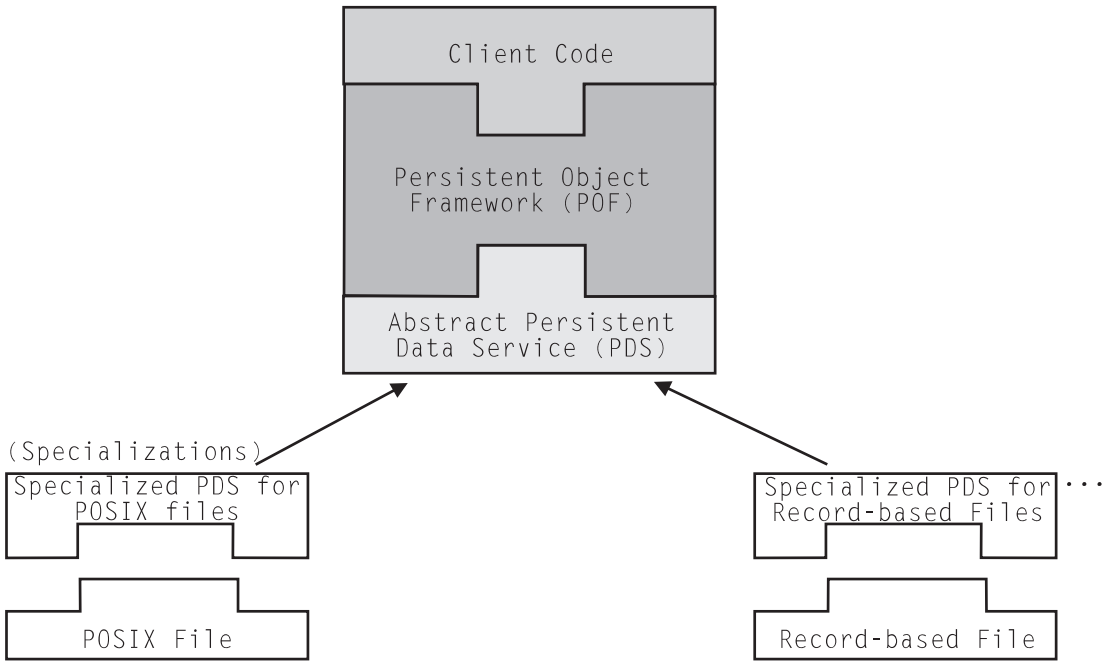

Figure 6-3. POSSOM Architecture Conceptual View

Framework Components

Figure 6-4 on page 6-7 shows the POSSOM framework components in more detail. The client and datastore pluggable interfaces are shown, as well as the components of the persistent object framework. Each framework component is described in the following sections.

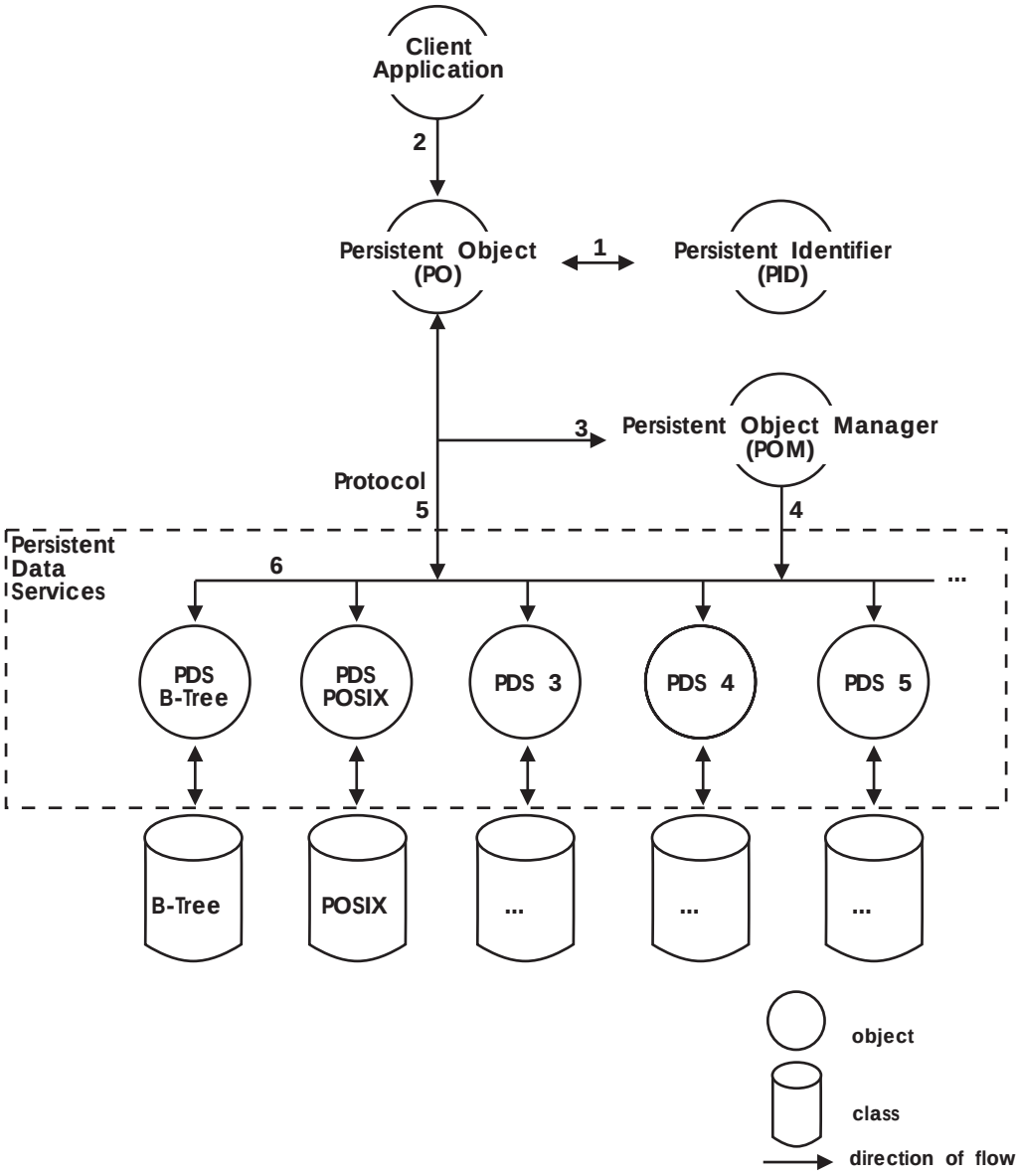

Figure 6-4. POSSOM Framework Architecture Components

Persistent Object (PO)

The POSSOM Persistent Object framework component is the interface that the client uses to make their objects persistent. The explicit persistence (PO) object interface is used by the client to control when objects are stored or restored. The client uses this interface depending on how much control it wants to have over when the objects are stored or restored. See “Explicit Persistence Programming Model” on page 6-11 for a complete description of this programming model.

A persistent object sub-classed from the persistent object interface provided by POSSOM (**PO**) is always an OS Server object. This means that you must register this object with the appropriate OS Server. This also means that your persistent object will have both persistent state data (managed by POSSOM) and persistent object references (managed by the OS Server).

Persistence Identifier (PID)

A PID is the information that locates the persistent state data of an object. It is used by the POM to locate a PDS that understands how to store the data to the proper datastore type. It is also used by the PDS to resolve where in that datastore type the object is to be placed.

For example, if each object is stored in its own POSIX file, the PID contains a flag indicating a POSIX datastore, a path name and a file name. If the multiple objects are stored in a VSAM dataset, then the PID contains a flag indicating a VSAM (B-Tree) datastore, a fully qualified dataset name, and a key used within the dataset to locate the state data of a particular object.

The **set_p** and **get_p** methods are used to set and retrieve the PID values. The PID always uses the Streamable protocol to internalize and externalize its data.

The PID is a DSOM object and must be registered with the appropriate server (which must be a DSOM Server, but can also be an OS Server).

Persistence Object Manager

The major function of the **Persistence Object Manager (POM)** is to route the persistent object (**PO**) to an appropriate **PDS**. The **POM** is the POSSOM component that ensures datastore independence. The POM is a DSOM object and must be registered with the appropriate server.

The **POM** function is depicted in Figure 6-5 on page 6-9. The routing is based upon the protocol supported by the object and the datastore type stored in the **PID**. Once routing has been accomplished, the appropriate **PDS** communicates directly with the persistent object (PO) to transmit the data.

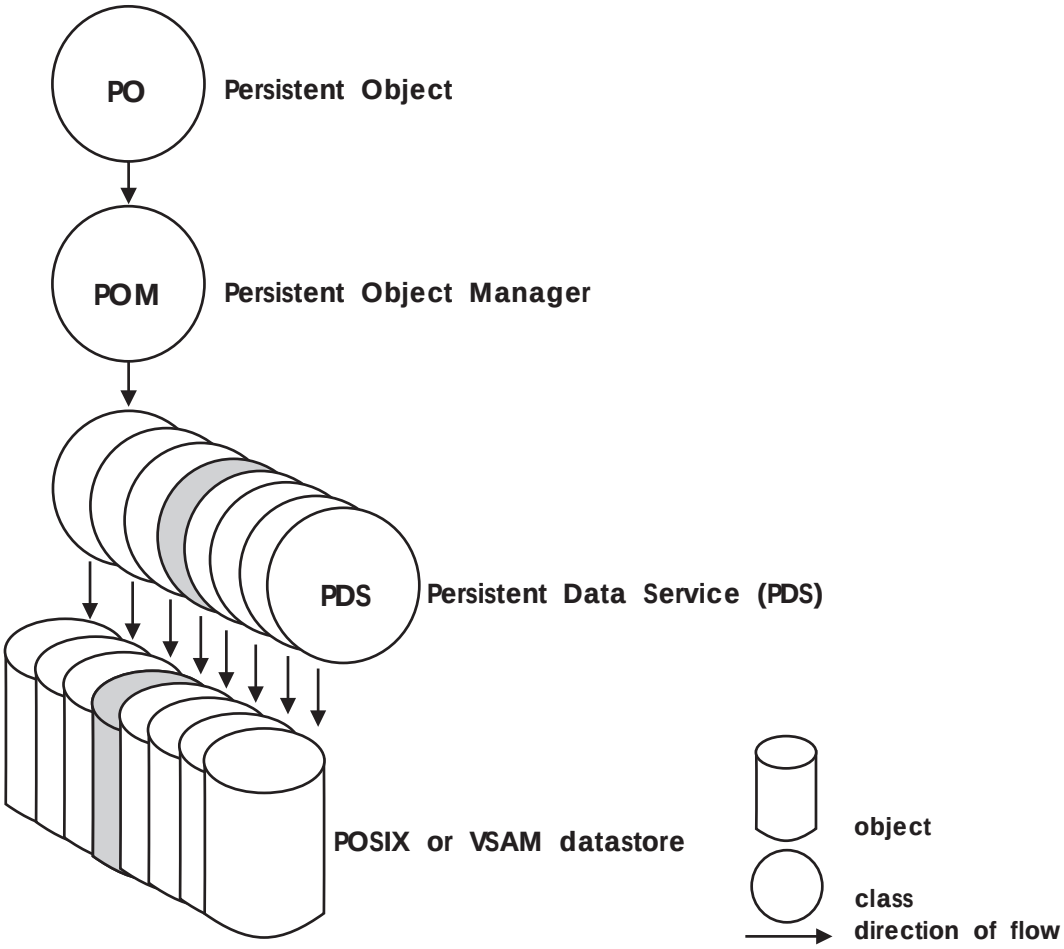

Figure 6-5. Routing Function of POM

When a **POM** is instantiated, it reads in a text file, named **SOMMVS.SGOSMISC(GOSPMDAT)**, that contains definitions used to map between protocols, datastores, and PDSs. The file contains one line per mapping, with lines in the following format:

```
Protocol        Datastore_Type      PDS_Class
```

The path of this file is kept in the gosenv.ini file under POS_POMDATA. See“SOMMVS.SGOSMISC(GOSPMDAT) Dataset” on page 6-37 for details about the format of the text file contained in this dataset.

When the POM is first instantiated, this file is read and stored in an internal POM table. POM uses this table to determine which PDS can handle a given protocol for a given datastore. When the POM is passed a persistence operation, it searches the table from top to bottom for a match based upon the **PO** and the information associated with the **PID**, using the first match found in the table.

The following algorithm is run for the **POM** table search:

```
found = FALSE;
for (n=0; n<NumberOfMappings; n++) {
     mapping = getMapping(n);
     protocol = getProtocol(mapping);
     datastore = GetDatastore(mapping);
     PDS = getPDS(mapping);
     if ((object is derived from protocol) &&
     (object is being stored to datastore)) {
     found = TRUE;
     break;
     }
     if (!found) return exception;
}
```

Protocol

For a **PDS** to get data from an object, it must understand some mechanism the object has provided to make its data available. There must be a similar mechanism the object has provided to receive new data.The mechanism for getting data in and out of the object is called the *protocol*. The IBM-supplied PDS (B-Tree, POSIX) supports a stream-based protocol called the **Streamable** protocol, that is based upon the OMG Externalization Service. See “Streamable Protocol” on page 6-13 for more details.

Persistent Data Service (PDS)

One of the important functions of POSSOM is to store and restore an objects data in a datastore independent manner. To do this, there must be a component of the POSSOM framework that interacts with the object to get data from the object and put data into the underlying datastore. The POSSOM framework component that accomplishes this is the **Persistent Data Service (PDS)**.

Two specialized **PDS** implementations ship with POSSOM for storing objects into a VSAM key-sequenced data set, and a POSIX flat file. These are referred to as the "IBM-supplied PDSs" for purposes of this documentation. See “IBM-Supplied Datastores” on page 6-12 for more information.

The POSSOM PDS interface also allows additional datastore PDSs to be written and "plugged" into the framework at any time.

The PDS is a DSOM object and must be registered with the appropriate server.

Example store operation

Referring to Figure 6-4 on page 6-7, a **store** operation (a **restore** or **Delete** operation would be very similar) flows through the various persistence architecture components as follows:

1. The client code (application) sets up information in the **PID** regarding where to store the object's state data.

   The PID contains datastore specific information. For example, let's assume that the object is being stored in a POSIX file. In this case, the **PID** contains the datastore type (IBM_POSIX) and the path name of the POSIX file in which the object is being stored.

2. The object to be made persistent must inherit from the PO interface (somPersistencePO::PO) and a protocol interface (in the case of the IBM- supplied datastores, the protocol interface is the **somStream::Streamable** interface). In general, the PO interface provides the **store, restore** and **Delete**

methods. These methods can be overridden if desired, however a default implementation is provided by the framework.

If, for example, you needed to provide a function to copy the object's state data from a temporary buffer prior to the store operation, this could be done by overriding the **store** method and providing your own implementation. The default **store** method is sufficient for most cases, however.

Streamable provides the interface for externalizing the object's state data to be stored into one of the IBM-supplied datastores (VSAM (B-Tree) or POSIX). This externalization interface is referred to as the *protocol*. Let's assume that you are using the explicit persistence programming model (see "Explicit Persistence Programming Model"). For the store operation, the client tells the object to store itself by invoking the **store** method inherited from **PO**.

3. **PO** routes the **store** request to the **Persistent Object Manager (POM)**. If a **PID** was passed on the **store** method, **PO** temporarily uses the new **PID** value for that **store** operation. The object's **PID** (the **PID** is an attribute of **PO**) is not updated, however.
4. The **POM** finds the appropriate **Persistent Data Service (PDS)** based upon the information in the text file read by the POM, (see "SOMMVS.SGOSMISC(GOSPMDAT) Dataset" on page 6-37) and the **PO**, that includes the **PID**. The **POM** routes the **store** request to the **PDS**. Once the **POM** has routed the **store** operation to the **PDS**, the **POM** is no longer involved in the actual storing of the data to the datastore.
5. The **PDS** retrieves the state data that is to be stored persistently from the object. The **PDS** does this through the *protocol* interface of the **PO**.

   For the IBM-supplied PDS implementations (VSAM (B-Tree) and POSIX), the protocol is always a stream protocol. Object implementation can use interfaces (such as **write_int, write_float**) provided by **somStream** to build-up the object's data into the stream.
6. Once the **PDS** gets the data through the protocol, the **PDS** stores the data into the datastore using the datastore specific APIs. A datastore provider can supply a **PDS** for their particular datastore, or one of the IBM-supplied PDSs (VSAM (B-Tree) or POSIX) shipped with SOMobjects can be used.

Explicit Persistence Programming Model

POSSOM provides one Programming Model, the explicit persistence model (PO). This is used by the client to to *explicitly* store, restore, or delete the object's state data. The interface for the explicit persistence model provides three methods, **store**, **restore** and **delete**, that manipulate the object into and out of the persistent storage upon demand.

As the client programmer you can have your object support the explicit persistence model by having it inherit from the PO base class.

Explicit persistence is so named because clients take explicit control over when objects are stored and restored to and from the disk. This is the model of store and restore with which most people are familiar. For example, in word processors, you explicitly say when to save your document. With relational databases, you write explicit SQL update commands to move data from variables in memory to rows in a table.

When an object supports explicit persistence, it inherits from the **somPersistencePO::PO (PO) interface**. The **PO** interface provides **store, restore**, and **Delete** methods.

See “Steps to Creating an Explicit Persistent Object” on page 6-24 for programming examples for creating and using an explicit persistent object.

Benefits/Drawbacks of Explicit Model

The explicit persistence model provides flexibility for the client program. The client program can control when the object is stored and restored using the interfaces provided.

While using the explicit model, when the parent object is restored, the client code must determine when and how to restore any children (contained) objects. If the client code chooses to restore all of the children (contained) objects when the parent object is restored, performance could be impacted, especially if only a few of the children objects actually needed to be fully restored.

The explicit persistent model allows a standard way to read in a PO object from one datastore and write it out to another. It also provides a standard way to read in the data from one file and write it out to another. This capability is provided by using the set_p method on PO objects. Objects implemented as explicit persistent objects can be read from a stream file and, in turn, be written to another file or database.

Persistent Object References

A typical CORBA (Common Object Request Broker Architecture) object reference is valid for the life of an object. This means that if the process where the object resided was terminated, then all references to that object are no longer valid.

A PID is valid for the life of the state data. This means that if the process where the persistent object resided is terminated, then that object can still be referenced. The first time that object is referenced after the process is reactivated, the object is recreated and its state data restored from the underlying datastore.

POSSOM uses the OS Server to make a CORBA object reference valid for the life of the state data. A persistent object sub-classed from the POSSOM interface (PO) inherits from **somOS::ServiceBasePRef**. This means that not only is the object state data persistent, but the object reference is also persistent.

See also “Guidelines for Creation and Destruction of Persistent Objects” on page 6-20 which describes and explains the programming ramifications of having persistent object references.

IBM-Supplied Datastores

The Persistent Object Service for SOM ships with two specialized PersistentDataService (PDS) implementations for storing objects into VSAM (B-Tree) and POSIX datastores. These IBM-supplied implementations can be used as-is to interface to a VSAM key-sequenced dataset (KSDS) or a POSIX flat file.

There are some considerations for which datastore to use, depending on your implementation requirements. These considerations are outlined in the overview section for each PDS.

Multi-user

You may want to have multiple users accessing your persistent objects. A common scenario is having multiple client programs accessing persistent objects in one or many server processes. These client programs may store and restore the persistent objects many times. If the underlying datastore does not provide locking, the data may not be consistent with each store operation. In other words, one client may have stored into the datastore and overwrote another clients changes.

The VSAM (B-Tree) PDS is the only IBM-supplied PDS that provides simultaneous multi-user access.

The VSAM datasets are used across the sysplex, therefore the sombt stanza in the global configuration file should reflect that. See the section “The Configuration File and Environment Variables” in *OS/390 SOMobjects Configuration and Administration Guide* for a syntax example of how to set up your configuration file with the sombt stanza.

Security

The IBM-supplied PDS implementations rely on the underlying file system for security. This means that the client program is responsible for storing the persistent objects in a datastore that has the proper security settings.

Additionally, there is server identity security. This provides security at the server level and the method level. See the description of how to configure security for SOMobjects in *OS/390 SOMobjects Configuration and Administration Guide* for more information.

Streamable Protocol

Both IBM-supplied PDS implementations (VSAM (B-Tree) and POSIX) use a stream protocol to externalize the object's data to the datastore. In the stream-based protocol, the PDS assumes the object supports the SOMobjects Externalization Service's **externalize_to_stream** and an **internalize_from_stream** operation (for simplicity, these will be referred to as **externalize** and **internalize** in the following discussion).

Some definitions to be aware of in the following discussion:

- **StreamIO** - An interface of the Externalization Service that writes and reads object state data to the appropriately converted external form.
- **Streamable** - An interface of the Externalization Service used to get state data from the object for externalization, and put state data into the object for internalization. Objects which support these operations are said to be streamable.
- **Stream** - The client interface of the Externalization Service for externalizing and internalizing objects. The **Stream** object uses the **StreamIO** and **Streamable** interfaces. It manages the creation of **Streamable** objects and restores references between **Streamable** objects during internalization.

Stream protocols typically require implementation code in the object. The current stream implementation for the IBM-supplied PDSs requires that the elements of the stream be retrieved in the same order as they were written into the stream because it is not a self- describing stream.

The OMG Object Externalization Service, or streaming, provides a way to write a single pair of internalize/externalize operations. For example, suppose you have the following objects as illustrated in Figure 6-6.

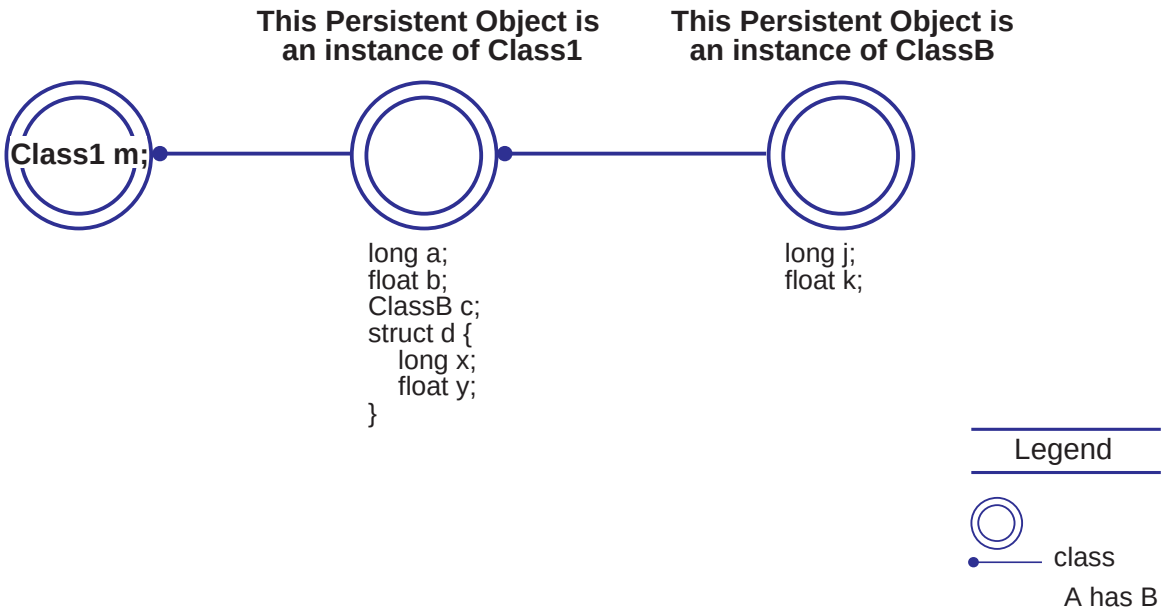

Figure 6-6. Streamable Interface

Each of these classes would inherit a standard **Streamable** interface which includes **internalize** (move data from a **StreamIO** object into the **Streamable** object) and **externalize** (move data from the **Streamable** object into the **StreamIO** object) operations.

The **externalize** method for Class1, which puts the Persistent Object's data in a **StreamIO** object, would be:

```
Class1::externalize_to_stream(Stream s)
{
     s->write_int(a); // put po's member named "a" in Stream
     s->write_float(b);
     s->write_object(c);
     s->write_int(d.x);
     s>write_float(d.y);
}
```

The **internalize** method for Class1, which fills the Persistent Object's data from a **StreamIO** object, would be the same as **externalize**, except that s->write_* would be replaced by s->read_*.

It is up to the **StreamIO** (not the streamable Persistent Object) class implementor to determine how these read and write operators are implemented. This means that the Persistent Object's **internalize/externalize** methods are identical at the binary level and will not need to be modified for use with different stream implementations.

The following example illustrates the flexibility of this design. The **StreamIO** operation s->write_object(c) could be implemented to do either of the following cases:

```
a. add object_to_string(c) to the byte stream; //add object ref

b. add object_to_string(c) to the byte stream;
   c->externalize_to_stream(s); // put c's data in same stream
```

Case (a) saves the object id for object c. This is useful for when object c is stored using object s's PID (that is, s's and c's persistent data are stored together). Case (b) saves both the object id and the state of object c in the same stream. This is useful for when object c has its own PID. In either case, the **internalize/externalize** methods of the streamable Persistent Object are identical at the binary level.

This approach also allows the actual byte stream format up to be up to the **StreamIO** implementation and independent of the streamable Persistent Object's binary code.

Adding new state data to a Persistent Object

A persistent object purchased in the component industry might need to be subclassed to fit the buyer's specific application. Sometimes this requires adding a new persistent state to the persistent object. For example, a subclass Class2 of the above Class1 (Class2: Class1) might need an additional member data:

```
float e;
```

The **internalize/externalize** methods can be overridden easily by first invoking the parent Class1's **externalize** (to get the data defined within Class1) and then adding the new member data to the stream:

```
Class2::externalize_to_stream(Stream s)
  {
     this->Class1::externalize_to_stream(s);
     s->write_float(e);
  }
```

The implementor of Class2 did not have to know anything about the implementation of Class1 to do this.

VSAM (B-Tree) PDS Overview

The VSAM (B-Tree) interface is used to **store**, **restore** and **Delete** multiple persistent objects to and from a single VSAM dataset. If the file has not already been created, the PDS creates it. VSAM is used as the underlying datastore for the B-tree datastore implementation.

Each persistent object within the VSAM dataset is uniquely identified by an object key. The object key is a string of any length up to a maximum of 255. The value of each object key is determined by the user of the VSAM (B-Tree) interface.

The persistent object data that is associated with a particular object key can have a maximum length of 32,501. There is no requirement that all the object keys or all

the persistent object data be of the same size within a single VSAM dataset. This allows different types of persistent objects to be stored within the same dataset as long as key collisions do not occur.

Considerations for VSAM (B-Tree) Datastore

The VSAM (B-Tree) datastore PDS implementation is similar to POSIX in that object data is stored in a file. The difference is that VSAM (B-Tree) uses a *VSAM key-sequenced* dataset so that there can be multiple objects stored per file. This saves on having a file created for every persistent object and on having the file opened and closed on every persistent operation (**store**, **restore**).

POSIX PDS Overview

The POSIX interface is used to **store**, **restore** and **Delete** a single persistent object to and from a single POSIX datastore file.

The POSIX PDS is so named because it uses the standard POSIX file I/O commands, such as **fopen()** and **fclose()**. Standard system file naming conventions should be followed. If the file has not already been created, the POSIX PDS creates it.

The POSIX datastore path name always follows the forward slash convention ("/") even though the conventions may be different for that platform. If a path name is not specified, the current directory is used. If the file name is too long (per the system rules for length of filename), the file is not created. Using the double slash convention ("//"), a persistent objects state data may even be stored in a standard OS/390 sequential or partitioned dataset.

The **Delete** method deletes the file and the persistent object state data stored in that file.

Considerations for POSIX datastore

The POSIX datastore PDS implementation is basically a flat file which holds the object's persistent data. A limitation of the IBM-supplied POSIX PDS implementation is that each object is put in its own separate file. If you need to manipulate hundreds of persistent objects, the POSIX implementation creates hundreds of files.

In addition, the file is opened and closed for every persistence operation (**store**, **restore**). Be aware of the overhead of opening and closing the file for every persistent operation and the performance implications before using the IBM-supplied POSIX PDS.

Providing a new Datastore (PDS)

Because POSSOM is written as a framework, it allows for extensible datastore implementations. In other words, additional datastores can be plugged into the POSSOM framework at any time by implementing a new **Persistent Data Service (PDS)**.

Implementing a new Datastore (PDS): The How To's

A new datastore PDS can plug into the Persistence Object (POSSOM) framework by executing the following steps:

1. Ensure that the new datastore's PDS is included in the text file used by the POM. Refer to “SOMMVS.SGOSMISC(GOSPMDAT) Dataset” on page 6-37 for details about the **SOMMVS.SGOSMISC(GOSPMDAT)** dataset.
2. Derive a new PID class from the generic IBM **PID_DS** class. You create a new PID class by inheriting from **somPersistencePID::PID_DS.**
3. Override the following inherited **PID_DS** methods in the new PID class. You should be aware that the PID class must use the streams provided by the Object Externalization Service for externalizing and internalizing its data, thus the need to override the following methods.

   • **updatePIDStream()**

   This method must be overridden by any class derived, directly or indirectly, from **PID_DS**. The overridden version of this method should do the following:

   a. Call this method on the parent class.

   b. Get the stream returned by **_get_stream**.

   c. Place any new attributes for this PID derivation in the stream. Your new PID class should contain attributes which define any datastore specific information that is required by your new PDS. Examples of datastore specific attributes in the IBM-supplied PDSs are datastore name and key in the VSAM (B-Tree) PDS. Any PID attributes which are placed in the stream will become part of the metastate which is saved by the OS Server as part of managing persistent object references. If you do not need a particular datastore attribute saved as metastate, then do not place it in the stream.

   • **readFromPIDStream()**

   This is the inverse of updatePIDStream. This method also must be overridden by any class derived, directly or indirectly, from **PID_DS**. The overridden version of this method should do the following:

   a. Call this method on the parent class.

   b. Get the stream returned by **_get_stream**.

   c. Retrieve any new attributes for this derivation from the stream (retrieve the attributes in the same order as they were placed in the stream by the **updatePIDStream**).

   • **somDefaultInit()**

   This method must be overridden if any of the following attributes need to be initialized by the new PID class:

   a. **datastore_type** is an attribute that defines the datastore type. For example, this attribute is set to IBM_POSIX for the IBM-supplied POSIX datastore. As a PDS developer, you can choose to have the client set this, so you can optionally choose to initialize it here. There are **get** and **set** methods for this attribute.

   b. Initialize the "stream type" attributes inherited from the **PID_DS** class. **PID_DS** initializes the "stream type" attributes to null. These only need to

be initialized if your PDS is using the stream protocol provided by the Object Externalization Service. The "stream type" attributes are:

stream_type: setting this attribute to the full class name of a Stream allows you to pass information to the PDS about the type of Stream to use. It is up to the PDS implementor to decide whether or not to make use of this information. There are **get** and **set** methods for this attribute.

streamio_type: setting this attribute to the full class name of a StreamIO allows the user to pass information to the PDS about the type of Stream IO to use. It is up to the PDS implementor to decide whether or not to make use of this information. There are **get** and **set** methods for this attribute.

4. Derive new PDS class from the generic IBM PDS class.

   You create a new PDS by inheriting from **somPersistencePDS::PDS**. The inheritance relationship for **PID** and **PDS** is shown in Figure 6-7 on page 6-19 for the POSIX PDS.

5. Override any of the following methods in your new PDS that you want it to support (for example, a PDS might not support connect and disconnect).

   connect()
   disconnect()
   store()
   restore()
   Delete()

6. When the POM creates a PDS, it calls the **initialize()** method (inherited from PDS) to pass the PDS a StreamFactory. It is not necessary to override the **initialize** method if you want the PDS to decide which process to create the stream in (PDS process versus POM process) or if you are not using the stream protocol to externalize your data.

   If you want the stream creation to be local to the PDS, the GOSENV.INI file should have the POSSOM "stream creation" set to LOCAL (for example POSIX_StreamCreation=LOCAL). If you want to have the stream creation to be in the same process as the POM, the "stream creation" should be set to NONLOCAL.

   If you want to continue to use the StreamFactory beyond the **initialize** method call, the **initialize** needs to call the **duplicate** method.

7. Register your new PDS with the appropriate server (e.g. using the regimpl command).

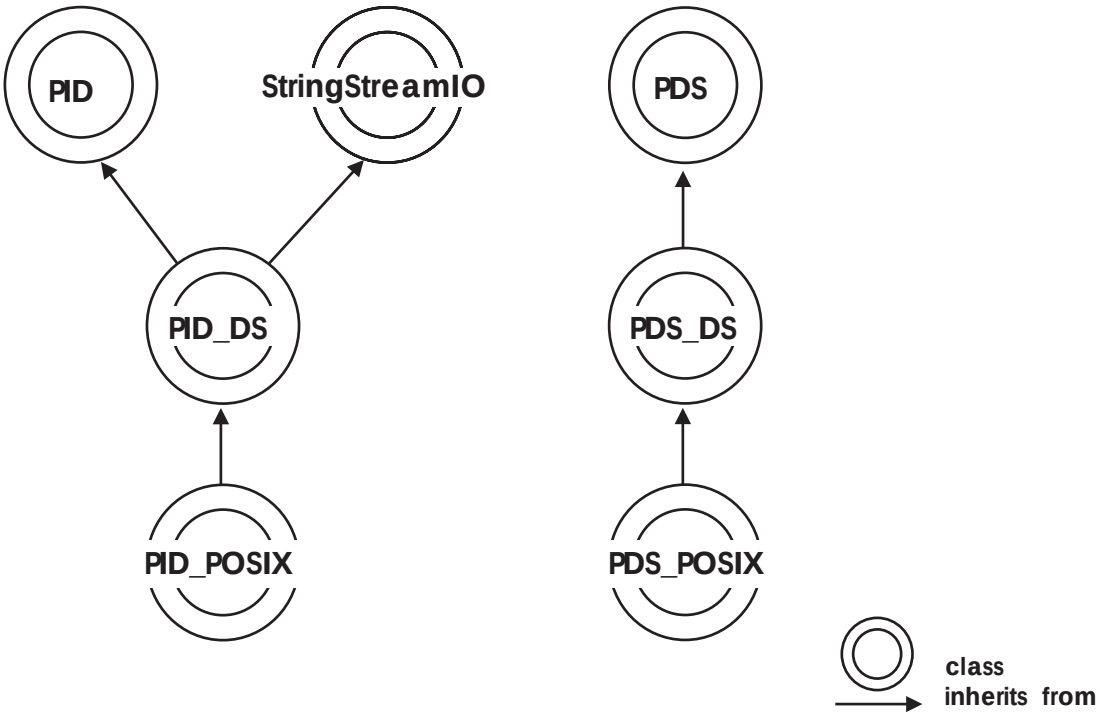

Figure 6-7. PID and PDS: Inheritance Relationships POSIX example.

PDS Design Considerations

This section describes things you should consider when designing a new or replacement PDS.

Opening and Closing a Datastore: Opening a datastore is typically the most time-consuming part of storing data. Therefore, careful attention must be paid to its optimization.

POSSOM consists of a series of routing mechanisms, ultimately ending up at the Persistent Data Service (PDS) that knows how to interact with the object (to get and put object data) and the datastore (to get and put datastore data). When a PDS is asked to perform any I/O operation, it is always passed a Persistent ID (PID), which identifies the actual instance of the datastore necessary at the time that the I/O is initiated. Since it is likely that further I/O requests will be made to the same datastore, the PDS should leave the datastore in an open state. This allows further I/O requests to proceed without requiring the open overhead. When the PDS is formally brought down, it should close any datastores that it currently has opened.

Scenarios for Creating and Using Persistent Objects

This section describes the steps necessary to create persistent objects using the explicit programming model. A step-by-step description of which interfaces to inherit from, which methods to override, and so forth is provided.

How this section is organized

The section below describes how an object implementor creates explicit persistent objects and how an application developer writes client code (an application) to use explicit persistent objects.

This section is organized as follows:

- Guidelines for achieving datastore independence.
- Guidelines for creation and destruction of persistent objects.
- Overview of general steps (an outline) to creating and using persistent objects.
- Specific steps to creating and using persistent objects, where:
  - Each of the general steps are expanded upon for the programming model (explicit persistence).
  - Specific, detailed examples are provided in each section. The IBM-supplied PDSs (VSAM (B-Tree), and POSIX) are used in the examples to show differences in programming to the various datastore interfaces.

Guidelines for Achieving Datastore Independence

Application developers will want to ensure that their client programs are datastore independent, as much as possible. To do this, application developers should partition application code into datastore-dependent and datastore-independent parts. The only functionality that should be in the datastore-dependent part is the creation and setting of PIDs. A PID can be input to the datastore-independent part as a parameter either as an object reference to the PID or as a PID string (from which a PID can be created through the **PIDFactory**).

Object implementers will also want to ensure that their objects are independent of the datastore as much as possible. To do this, object implementers should specify at object creation time whether an object is transient or persistent. This is done by subclassing from PO.

Guidelines for Creation and Destruction of Persistent Objects

Because the POSSOM persistent object interfaces (PO) inherit from **somOS::ServiceBasePRef** to provide persistent object references, the object implementor must be aware of the OS Server initializers and uninitializers and how to implement them appropriately. Application developers must know when to use the methods appropriately. This section clarifies object creation and destruction for the object implementor as well as the application developer.

Lifecycle of an Object

In the following descriptions, a concept of object lifecycle and ownership is introduced. An object referred to as "owned" is one that your object is responsible for the lifecycle of (creating and destroying). Any object that you create and destroy is "owned" by your application. Your object may also reference other objects. If your object is responsible for creating and destroying those objects, then you "own" those objects.

On the other hand, your object may contain references (pointers) to other objects that it is not responsible for creating or destroying. Some other application or process is responsible for the lifecycle of that object. This case is referred to as an object that is "not owned".

Considerations for Storing Contained Objects

Your application may contain many persistent objects, and these objects can have references to each other. At some point you may need to store all of the objects.

If the object is a PO object, storing the object can be solved in several ways, such as creating an "instance manager" which is responsible for managing the storing of all persistent objects which are currently instantiated.

Restore is not a problem. Since the OS Server manages all object references, when a **read_object** is invoked as part of the **internalize_from_stream**, the state data of that object is restored because the OS Server invokes **restore**.

Guidelines for Object Implementers

Object implementers sub-classing from PO interfaces should follow these guidelines:

- Override all of the following initializers and uninitializers and perform the following functions (as appropriate):
  - **somDefaultInit**
    - call to parents (see below for more description of parent calls because default emitted code may need to be altered)
  - **init_for_object_creation**
    - create any memory required by the object (such as strings, buffers, sequences)
    - establish any references that are required (to objects not owned)
    - initialize any attributes to default initial values
    - create any objects that are owned
    - leave object in a state that **internalize_from_stream** can use
    - call to parents (see below for more description of parent calls because default emitted code may need to be altered)
  - **init_for_object_reactivation**

    (this method is called only by OS Server, but you must override it)
    - create any memory required by the object (such as strings, buffers, sequences)
    - establish any references that are required (to objects not owned)
    - leave object in a state that **internalize_from_stream** can use
    - call to parents (see below for more description of parent calls because default emitted code may need to be altered)
  - Other initializers (required only if attributes need to be set to initial values)
    - create any memory required by the object (such as strings, buffers, sequences)
    - establish any references that are required (to objects not owned)
    - initialize any attributes to default initial values
    - call to parents
  - **internalize_from_stream, externalize_to_stream**
    - make the object whole (store/restore from persistent datastore).
    - expect that **init_for_object_reactivation** or **somDefaultInit** has happened
  - **uninit_for_object_destruction**

- destroy any objects that are owned
- call to parents

– **uninit_for_object_passivation**

- call to parents

– **somDestruct**

- release any references that are not owned
- free any memory owned by object (such as strings, buffers, sequences)

– **init_for_object_copy** and **uninit_for_object_move**

SOMobjects does not implement at this time. Thus, these methods will never be called. However, it is recommended that you supply them for whenever copy and move object is added in the future.

- If all of your object's parents inherit from **somOS::ServiceBase** (which PO does), then you do not need to modify any of the default emitted **init_for_object_creation, init_for_object_reactivation, init_for_object_copy** methods because the default emitted code has the correct parent calls.
- If one or more of your object's parents are non-ServiceBase parents (meaning they do not inherit from **somOS::ServiceBase** classes), then you need to copy some of the lines emitted by the SOM compiler from **somDefaultInit** into all three **init_for_object_creation, init_for_object_reactivation, init_for_object_copy** as follows:

  Copy these lines:

```
somInitCtrl globalCtrl;
mymodule_myclass_BeginInitializer_somDefaultInit;
mymodule_myclass_Init_parentclass1_somDefaultInit(somSelf, ctrl);
mymodule_myclass_Init_parentclass2_somDefaultInit(somSelf, ctrl):
```

  where, parentclass1 and parentclass2 are all non-ServiceBase parents.

  Repeat for more parents, as necessary. Make sure you keep the emitted parent calls to **init_for_object_creation** as well.

  You do not need to modify the emitted code for **somDefaultInit**. In fact, it should never be called. The only reason you must override it is so that you can copy the parent calls out of it into the **init_for_object_xxx** methods.

Guidelines for Application Developers

Application developers using POSSOM persistent objects need to use these initializers and uninitializers, depending upon the situation, as follows:

- Creating a new object
  – Locate the appropriate class object in the OS Server by using **somdGetClass**
  – If a PO object, invoke **somNewNoInit** on the class object and invoke **init_for_object_creation** to get the persistent reference, or
  – For PO objects, if you don't care which server your object is created on:

    Invoke **somdCreate** with a default initialization parameter of FALSE
    Invoke **init_for_object_creation** (to create a new object reference)

Invoke **release** on the old object reference (pointer created by **somdCreate**)

- Destroying a PO:
  - Invoke **uninit_for_object_destruction** (destroys persistent meta data managed by OS Server as part of the persistent object reference)
  - Invoke **Delete** (destroys the object's state data in the persistent datastore)
  - Invoke **somFree** (removes the object from memory and invalidates the persistent object reference) (**somFree** calls **somDestruct**)

    If the persistent state data for the object being destroyed is used for other things (such as legacy data in a relational database used by both procedural and object oriented applications), then the **Delete** step should be eliminated.
- Cleaning up object references (when an object reference is no longer needed):
  - Invoke **release** (if the object is still going to be used by another object)
  - Invoke **somFree** (if the object is not going to be used by another object)

See Chapter 5, “Object Services Server” on page 5-1 for more details on object creation and destruction. Also refer to the samples in the following sections for programming examples.

General Steps to Creating a Persistent Object

There are some general steps to creating a persistent object. If you are familiar with creating SOM or DSOM objects, these steps should look very familiar.

Parts that are highlighted (bold) in the following list require specific implementations for persistent objects and are explained in detail in the following sections and examples.

Note: Object implementers use steps 1-4 to create a persistent SOM object.

Application developers who want to develop their own objects use steps 1-10. Application developers who want to use pre-built objects, use steps 5-10.

In general, the steps to creating a persistent SOM object are:

1. Write the interface (IDL) for your class.
2. Compile the IDL using the SOM complier with the C or C++ emitter. Use the resulting C or C++ code file as the basis for the next step.
3. Write the C or C++ code for your class, using the file created in the previous step. This is where you need to complete code for any persistence-related overridden methods. The complier generates code stubs.
4. Compile your C or C++ class object code and package into a DLL file.
5. Write your main program (application that uses the SOM persistent object).
6. Compile and link your main program with the DLL created in step 4. Use the resulting load module file to invoke your application (client code).
7. Create/update your SOM IR file.
8. Create your implementation repository using regimpl. Since several of the POSSOM framework objects (PDS, PID, StreamIO) are DSOM or OS Server objects, you need to register them in the implementation repository, along with

any persistent objects (PO) that you create (remember, PO is always an OS Server object).

9. Define and manage your servers with Workload Manager (WLM). See *OS/390 SOMobjects Configuration and Administration Guide* for more information.
10. Create the text file that the POM will read. A default **SOMMVS.SGOSMISC(GOSPMDAT)** dataset is shipped with POSSOM. You only need to update the dataset if you want to use a PDS other than those supplied by IBM with SOMobjects. If you need to customize the dataset , you need to add to the default **SOMMVS.SGOSMISC(GOSPMDAT)** dataset shipped with POSSOM. See "SOMMVS.SGOSMISC(GOSPMDAT) Dataset" on page 6-37 for details.
11. Run your application. Once the above steps are complete, you should be ready to run your application which uses persistent objects.

Steps to Creating an Explicit Persistent Object

The following example is written in C++. The steps from "General Steps to Creating a Persistent Object" on page 6-23 have been expanded to explain what you need to do to create an explicit persistent object (PO). This example is shipped with SOMobjects in the samples directory, so you can compile and execute this example if you want to.

For the purpose of this example, let's assume that you want to create a class that defines an entry in a company's phone directory. The class is named **phoneEntry**. When instantiated, each object of type **phoneEntry** contains the employee's name, phone number, and office location. The objects of type **phoneEntry** are explicit persistent objects (PO) and they are stored in a POSIX flat file datastore.

Step 1 - Write IDL for phoneEntry Class

Sample code for the following steps is shown in Figure 6-8 on page 6-25. Steps unique to persistence are highlighted.

In the IDL for your **phoneEntry** class, the following things are unique for persistence:

1. Include **somestrm.idl** and **somppo.idl** files.

   These are the IDL for the **Streamable** and **PO** interfaces that the **phoneEntry** class inherits from.See lines 004 and 005 in Figure 6-8 on page 6-25.

2. Inherit from **somStream::Streamable** and **somPersistencePO::PO** interfaces.

   Because we are using one of the IBM-supplied PDSs (POSIX), we must inherit from **Streamable** to provide the definition of the *protocol* for our persistent object. The *protocol* defines how persistent data is put in and out of the object by the PDS. All IBM-supplied PDSs use a stream protocol.

3. In this example we inherit from **somStream::Streamable** for the **externalize_to_stream** and **internalize_from_stream** methods rather than **CosStream::Streamable**. Since **somStream::Streamable** is a **somOS::ServiceBase** class, both parent classes (**somStream::Streamable** and **somPersistencePO::PO**) are ServiceBase classes (see "Guidelines for Creation and Destruction of Persistent Objects" on page 6-20 for more information).

   See line 007 in Figure 6-8 on page 6-25.

When using an IBM-supplied PDS, you must override the **externalize_to_stream** and **internalize_from_stream** methods.

4. Override the **externalize_to_stream** and **internalize_from_stream** methods and the initializers and uninitializers. See lines 033 - 042 in Figure 6-8.

```
001 #ifndef phone_IDL
002 #define phone_IDL
003
004 #include <somppo.idl>
005 #include <somestrm.idl>
006
007 interface phoneEntry : somPersistencePO::PO, somStream::Streamable
008 // CLASS NAME: phoneEntry
009 //
010 // DESCRIPTION: Each phone entry will consist of the person's name,
011 //              phone number and office number.
012 {
013
014     // Attributes:
015     attribute string name;              // person's name
016     attribute string phone_number;      // person's phone number
017     attribute short office;             // person's office number
018
019 #ifdef __SOMIDL__
020     implementation {
021         memory_management = corba;
022         releaseorder : _get_name,         _set_name,
023                       _get_phone_number, _set_phone_number,
024                        _get_office,       _set_office;
025
026         functionprefix = MVS_phoneEntry_;
027
028         // Attribute Modifiers:
029         name:           noget, noset;
030         phone_number:   noget, noset;
031
032         // Overrides:
033         init_for_object_creation:       override;
034         init_for_object_reactivation:   override;
035         init_for_object_copy:           override;
036         somDestruct:                    override;
037         uninit_for_object_destruction:  override;
038         uninit_for_object_passivation:  override;
039         uninit_for_object_move:         override;
040
041         somDumpSelfInt:                 override;
042         override: externalize_to_stream, internalize_from_stream;
043
044         // Class Modifiers:
045         dllname = "phone.dll";
046         metaclass = "SOMClass";
047
048         };
049 #endif /* __SOMIDL__ */
050 };
051 #endif /* phone_IDL */
```

Figure 6-8. Sample IDL for PhoneEntry (PO) class (file: phone.idl)

Step 2 - Compile the IDL

We compiled the IDL created in the previous step using the SOM compiler and the C++ emitter. See the *OS/390 SOMobjects Programmer's Guide* for details on using the SOM complier. The resulting C++ code file is used as the basis for the next step.

Step 3 - Write code for phoneEntry class

For this example, any code that we have added is marked by ***//+***. Any code unique to persistence that we added (that was not generated by the SOM compiler) is highlighted (bold). See Figure 6-9 on page 6-27. The SOM compiler generated much of the code.

Using the C++ code file created in the previous step, we completed the code stubs for the overridden **externalize_to_stream** and **internalize_from_stream** methods.

The **somStream::StreamIO** class will provide the methods used to build-up the stream (such as **write_int, write_float**, etc.). **write_object** should be used to externalize any contained object references (our example does not have any contained objects, however). See lines 222 through 224 and lines 243 through 245 in Figure 6-9 on page 6-27.

The parent's externalize and internalize methods need to be called to get the parent's persistent data into the stream. That call is generated by the SOM compiler (see line 220 and 240 in Figure 6-9 on page 6-27).

The **internalize_from_stream** uses a Lifecycle factory finder as an input parameter (generated by the SOM complier). The reason that a factory finder is used is in the situation where the persistent object to be restored contains other objects. When the **restore** method needs to restore the contained objects, there may be no class object available from which to instantiate. The Lifecycle factory finder first creates a class object of the correct type and then instantiates the contained object. **Restore** then retrieves the contained object state data from the stream.

Figure 6-9 on page 6-27 is the sample C++ code for PhoneEntry. It can be found in SOMMVS.SGOSSMPX.CXX(PHONE).

```
001 #ifndef SOM_Module_phone_Source
002 #define SOM_Module_phone_Source
003 #endif
004 #define phoneEntry_Class_Source
005
006 #include <phone.xih>
007
008 /*
009  * person's name
010  */
011
012 SOM_Scope string  SOMLINK MVS_phoneEntry__get_name(phoneEntry *somSelf,
013                                                    Environment *ev)
014 {
015     phoneEntryData *somThis = phoneEntryGetData(somSelf);
016     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry__get_name");
017 //+
018     return(strcpy(((string)SOMMalloc(strlen(somThis->name)+1)) ,som    This->name));
019 //+
020 }
021
022 /*
023  * person's name
024  */
025
026 SOM_Scope void  SOMLINK MVS_phoneEntry__set_name(phoneEntry *somSelf,
027                                                  Environment *ev,
028                                                 string name)
029 {
030     phoneEntryData *somThis = phoneEntryGetData(somSelf);
031     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry__set_name");
032 //+
033     if (somThis->name != NULL)
034         SOMFree(somThis->name);
035     somThis->name = (string) SOMMalloc(strlen(name)+1);
036     if (somThis->name != NULL)
037         strcpy(somThis->name, name);
038 //+
039 }
040
041 /*
042  * person's phone number
043  */
044
045 SOM_Scope string  SOMLINK MVS_phoneEntry__get_phone_number(phoneEntry *somSelf,
046                                                            Environment *ev)
047 {
048     phoneEntryData *somThis = phoneEntryGetData(somSelf);
049     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry__get_phone_    number");
050 //+
051     return(strcpy(((string)SOMMalloc(strlen(somThis->phone_number)+1)),
052                                      somThis->phone_number));
053 //+
054 }
055
056 /*
057  * person's phone number
058  */
059
060 SOM_Scope void  SOMLINK MVS_phoneEntry__set_phone_number(phoneEntry *somSelf,
061                                                          Environment *ev,
062                                                         string phone_number)
```

Figure 6-9 (Part 1 of 4). Sample C++ code for PhoneEntry.

```
063 {
064     phoneEntryData *somThis = phoneEntryGetData(somSelf);
065     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry__set_phone_number");
066 //+
067     if (somThis->phone_number != NULL)
068         SOMFree(somThis->phone_number);
069     somThis->phone_number = (string) SOMMalloc(strlen(phone_number)+1);
070     if (somThis->phone_number != NULL)
071         strcpy(somThis->phone_number, phone_number);
072 //+
073 }
074
075 SOM_Scope SOMObject*  SOMLINK MVS_phoneEntry_init_for_object_creation(phoneEntr\
076 y *somSelf,
077                                                                        Environm\
078 ent *ev)
079 {
080 //+
081     SOMObject* newsomSelf;
082 //+
083     phoneEntryData *somThis = phoneEntryGetData(somSelf);
084     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_init_for_object_creation\
085 ");
086     newsomSelf = phoneEntry_pcall_init_for_object_creation(somSelf,
087                                                            ev);
088 //+
089     somThis->name = (string)NULL;
090     somThis->phone_number = (string)NULL;
091     somThis->office = 0;
092     return newsomSelf;
093 //+
094 }
095
096 SOM_Scope SOMObject*  SOMLINK MVS_phoneEntry_init_for_object_reactivation(phone\
097 Entry *somSelf,
098                                                                        Envi\
099 ronment *ev)
100 {
101 //+
102     SOMObject* newsomSelf;
103 //+
104     phoneEntryData *somThis = phoneEntryGetData(somSelf);
105     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_init_for_object_reactiva\
106 tion");
107
108     newsomSelf = phoneEntry_pcall_init_for_object_reactivation(somSelf,
109                                                                ev);
110 //+
111     somThis->name = (string)NULL;
112     somThis->phone_number = (string)NULL;
113     somThis->office = 0;
114     return newsomSelf;
115 //+
116 }
117
118 SOM_Scope SOMObject*  SOMLINK MVS_phoneEntry_init_for_object_copy(phoneEntry *s\
119 omSelf,
120                                                                    Environment \
121 *ev)
122 {
123 //+
124     SOMObject* newsomSelf;
125 //+
126     phoneEntryData *somThis = phoneEntryGetData(somSelf);
127     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_ini
128     t_for_object_copy");
```

Figure 6-9 (Part 2 of 4). Sample C++ code for PhoneEntry.

```
129     newsomSelf = phoneEntry_pcall_init_for_object_copy(somSelf, ev);
130 //+
131     somThis->name = (string)NULL;
132     somThis->phone_number = (string)NULL;
133     somThis->office = 0;
134     return newsomSelf;
135 //+
136 }
137
138 SOM_Scope void SOMLINK MVS_phoneEntry_somDestruct(phoneEntry *somSelf,
139                                                   octet doFree,
140                                                   som3DestructCtrl* ctrl)
141 {
142     phoneEntryData *somThis; /* set in BeginDestructor */
143     somDestructCtrl globalCtrl;
144     somBooleanVector myMask;
145     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_somDestruct");
146     phoneEntry_BeginDestructor;
147
148     /*
149      * local phoneEntry deinitialization code added by programmer
150      */
151
152     phoneEntry_EndDestructor;
153 }
154
155 SOM_Scope void  SOMLINK MVS_phoneEntry_uninit_for_object_destruction(phoneEntry\
156  *somSelf,
157                                                                      Environme\
158 nt *ev)
159 {
160     phoneEntryData *somThis = phoneEntryGetData(somSelf);
161     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_uninit_for_object_destru\
162 ction");
163 //+
164     if (somThis->name != NULL)
165 {
166           SOMFree(somThis->name);
167         somThis->name = NULL;
168 }
169     if (somThis->phone_number != NULL)
170 {
171       SOMFree(somThis->phone_number);
172       somThis->phone_number = NULL;
173 }
174 //+
175
176     phoneEntry_pcall_uninit_for_object_destruction(somSelf, ev);
177 }
178
179 SOM_Scope void  SOMLINK MVS_phoneEntry_uninit_for_object_passivation(phoneEntry\
180  *somSelf,
181                                                                      Environme\
182 nt *ev)
183 {
184     phoneEntryData *somThis = phoneEntryGetData(somSelf);
185     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_uninit_for_object_passiv\
186 ation");
187
188     phoneEntry_pcall_uninit_for_object_passivation(somSelf, ev);
189 }
190
191 SOM_Scope void  SOMLINK MVS_phoneEntry_uninit_for_object_move(phoneEntry *somSe\
192 lf,
193                                                               Environment *ev)
```

Figure 6-9 (Part 3 of 4). Sample C++ code for PhoneEntry.

```
194 {
195     phoneEntryData *somThis = phoneEntryGetData(somSelf);
196     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_uninit_for_object_move");
197
198     phoneEntry_pcall_uninit_for_object_move(somSelf, ev);
199 }
200
201 SOM_Scope void  SOMLINK MVS_phoneEntry_somDumpSelfInt(phoneEntry *somSelf,
202                                                       long level)
203 {
204     phoneEntryData *somThis = phoneEntryGetData(somSelf);
205     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_somDumpSelfInt");
206
207     phoneEntry_pcall_somDumpSelfInt(somSelf, level);
208 }
209
210 SOM_Scope void  SOMLINK MVS_phoneEntry_externalize_to_stream(phoneEntry *somSel\
211 f,
212                                                              Environment *ev,
213                                                             CosStream_StreamIO\
214 * stream)
215 {
216     phoneEntryData *somThis = phoneEntryGetData(somSelf);
217     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_exte
218     rnalize_to_stream");
219
220     phoneEntry_pcall_externalize_to_stream(somSelf, ev, stream);
221 //+
222     stream->write_string(ev, somThis->name);
223     stream->write_string(ev, somThis->phone_number);
224     stream->write_short(ev, somThis->office);
225 //+
226 }
227
228 SOM_Scope void  SOMLINK MVS_phoneEntry_internalize_from_stream(phoneEntry *somS\
229 elf,
230                                                               Environment *ev,
231                                                              CosStream_Stream\
231 IO* stream,
233                                                              CosLifeCycle_Fac\
234 toryFinder* ff)
235 {
236     phoneEntryData *somThis = phoneEntryGetData(somSelf);
237     phoneEntryMethodDebug("phoneEntry","MVS_phoneEntry_internalize_from_stream"\
238 );
239
240     phoneEntry_pcall_internalize_from_stream(somSelf, ev, stream,
241                                              ff);
242 //+
243     somThis->name = stream->read_string(ev);
244     somThis->phone_number = stream->read_string(ev);
245     somThis->office = stream->read_short(ev);
246 //+
247 }
248
249 //+
250 SOMEXTERN void SOMLINK SOMInitModule (integer4 majorVersion,
251                                       integer4 minorVersion)
251 {
253     phoneEntryNewClass(0, 0);
254 }
255 //+
```

Figure 6-9 (Part 4 of 4). Sample C++ code for PhoneEntry.

Step 4 - Compile C++ code

The compiled C++ code from the previous step is packaged into a DLL.

We compiled the IDL created in the previous step using the SOM compiler. See *OS/390 SOMobjects Programmer's Guide* for details.

Step 5 - Write code for Main program (for POSIX datastore)

Now that the persistent class object code has been created, use the persistent class object to instantiate persistent **phoneEntry** objects and **store, restore** and **Delete** them from a POSIX datastore. POSIX acts just like a flat file, with one persistent object stored per file. If you have a very large number of objects, you might want to consider a different datastore.

Because we are using the explicit persistence model, we must explicitly store and restore the object using the **store** and **restore** methods inherited from PO.

Sample code for the following steps is given in Figure 6-10 on page 6-33, with syntax unique to persistence highlighted. Key elements of the Main program are:

1. Define a pointer to the **phoneEntry** class object (called **PhoneEntry** in this example).

   See line 013 in Figure 6-10 on page 6-33.

2. Instantiate the pid which is a DSOM object of **somPersistencePOSIX_PID_POSIX** class type. In the example, because of multiple inheritance, the (void*) must be specified.

   See line 023 in Figure 6-10 on page 6-33.

3. In the pid, set the full path name for the POSIX file (where the object's persistent data is to be stored). In the example, this is phone.dat.

   See line 025 in Figure 6-10 on page 6-33.

4. Instantiate the **phoneEntry** object (using **somdCreate**). **PhoneEntry** is an OS Server object of **phoneEntry** class type. Initialize the **phoneEntry** object and get a persistent object reference (using **init_for_object_creation**). Release the old reference created by **somdCreate**.

   See lines 028 - 030 in Figure 6-10 on page 6-33.

5. Define the **PhoneEntry** object's persistent data (in the example: person's name, phone number, and office location).

   See lines 032, 033, 034 in Figure 6-10 on page 6-33.

6. Set the pid in the **PhoneEntry** object.

   See line 035 in Figure 6-10 on page 6-33.

7. Store the **PhoneEntry** object, passing the pid on the **store** method. See line 036 in Figure 6-10 on page 6-33. A **checkError** method is used to check for any exception conditions. See line 37 and lines 079 - 093 in Figure 6-10 on page 6-33.

8. Add the name, phone number, and office location data for the **PhoneEntry** object and update the POSIX file by issuing a **store**. The second **store** operation does not require a pid (pass in NULL) because the pid only needs to be set the first time that the object is stored (the pid is an attribute of the Persistent Object). The second **store** operation overwrites the previous data.

See lines 040, 041, 042, and 043 in Figure 6-10 on page 6-33.

9. Restore the persistent **PhoneEntry** object from the POSIX file and display the phone entry. See line 047 and lines 055 - 057 in Figure 6-10 on page 6-33.
10. Destroy the phoneEntry object, this includes the persistent object reference and the state data in the persistent datastore. Also delete the pid.

    See lines 061 - 066 and line 069 in Figure 6-10 on page 6-33.

Figure 6-10 on page 6-33 can be found in SOMMVS.SGOSSMPX.CXX(PHONMAIN).

```
001 #include <somd.xh>
002 #include <sompposx.xh>
003 #include "phone.xh"
004
005 boolean checkError(Environment *ev);
006
007 int main (int argc, char *argvfff“)
008 {
009
010     // declare variables
011     Environment ev;
012     somPersistencePOSIX_PID_POSIX *pid;
013     phoneEntry *PhoneEntry, *TempPhoneEntry;
014     string phone_number;
015     string name;
016     short office;
017
018     // initialize environment
019     SOM_InitEnvironment (&ev);
020     SOMD_Init(&ev);
021
022     // instantiate and set up the pid
023     pid=(somPersistencePOSIX_PID_POSIX *)
(void *) somdCreate(&ev,"somPersistencePOSIX::PID_POSIX",TRUE);
024
025     pid->_set_pathName(&ev, "phone.dat");
026
027     // instantiate and set up the phone entry
028     TempPhoneEntry=(phoneEntry *) (void*) somdCreate(&ev,"phoneEntry", FALSE);
029     PhoneEntry=(phoneEntry *) ((void*)TempPhoneEntry->init_for_object_creation(&ev));
030     ((SOMDObject *) (void*) TempPhoneEntry)->release(&ev);
031
032     PhoneEntry->_set_name(&ev,"John Smith");
033     PhoneEntry->_set_phone_number(&ev,"507-253-0000");
034     PhoneEntry->_set_office(&ev,42);
035     PhoneEntry->_set_p(&ev,pid);
036     PhoneEntry->store(&ev,pid);
037     checkError(&ev);
038
039     // this entry will overwrite the "John Smith" entry
040     PhoneEntry->_set_name(&ev,"John Doe");
041     PhoneEntry->_set_phone_number(&ev,"555-555-1111");
042     PhoneEntry->_set_office(&ev,61);
043     PhoneEntry->store(&ev,NULL);
044     checkError(&ev);
045
046     // restore the entry
047     PhoneEntry->restore(&ev,pid);
048     if (checkError(&ev) == FALSE)
049       {
050         name = PhoneEntry->_get_name(&ev);
051         phone_number = PhoneEntry->_get_phone_number(&ev);
052         office = PhoneEntry->_get_office(&ev);
053
054         // print the restored phone entry
055         somPrintf("Name: %s\n", name);
056         somPrintf("Phone Number: %s\n", phone_number);
057         somPrintf("Office: %d\n", office);
058       }
```

Figure 6-10 (Part 1 of 2). Sample C++ code for main program using explicit persistence.

```
059
060     // delete the persistent object
061     PhoneEntry->uninit_for_object_destruction(&ev);
062
063     PhoneEntry->Delete(&ev,pid);
064     checkError(&ev);
065
066     PhoneEntry->somFree();
067
068     // delete the pid
069     pid->somFree();
070     // cleanup the environment
071     SOMD_Uninit(&ev);
072     SOM_UninitEnvironment (&ev);
073
074     return 0;
075 }
076
077
078 // check for Errors
079 boolean checkError(Environment *ev)
080 {
081     char *exID;
082
083     // check for any exception
084     if (ev->_major != NO_EXCEPTION)
085       {
086         exID = somExceptionId(ev);
087         somPrintf("Error Occurred - Exception Id = %s\n", exID);
088         somdExceptionFree(ev);
089         return TRUE;
090        }
091     else
092        return FALSE;
093 }
```

Figure 6-10 (Part 2 of 2). Sample C++ code for main program using explicit persistence.

Step 5 - Write code for Main program (for VSAM (B-Tree) datastore)

If we want to use a VSAM (B-Tree) datastore, rather than a POSIX datastore, some lines in the Main program change, as follows:

- Changes to the Main program in Figure 6-10 on page 6-33 for the B-Tree datastore are:
  - Line 002 is replaced with: #include <sompbt.xh>
  - Line 012 is replaced with: somPersistenceBTREE_PID_BTREE *pid;
  - Line 023 is replaced with: pid=(somPersistenceBTREE_PID_BTREE *)(void *) somdCreate(&ev, "somPersistenceBTREE::PID_BTREE","TRUE");
  - Because B-Tree requires a datastore name and a key, line 025 is replaced with the following 2 lines:

    ```
    pid->_set_datastore_name(&ev, "phone.dat");
    pid->_set_object_key(&ev, "John_key");
    ```

Step 6 - Compile and link Main program

Compile and link the main program. In general, a client program needs to link the following libraries when using POSSOM:

```
INCLUDE IMPORT(GOSSOMK)
INCLUDE IMPORT(GOSSOMTC)
INCLUDE IMPORT(GOSSOMD)
INCLUDE IMPORT(GOSPPOF)
INCLUDE IMPORT(GOSPPOSX)
INCLUDE IMPORT(GOSESTRM)
INCLUDE IMPORT(GOSOS)
INCLUDE IMPORT(GOSSOMU)
INCLUDE IMPORT(PHONE)
```

To use the BTREE(VSAM) datastore instead, the include for the GOSPPOSX would change to: INCLUDE IMPORT(GOSPBT), or a combination of both. For more information on prelinking and linking, see the *OS/390 SOMobjects Programmer's Guide*.

Step 7 - Create/update your SOMIR

Update the **SOMIR** with the interface definition of the phone entry (file phone.idl).

By using the SOM compiler options **-u** and **-sir**, the SOM interface repository (**SOMIR**) is created. See *OS/390 SOMobjects Programmer's Guide* for more details.

Step 8 - Create your Implementation Repository (regimpl)

regimpl is used to create the registration of DSOM or OS Server objects in the implementation repository. The implementation repository determines where DSOM objects reside (in which server or client processes). Because objects provided by POSSOM framework (such as PID, PDS) are DSOM objects, you need to decide where those objects should be instantiated. Since PO objects are OS Server objects, you have to register them also.

For the PhoneEntry example, we registered the following objects as OS Server objects in the implementation repository:

- somPersistencePOSIX::PID_POSIX
- somPersistencePOSIX::PDS_POSIX
- phoneEntry

See “Implementation Repository” on page 6-36 for details on creating your implementation repository and tips on where these objects should reside.

Step 9 - Define and Manage Your Servers with Workload Manager (WLM).

WLM manages workload distribution, workload balancing, and provides a solution for distributing resources to competing workloads.

The following tasks are necessary in order for your servers and classes to run and have their performance managed in the SOMobjects environment:

- Defining your servers to WLM.
- Managing performance with WLM.
- Managing your servers with WLM.

For more information on how to do this, see *OS/390 SOMobjects Configuration and Administration Guide*.

Step 10 - Create text file to be read by the POM

A default text file which is read by the POM is shipped with POSSOM. You do not need to update this file unless you want to add a new PDS. For this example, the default **SOMMVS.SGOSMISC(GOSPMDAT)** dataset is used. A **SOMMVS.SGOSMISC(GOSPMDAT)** dataset must be available on each machine where the POM is running.

The format of the default text file is shown in "Default SOMMVS.SGOSMISC(GOSPMDAT) dataset shipped with POSSOM" on page 6-37.

General POSSOM Administration

Implementation Repository

When creating your implementation repository, you need to decide where the following objects related to POSSOM reside:

- **PDS**. You need to determine where the PDS object resides. The PDS object is a DSOM object. It is recommended that you create and store your PDS in the same server (process) as your datastore. The PDS does many operations with the datastore, so for performance reasons the PDS and datastore should reside in the same process space.
- **PID**. You need to determine where the PID object resides. The PID object is a DSOM object We recommend that the PID reside in the same process as the POM (local/client process) or in the same process as the PDS.
- **StreamIO**. You need to determine where the StreamIO object, which is the container for your object's persistent data, is to reside. The StreamIO object is a DSOM object. Since there is so much interaction between your object and the StreamIO object, it is recommended that these be in the same process.
- **Your persistent object**. You need to determine where your persistent object (PO) resides. The persistent object is always an OS Server object. Wherever you decide that your object should reside, the stream object should reside in the same process.

Tips for Registering Classes

Sample line commands to register your OSServer and classes assuming a POSIX PDS:

- Start the SOM daemon (if it is not already running)
- Register the Server:

```
regimpl -A -i PhoneServer -v somOS::Server -t SOMD_TCPIP
```

> **Important**: The first time you run myOSServer after using the **regimpl** command, you must start it manually to initialize the server via "somossvr -a PhoneServer - i". Once this is done, your server will be started whenever it is needed (this is done via DSOM & the Naming Service).

SOMMVS.SGOSMISC(GOSPMDAT) Dataset

The dataset containing the text file read by the POM must be available on each machine where the POM is running. The default dataset shipped with POSSOM is shown in “Default SOMMVS.SGOSMISC(GOSPMDAT) dataset shipped with POSSOM.” Any number of tabs or spaces may separate the entries in the table.

The location of the **SOMMVS.SGOSMISC(GOSPMDAT)** dataset is defined in the environment variables file, GOSENV.INI. You must manually update this file with the path name for the **SOMMVS.SGOSMISC(GOSPMDAT)** dataset. To update the GOSENV.INI file, do the following:

1. Find the POSSOM stanza in the GOSENV.INI file (identified by the string SOM_POSSOM)
2. Update the line "POS_POMDATA = " with the *pathname* of the text file which is to be read by the POM. For example, if the text file resides in the HFS with a file name of pom.dat, you would have: POS_POMDATA=/pom.dat. You also can use a fully qualified path name.

   POS_POMDATA can point to one of the following:

   - a file which resides in the HFS. (e.g. pom.dat or /u/userid/pom.dat)
   - a sequential or partitioned system dataset. (e.g. //pom.dat or //'hlq.pom.dat' or //'hlq.sgosmisc(gospmdat)'

     Note: as mentioned in the GOSPMDAT comments, when this environment variable is not set, a default path of //'<hlq>.<sombase>.sgosmisc(gospmdat)' will be used where <hlq> is the high level qualifier name, and <sombase> is the value of the SOMBASE environment variable from the GOSENV.INI file.

Default SOMMVS.SGOSMISC(GOSPMDAT) dataset shipped with POSSOM:
The following is the default SOMMVS.SGOSMISC(GOSPMDAT) dataset shipped with POSSOM.

```
;
;  GOSPMDAT
;
;  This file is used by the Persistent Object Manager (POM) to route
;  PO functions to the appropriate PDS.  This file can contain blank lines
;  or comments (preceded by ';' or '#').  When the POM is created, it
;  loads this file.  When a POM function is called (e.g. store) it checks
;  each mapping (first to last) until it finds a match.  A match is when
;  the PO object inherits from Protocol (somIsA) and the PID datastore_type
;  equals datastore.  The datastore does not correspond to anything, it is
;  just a string in this table and in the PID. The POS_POMDATA environment
;  variable in the SOM_POSSOM stanza of the GOSENV file must contain the
;  path and filename to this file.
;
;  ProtocolType            datastore  PDSType
;  --------------          ---------  --------
CosStream::Streamable      IBM_POSIX  somPersistencePOSIX::PDS_POSIX
CosStream::Streamable      IBM_BTREE  somPersistenceBTREE::PDS_BTREE
```

Note: Any changes to the POM text file do not take effect until the POM object is instantiated (which might not be until the next time the server is restarted.)

Index

O

Communicating Your Comments to IBM

OS/390
SOMobjects
Object Services

Publication No. SC28-1995-01

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing an RCF from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+914+432-9405
- If you prefer to send comments electronically, use this network ID:
  - IBMLink: (United States customers only): KGNVMC(MHVRCFS)
  - IBM Mail Exchange: USIB6TC9 at IBMMAIL
  - Internet e-mail: mhvrcfs@us.ibm.com
  - World Wide Web: http://www.s390.ibm.com/os390

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

Reader's Comments — We'd Like to Hear from You

OS/390
SOMobjects
Object Services

Publication No. SC28-1995-01

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: _____

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

[ ]	As an introduction	[ ]	As a text (student)
[ ]	As a reference manual	[ ]	As a text (instructor)
[ ]	For another purpose (explain)		

_____

_____

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Name	Address
Company or Organization	
Phone No.	

Reader's Comments — We'd Like to Hear from You
SC28-1995-01

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie NY 12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5696-822

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

Drop in
Back Cover
Image Here.

SC28-1995-01