Component Broker Programming Guide Release 2.0

Document Number G04L-2376-04

November 20, 1998

Component Broker

G04L-2376-04

Programming Guide

Release 2.0

IBM

G04L-2376-04

Component Broker

Programming Guide

Release 2.0

Note

Before using this information and the product it supports, be sure to read the general information under Appendix E, “Notices” on page 355.

Fifth Edition (December, 1998)

This edition applies to Release 2.0 of Component Broker and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

What's New!

The following changes were made to this publication since the previous edition:

- Added programming model documentation changes in theMOFW Client Programming Model chapter.
- Added information on loading C++ DLLs from Java BO in the Java Server Programming Model chapter.
- Improved the ActiveX Client documentation in theMOFW – ActiveX Client Programming Model chapter.
- Many platform-specific sections were moved to Assembling and Installing Business Objects on AIX and Windows NT chapter. For platform-specific information on OS/390 Component Broker, see *OS/390 Component Broker Programming: Assembling Applications* .
- The Unit Test chapter has been moved to an appendix section, see Unit Test Environment.
- An index has been added along with other various technical and editorial changes.

About This Book

The *Component Broker Programming Guide* describes the Component Broker programming model and presents sample code showing common tasks required to develop typical object-oriented applications.

This preface includes the following sections:

- “Who Should Read This Book”
- “How this Book is Organized”
- “Documentation Conventions” on page xvi
- “Notation” on page xvii
- “The Component Broker Documentation” on page xxi

Who Should Read This Book

The *Component Broker Programming Guide* is intended for application developers who use the Component Broker environment to build robust, distributed object-oriented applications.

The examples are written in C++; therefore, programming experience in C++ and a background in object-oriented programming is required. A familiarity with Java is also helpful, but not required.

This book is not a programming manual; it is for experienced programmers who are going to use this product.

How this Book is Organized

Chapter 1, “Introduction” on page 1 gives an introduction and describes some common object-oriented analysis and design concepts that pervade the Component Broker programming model. It gives a simple overview of a deployed, distributed object-oriented application. It presents a simple categorization of roles and responsibilities later in the chapter. Throughout this document, each programming model concept and task, and the roles that perform each task are explained. This chapter concludes with a definition of extended Object Modeling Technique (OMT Rumbaugh) notation and coding conventions used in the rest of the document.

Chapter 2, “Personal Life Insurance Application Example” on page 15 describes a simple Personal Life Insurance Application. Aspects of this application development scenario are used in the rest of the document wherever a real example is needed for illustration purposes.

Chapter 3, “The Managed Object Framework” on page 29 introduces the Managed Object Frameworks and positions them with another set of frameworks called the Component Broker Frameworks. This chapter tells how Component Broker implements and exposes business objects that are important for all readers to understand.

Chapter 4, “MOFW Client Programming Model” on page 33 describes the basic cient programming model. This chapter is organized in terms of basic tasks that the developer of a Component Broker client application (object) performs to use Component Broker server objects.

Chapter 5, “MOFW Server Programming Model” on page 57 provides the server side details of the Managed Object Framework, and explains the Component Broker server programming model. This section explains how to:

- Implement business objects using the MOFW.

- Map Component Broker objects onto existing applications and databases to achieve operational reuse.

Chapter 6, “MOFW Client Programming Model – Advanced Concepts” on page 83 provides additional options and capabilities for programmers who are creating application objects.

Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105 provides additional options and capabilities to be considered when creating managed objects. These advanced features build upon the basics laid out in Chapter 5, “MOFW Server Programming Model” on page 57.

Chapter 8, “MOFW – ActiveX Client Programming Model” on page 143 tells how to access and update Component Broker server objects from ActiveX/COM clients. Component Broker provides special support for these applications.

Chapter 9, “MOFW - Java Client Programming Model” on page 157 replays the client programming model described in Chapter 4, “MOFW Client Programming Model” on page 33; however, it is focused on Java-CORBA clients and how business objects are accessed and used from a Java client.

Chapter 10, “Java Server Programming Model” on page 171 describes how to use Java to implement managed objects that run in a Component Broker server.

Chapter 11, “Assembling and Installing Business Objects on AIX and Windows NT” on page 197 discusses how to use the Component Broker Application Development tools to build client and server objects. This information is for those who want to go it alone, with command line tools. This chapter includes the customizations necessary to install and run business objects on particular instance managers.

Appendix A, “Artifacts Produced in Building Objects” on page 269 lists the specifics of the Component Broker object models and frameworks and provides summary documentation on their interfaces and semantics. A brief overview of terminology and OOA/OOD concepts is included. This section is necessary reading only for those interested in the mapping between object-oriented analysis (OOA) and object-oriented design (OOD) and Component Broker concepts.

Appendix B, “Interface Definition Language” on page 271 discusses IDL topics including name scoping, declarations, syntax, idlc and idl2com commands and the IDL-to-Java complier.

Appendix C, “C++ CORBA Programming” on page 299 discusses C++ topics including bindings, name scoping, storage management, and binding restrictions.

Appendix D, “Unit Test Environment” on page 333 describes an environment supplied by Component Broker that you can use to unit test business objects and applications.

Documentation Conventions

The following conventions distinguish different text elements:

plain	Window titles, folder names, icon names, and method names.
`monospace`	Programming examples, user input at the command line prompt or into an entry field, user output, and directory paths.
bold	Menu choices, push buttons, check boxes, radio buttons, group-box controls, drop-down list boxes, combo-boxes, notebook tabs, and entry fields.
italics	Programming keywords, variables, and attributes, titles of information units, initial use of unique terms, and emphasis.

The following icons are used to indicate platform-specific sections.

WIN	Denotes a section that applies only to the Windows 95 or Windows NT platform. Do not interpret this symbol to denote that an equivalent AIX section exists. **Note:** The Windows 95 platform only supports the Component Broker Java client.
AIX	Denotes a section that applies only to the AIX platform. Do not interpret this symbol to denote that an equivalent Windows section exists.
OS 390	Denotes a section does not apply to OS/390 Component Broker. Do not interpret this symbol to denote that an equivalent section exists in OS/390 Component Broker.

Notation

Even though there is a large degree of commonality between the content of various object methods, there is less agreement when it comes to the notations used to describe the content (that is, its form).

Throughout this document, the class diagrams use the Object Modeling Technique (OMT) notation developed by James Rumbaugh, with the exception that it is distinguished between interface and implementation inheritance.

Notations used in this document are:

- Interface inheritance
- Implementation inheritance
- Association
- Aggregation
- Qualified association

There is explanatory text in the examples to further clarify the graphical Rumbaugh notation.

If you are already familiar with these notations, skip ahead to “State Transition Diagrams” on page xx or “Data Flow Diagrams” on page xx.

Interface Inheritance

Figure 1 shows that a child class inherits just the interface, but not the implementation, of a parent class.

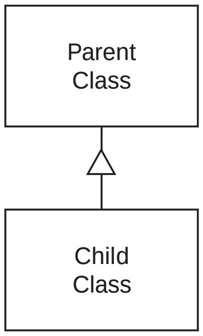


Figure 1. Class Inheriting Interface but not Implementation

Implementation Inheritance

The following notation shows that a child class inherits the implementation, not just the interface, of a parent class:

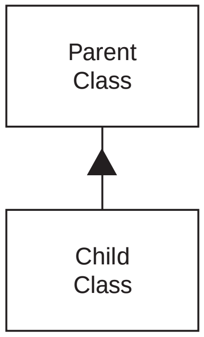


Figure 2. Class Inheriting Implementation and Interface

This notation is usually reserved for OOD models, as OOA models normally focus on semantics, and therefore interfaces, rather than implementations.

Associations

The notation for associations, sometimes referred to as *by reference* or *uses-a* relationships, is the following:

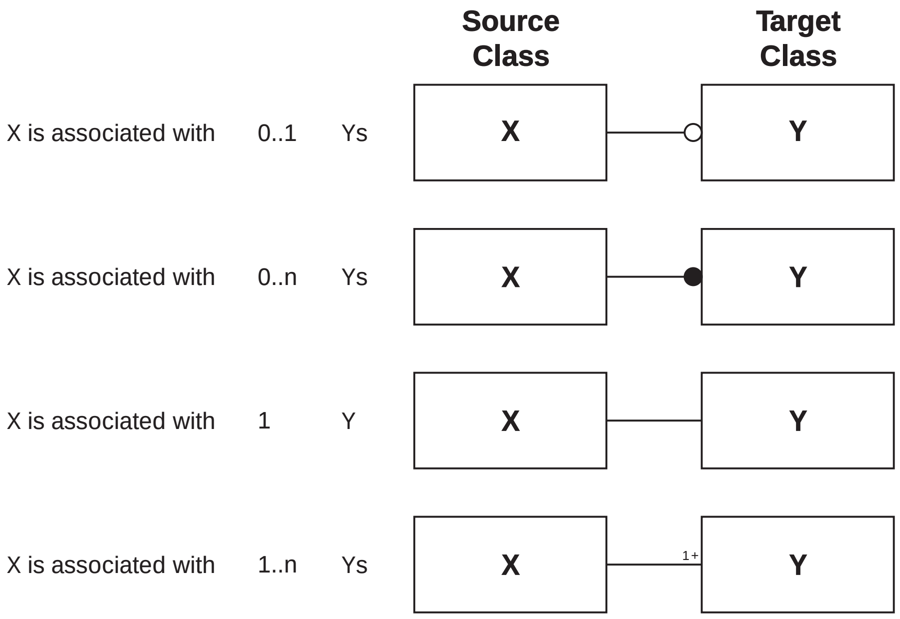

Source Class
Target Class
X is associated with 0..1 Ys
X
Y
X is associated with 0..n Ys
X
Y
X is associated with 1 Y
X
Y
X is associated with 1..n Ys
X
1+
Y

Figure 3. Associations Notation

Aggregation

The notation for aggregations, which are sometimes referred to as containment or *has-a* relationships, is the following:

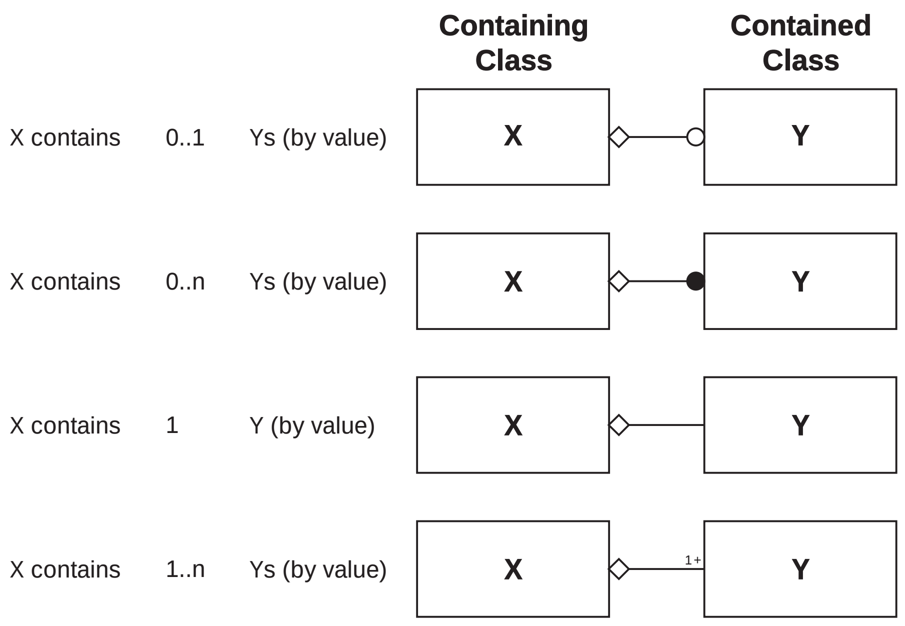

Containing Class
Contained Class
X contains 0..1 Ys (by value)
X contains 0..n Ys (by value)
X contains 1 Y (by value)
X contains 1..n Ys (by value)
X
Y
1+

Figure 4. Aggregation Notation

Qualified Association

A qualified association is a way to constrain the targets in a relationship without requiring that a special subclass be created. The technique is to add an attribute to the relationship that can be considered to be a unique identifier with respect to the source class.

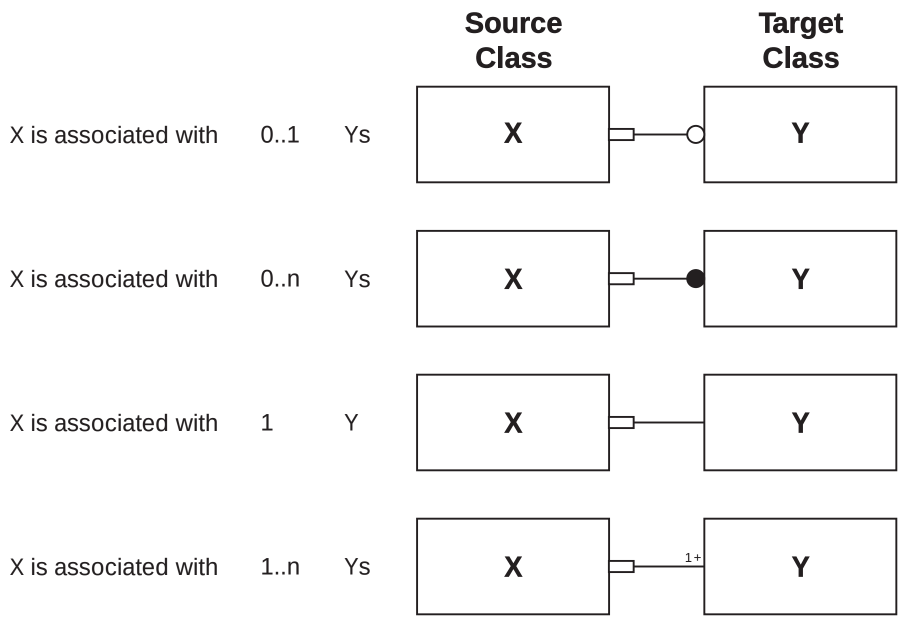

Source Class
Target Class
X is associated with 0..1 Ys
X is associated with 0..n Ys
X is associated with 1 Y
X is associated with 1..n Ys
X
Y
1+

Figure 5. Qualified Association Notation

Sometimes the attribute used to constrain the relationship appears in the box on the relationship.

State Transition Diagrams

When an object exhibits a complex life cycle model, a simplified form of state-transition diagram (STD) is used. This notation makes no distinction between initial and final states (because that can be ascertained from the transitions); however, it separates the conditions from the actions, where the action is a method on the object in question either from the static or recursively, the dynamic model as follows:

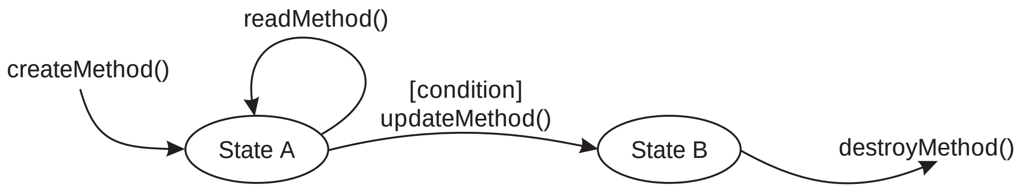


Figure 6. State Transistion Diagrams

When invoking the method on the object that can be assumed to be the *trigger*, no condition notation is required on the diagram although some programmers include one to show the objects that can invoke the method, and under what conditions, if any.

A loop back transition is no guarantee that a method is read-only as shown in the diagram. Of course, it does not change the overall state of the object as shown in the diagram. A recursive dynamic model might show a change of substate.

The method categorizations shown here are included as an introduction only. Their use within the Component Broker programming model to enable optimistic service implementations are described in Chapter 11, “Assembling and Installing Business Objects on AIX and Windows NT” on page 197.

Data Flow Diagrams

When the action in a transition, usually a method, is complex enough to warrant more than a simple text explanation or some basic pseudo-code, relatively standard Yourdon DeMarco data flow diagrams are used. The only difference is that objects appear as data source or *sink* objects, for example the box in the following diagram:

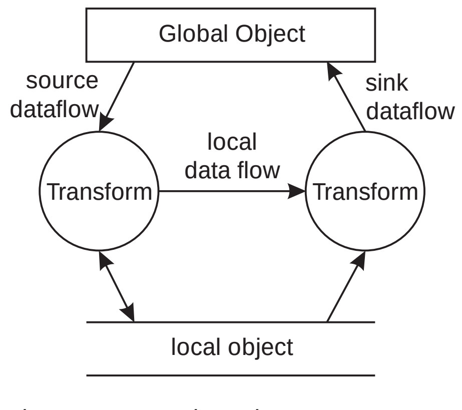


Figure 7. Data Flow Diagrams

As the diagram shows, a data flow can be labeled or unlabeled, depending on whether the entire object is used by the transform or not. Also, a data flow can be a read-and-write type data flow as required to reduce the amount of clutter.

The Component Broker Documentation

The following information is part of Component Broker:

- Help information is available from Component Broker product panels.
- The Component Broker online library can be viewed using a frames-compatible Web browser.
- *Component Broker for Windows NT and AIX Quick Beginnings*, G04L-2375 explains how to easily create and verify a starter Component Broker environment. These instructions walk the user through a typical server and client installation. Users can extend this configuration using the information in the *Component Broker for Windows NT and AIX Planning, Performance and Installation Guide* .
- *Component Broker for Windows NT and AIX Planning, Performance and Installation Guide*, SC09-2798 provides a comprehensive overview of the Component Broker environment, then guides the user through planning considerations including capacity planning, performance tuning, prerequisites, and migration. It also leads the user through installation options for all Component Broker environments.
- *Component Broker for Windows NT and AIX CICS and IMS Application Adaptor Quick Beginnings*, GC09-2703 provides a brief technical overview of the CICS and IMS application adaptor and guides the user through its installation and configuration. Step-by-step instructions guide the user through creating an initial CICS and IMS application using application development tools included in the CBToolkit package.
- *Component Broker for Windows NT and AIX Oracle Application Adaptor Quick Beginnings*, GC09-2733 provides a brief technical overview of the Oracle application adaptor and guides the user through its installation and configuration. Step-by-step instructions guide the user through creating an initial Oracle application using application development tools included in the CBToolkit package.
- *Component Broker for Windows NT and AIX System Administration Guide*, SC09-2704 provides information about configuring and operating one or more hosts managed by Component Broker. It also provides general information about using the System Manager User Interface.
- *Component Broker Application Development Tools*, SC09-2705 explains how to create and test Component Broker applications using the tools provided in the Component Broker Application Development Tools with a focus on common development scenarios such as inheritance and team development.
- *Component Broker Advanced Programming Guide*, SC09-2708 describes the Component Broker implementation for the CORBA Object Services and the Component Broker Object Request Broker (including remote method invocation and the Dynamic Invocation Interface (DII) procedures), Session Service, Cache Service, Notification Service, Interlanguage Object Model (IOM), and work-load management (WLM).
- *Component Broker Programming Reference*, SC09-2810 contains information about the APIs available to Component Broker application developers.
- *Component Broker for Windows NT and AIX Problem Determinaion Guide*, SC09-2810 explains how to identify and resolve problems within a Component Broker environment using the tools provided with Component Broker. The book includes information on installation problems, run time errors, debugging of applications, and analysis of log messages.
- *Component Broker Glossary*, SC09-2710 contains terms and definitions relating to Component Broker.
- *OS/390 Component Broker Introduction*, GA22-7324 describes the concepts and facilities of Component Broker and the value it has on the OS/390 platform. The audience is a knowledgeable decision maker or a system programmer.
- *OS/390 Component Broker Planning and Installation*, GA22-7331 describes the planning and installation considerations for Component Broker on OS/390.

- *OS/390 Component Broker System Administration*, GA22-7328 describes system administration tasks and operations tasks, as provided in the system administration user interface for OS/390.
- *OS/390 Component Broker Programming: Assembling Applications*, GA22-7326 provides information for assembling applications using Component Broker on OS/390.
- *OS/390 Component Broker Operations: Messages and Diagnosis*, GA22-7329 provides diagnosis information and describes the messages associated with Component Broker on OS/390.

Chapter 1. Introduction

Component Broker is a client/server solution for object-oriented applications, in two respects:

- A deployed solution using Component Broker has a set of processes that serve objects to a set of client applications. This logical, client/server process model can be implemented in a one, two, or three tier client/server system, and a process may be both a client and a server.
- Object-oriented programming is client/server by nature. A server object provides an implementation of an interface. Other client objects implement new functions using the interfaces and data provided by server objects. An object may be a client of some objects and a server for others.

Due to the client/server nature of object-oriented applications, the Component Broker programming model is divided into two sub-models, the client programming model and the server programming model.

Component Broker provides frameworks and classes that implement common functions needed to build distributed object-oriented commercial applications. Some examples are transactions and a Naming Service. Much of the value in Component Broker derives from these supplied services, which enable developers to focus on their business logic and relieves them from the burden of implementing basic, cross-application functions.

In addition to its pre-packaged services, Component Broker is extensible. End users, system integrators (SIs) and independent software vendors (ISVs) can extend Component Broker by providing additional sets of reusable services, or providing alternate pluggable replacements for the Component Broker services. Component Broker can be extended in many ways, but the most important extensions are:

- Add new *application adaptors*. An application adaptor provides interoperability between Component Broker objects and external environments such as DB2, CICS, and SAP. The application adaptor provides frameworks and services that:
  - Render or wrapper the external, non-object-oriented data and applications as Component Broker server objects.
  - Provide tailored implementations of Managed Object Framework and Component Broker Services that interoperate with the services provided by the external system. For example, a Component Broker application adaptor for DB2 provides an implementation of Object Transaction Services that interoperates with the databases transaction services.
  - Component Broker comes with a set of application adaptors. They do not support all external environments. Therefore, customers, ISVs, and SIs can extend the capabilities of a Component Broker server by implementing new application adaptors.
- Add new object services and frameworks that can be used by business objects. Some examples could be frameworks or services for advanced transaction models or a business rules engine.

Component Broker is intended as an infrastructure or technical architecture which naturally supports object-oriented applications produced through object-oriented analysis and design. This document explains how Component Broker concepts are related to or derived from object-oriented analysis (OOA) and object-oriented design (OOD) concepts.

Object-Oriented Application Development Paradigms

Object-oriented application development can be approached in the following ways:

Top-down
Modeling and analysis define the classes and behaviors that must be implemented.

Bottom-up
Existing applications and databases provide functions and data that must be wrapped through encapsulating classes and instances.

Meet in the middle
A combination of top-down for new functions and bottom-up for reuse and incremental reengineering of old applications.

The Component Broker programming model supports these scenarios.

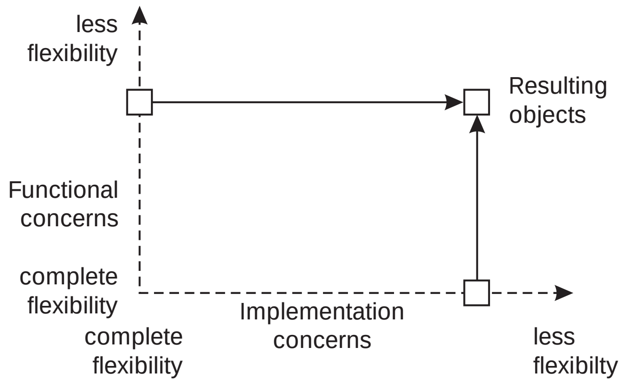


Figure 8. Degrees of Freedom During Application Development

The models can be characterized by their balancing of the concerns of the functional domain versus those of the underlying implementation domain or design constraints. This balancing defines the degrees of freedom possible during object-oriented analysis and design. Usually the degree of freedom depends on the amount of legacy applications and databases involved along that dimension.

Typically, a given application of any complexity uses a mix of all three models.

Forward Engineering New Functional Requirements (Top-down)

Top-down development can be considered to be the ideal model for object-oriented applications because objects directly derived from functional requirements are the goal. At a high level, the main steps in the top-down process are:

1. Business Process Modeling (BPM) defines the application requirements and functions that must be performed. This includes requirements specification and use case analysis.
2. Object-oriented analysis and object-oriented design uses a methodology and design guidelines to produce an Object Model. The Object Model defines the relationships between classes, for example inheritance, aggregation, and usage and the methods and attributes of the classes. Detailed object-oriented analysis and design may also define state transitions and data flows. Finally, the model may contain semantic metadata information that describes more details of the model and the behaviors of classes. This metadata is used as the starting point for implementing the classes, and may be the basis for code generation.
3. Object Implementation involves defining the classes in an implementation language such as C++, Java, or Smalltalk. This step also involves implementing the objects as directly as possible in an underlying database and the use of other object services.

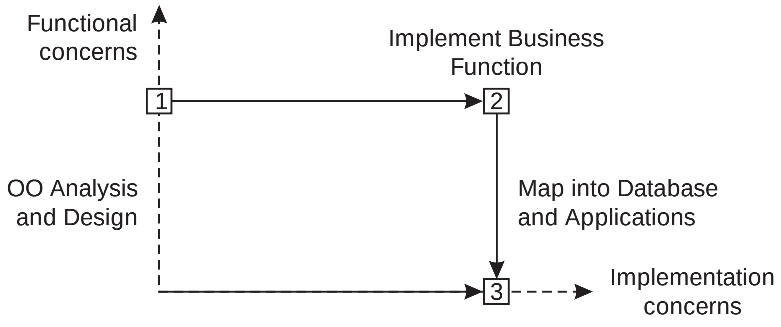


Figure 9. Top-down Development

The Component Broker programming model and tools support top-down development by:

- Loading Object Models produced by popular front-end BPM and OOA/OOD tools into the Component Broker application development tools.
- Providing an Object Builder that eases the development of newly defined business objects that are derived from the Managed Object Framework. The builder visually *tools* the Component Broker programming model and automates the implementation of some MOFW framework methods by generating code.
- Installing the new classes into an application adaptor to give the business objects access to object services, including persistence. This includes generation of a database scheme for persistent objects and emitting the implementation of methods and helper classes.

Reverse Engineering from the Legacy (Bottom-up)

Bottom-up development involves the composition of new applications from assembly and reuse of existing functions and applications. One of the touted benefits of object-oriented programming is improved productivity through reuse of existing classes. In many cases, however, a non-object-oriented legacy application or database must be reused. When applied to object technology, this technique is often called *wrappering*. That is, the objects developed are usually just thin wrappers around legacy code or data that must be preserved in the new object-oriented application.

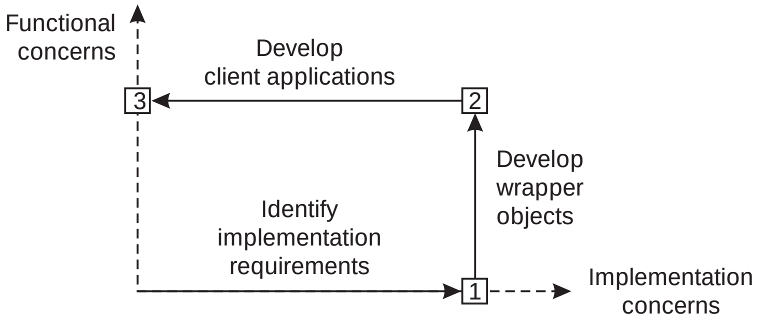


Figure 10. Steps Involved in Reverse Engineering

When wrappering legacy data, the objects usually are little more than a set of public attributes, or get and set methods, corresponding to a logical grouping of data, such as a row in a database table or record in a file.

For legacy code, it is usually the case that multiple transaction programs share a given parameter that tacitly identifies an underlying object. For example, a bank might have the following transactions:

- Create and open a checking account.

- Close a checking account.
- Credit to checking account.
- Debit from checking account.
- Read and update taxpayer's Social Security Number for the account.

In this simple example, the checking account number effectively identifies an object. The existing transactions are reused in the implementation of object methods. Transaction input and output parameters are mapped onto object state data, and the parameters of object methods.

The term operational reuse is used to describe the wrappering and reuse of existing databases and applications in the development of new object-oriented applications. For many readers of this document, the concept of reusing data is obvious. Why do customers want to reuse applications, instead of having the objects directly access the databases used by the old, legacy applications? There are two reasons for application reuse:

- The existing applications embody a substantial investment, and it is too expensive to rewrite the business logic. There are literally trillions of lines of existing application code which represents an investment of billions of dollars. The old applications need to be supported anyway, at least for a while, because not all client applications and terminals that use the legacy functions can be discovered and updated.

- In principle, new business logic could directly access the data. For example, the CheckingAccount::debit() method could directly update the database record. However, most apparently simple applications, like the debit transaction, implement imbedded business rules and database integrity constraints over multiple databases, and have side effects. Objects directly accessing the underlying data must exactly implement these semantics. To do so effectively results in rewriting the existing application, which is too expensive and would require ensuring consistent behavior over two bodies of code.

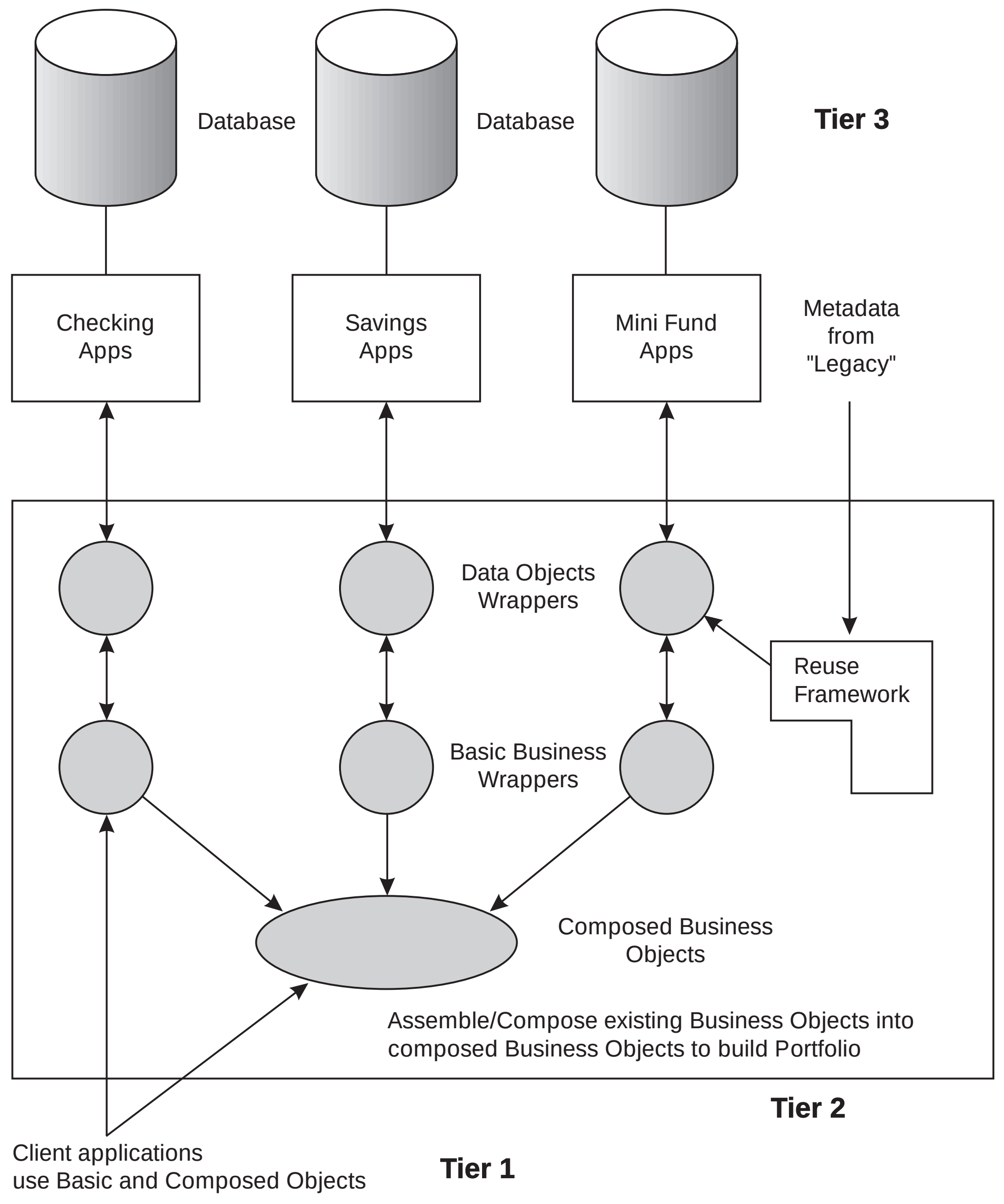

Database
Database
Tier 3
Checking Apps
Savings Apps
Mini Fund Apps
Metadata from "Legacy"
Data Objects Wrappers
Reuse Framework
Basic Business Wrappers
Composed Business Objects
Assemble/Compose existing Business Objects into composed Business Objects to build Portfolio
Tier 2
Client applications use Basic and Composed Objects
Tier 1

Figure 11. The Operational Reuse Model

The bottom-up development process follows a well defined set of steps:

1. Wrapper existing applications and databases through encapsulating data objects. The Data Objects provide a direct object-oriented rendering of the existing applications and databases.
2. Implement business objects that provide a veneer over the data objects to convert them into views required by the object-oriented model.
3. Implement business objects that derive their state data and behavior from multiple or other business objects. The composed objects provide a more natural view of the existing, legacy defined model and provide a more natural object-oriented model for newly developed applications.
4. Develop new object-oriented applications.

The Component Broker programming model and supporting Application Development tools enable bottom-up object-oriented development by:

- Introducing the concept of application adaptors that provide interoperability between business objects and existing legacy environments. For example, Component Broker comes with a Relational Database application adaptor that enables object reuse of existing relational databases.
- Providing a set of application adaptor specific class libraries and frameworks that are supported by Object Builder extensions to facilitate the construction of the wrapper objects from existing metadata. For example, in the Relational Database application adaptor, this metadata is the database catalog that defines tables and rows.
- Providing a framework and visual tool in the Object Builder for construction by parts based definition and implementation of new business objects.
- Enablement of client applications that use server objects by generating client parts such as Java Applets or ActiveX/COM components from the server class definitions.

Meet in the Middle, Incremental Development

The first two paradigms represent the extremes of new object applications and direct rendering of existing applications. Most large development projects are a mix of both approaches. When top-down meets bottom-up, what happens? A developer would be lucky if the classes developed bottom-up exactly matched the interface and function requirements of the newly developed top-down model. In the entire history of object-oriented application development, no one has ever been that lucky. To meld top-down and bottom-up, it is necessary to create *mapping* objects that:

- Implement a facade that transforms the old object into one required by the new client objects.
- Are composed from existing bottom-up objects and reuse their implementation; can override existing functions and can add new methods.

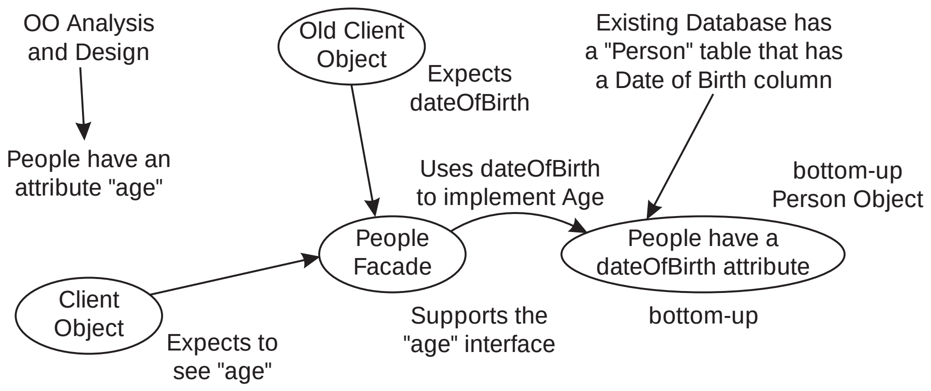

OO Analysis and Design
People have an attribute "age"
Old Client Object
Expects dateOfBirth
Existing Database has a "Person" table that has a Date of Birth column
Uses dateOfBirth to implement Age
bottom-up Person Object
People Facade
People have a dateOfBirth attribute
Client Object
Expects to see "age"
Supports the "age" interface
bottom-up

Figure 12. Meeting in the Middle

This meeting in the middle problem is also present when new business processes and objects are added to an existing object model. The new client objects defined by OOA/OOD often require changes or extensions to the existing legacy server objects.

The developer could change the old objects, but the *existing old* client objects still require the interfaces and semantics. So, as subsequent development occurs either top-down or bottom-up, the functional view defined by the model begins to differ from the implementation view defined by what exists. An incremental approach is required, which involves defining a mapping object with the correct interface. The implementation of the mapping class uses composition to maximally reuse the functions of the old object.

Usually this approach makes heavy use of implementation inheritance and possibly delegation in an attempt to reuse the work done before. The idea is to inherit (and implement) the interface of the objects corresponding to the new functional requirements and extend the implementation of the existing legacy objects.

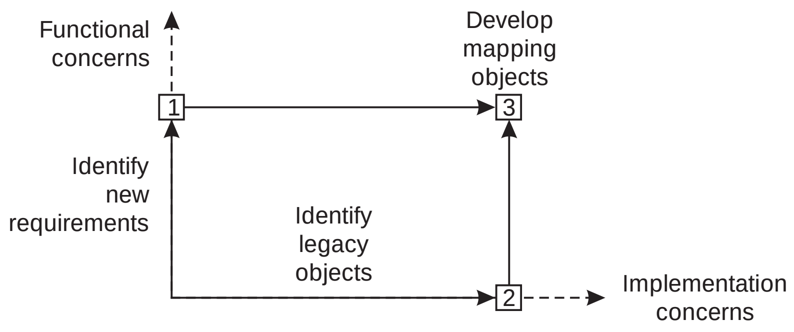


Figure 13. Meet in the Middle

Component Broker supports incremental, meet in the middle development by:

- Supporting both top-down and bottom-up development.
- Composing new classes from existing classes.
- Implementing the new class as an aggregate composed of an instance of the new class and an imbedded instance of the existing class.
- Delegating behavior from the new classes onto the imbedded instance.

Combining the Approaches

Component Broker is designed to fully exploit all three development paradigms, with the following considered to be a typical approach to migrating to and maintaining an object-oriented application:

1. Begin by bottom-up engineering the underlying legacy code and data into data objects and business objects that are contained in application adaptors.
2. Top-down engineer pure object-oriented applications to implement new business behavior.
3. Install the new objects in an application adaptor to support their business behavior with object services such as Concurrency Control, Transactions, Naming, Identity, and Persistence.
4. Use composition tools to build mapping and composed objects that join the bottom-up and top-down classes.
5. Publish the public server classes to client systems by generating client parts that can be loaded into client application development tools, for example, Java and Visual Basic. These client parts delegate their business and service implementations onto the server objects.

Component Broker defines frameworks for supporting the previous tasks and application development tools that facilitate using the frameworks.

Three-Tier Architecture Overview

Component Broker is based on a logical three-tier architecture. That is, the application design space is separated into three tiers (or layers) that serve as the *home*, or server, for objects that encapsulate:

1. Client code and presentation (graphical user interface).
2. Business logic and supporting frameworks and object services.
3. Data access and supporting frameworks and object services.

It is not within the scope of this document to detail the Component Broker architecture, but there are a few points you need to know:

- The tiers are logical, not physical.
- *N* physical tiers are possible, given a logical client/server relationship, with the constraint that *N* is greater than 0 and less than infinity.

Figure 14 on page 9 presents a conceptual overview of the types of objects used in the development of a distributed Component Broker application. On the logical, tier-1 client, there are:

- View objects that provide the end-user interface interactions and mapping onto business objects.
- Client Application objects implement business logic specific to the client and integration with other desktop applications, such as spreadsheets and document processors.
- Component Broker server objects are used on the client through proxy objects. A proxy supports the interface of the server object, but implements the interface by using object remote procedure call (RPC) to call the server object. Component Broker uses OMG's CORBA specification as its distributed object infrastructure.
- The Component Broker client programming model also supports access to business objects in a fashion that matches the client language and programming model. For example, Component Broker provides support for wrappering Component Broker proxies with ActiveX/COM objects to facilitate their use in Visual Basic applications.

On the logical tier-2 server, there are:

- Application objects (AOs) that implement server applications accessed from clients. Application objects are special kinds of business objects that focus on business logic and usage of other business objects analogous to the role performed by some application programs today. The AOs implement concepts such as processes or tasks defined by OOA/OOD. They implement business logic that is not properly modeled as a method on individual business objects. AOs are also the mechanism by which client logic can be moved onto a server to support thin clients.
- Business objects that are built from other business objects and implement a new view or transformation of the existing objects. These business objects transform the existing object model into new concepts that more naturally match the real world and OOA/OOD.
- Business objects that transform data objects to match the requirements of the object-oriented design.
- Data objects implement a direct object-oriented rendering of existing applications and databases.

All server objects are derived from the Managed Object Frameworks, are maintained in an application adaptor and use Component Broker frameworks and services.

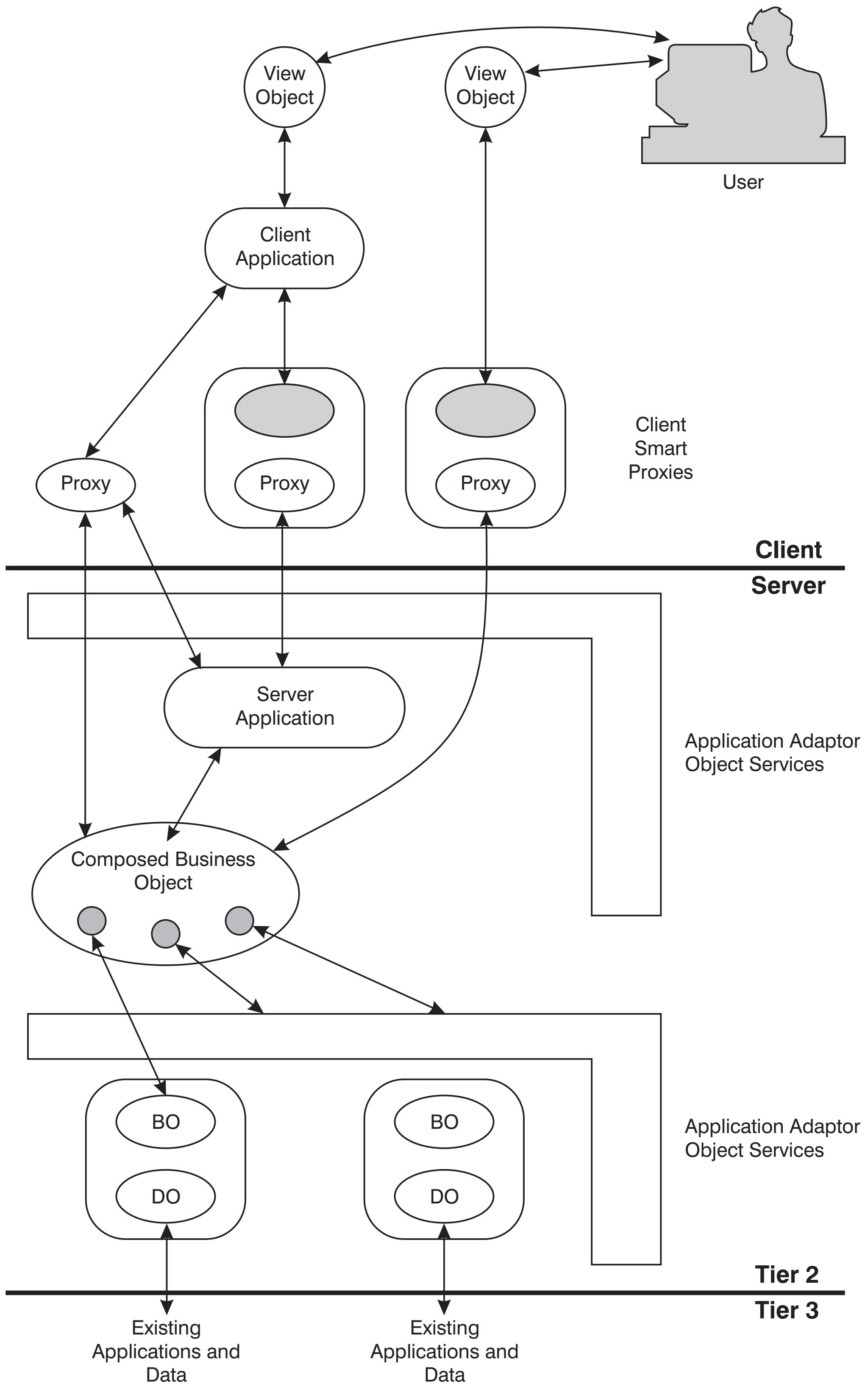


Figure 14. Object Topology

Programming Roles and Responsibilities

The Component Broker programming model defines the following roles and set of tasks performed by each role:

- The Data Object Builder:
  - Implements the data objects that wrapper existing applications and data for bottom-up development.
  - Implements the data objects and scheme definitions that provide persistence and services for data objects that support state data introduced by newly-defined top-down business objects.
  - Implements the data objects in an application adaptor
- The Business Object Builder:
  - Implements business objects that map data objects onto the required object-oriented model.
  - Implements business objects that provide aggregate interfaces or views over other business objects. A simple example is implementing a Portfolio class that provides functions for sets of Checking and Savings objects.
- The Application Object Builder implements business processes or tasks on the server by scripting behavior over multiple business objects.
- A Service Implementor extends the basic Component Broker server by adding new frameworks and object services.
- An Application Adaptor Implementor extends Component Broker by enhancing the functions of an existing application adaptor, or extends a Component Broker server by adding a new application adaptor.
- The Client Part Provider generates and implements smart proxies for specific client environments.
- The Client Programmer builds client side applications by using Component Broker proxies and smart proxies.
- The Interface Builder develops an end-user interface using proxies and client applications.
- The Application Installer packages sets of client and server classes into application DLLs or sets of Java Applets, and configures and installs the applications.

Programming Languages and Conventions

This section discusses the language options for Component Broker Programming, and conventions used in examples in this document.

Interface Definition Language

The primary method for defining Component Broker managed objects is the OMG Interface Definition Language (IDL). OMG IDL is a distributable, language-neutral form for defining interfaces, and can be mapped into almost any object-oriented language and many non-object-oriented languages.

The specific version used is CORBA 2.0. For more information about the syntax, see Appendix B, “Interface Definition Language” on page 271.

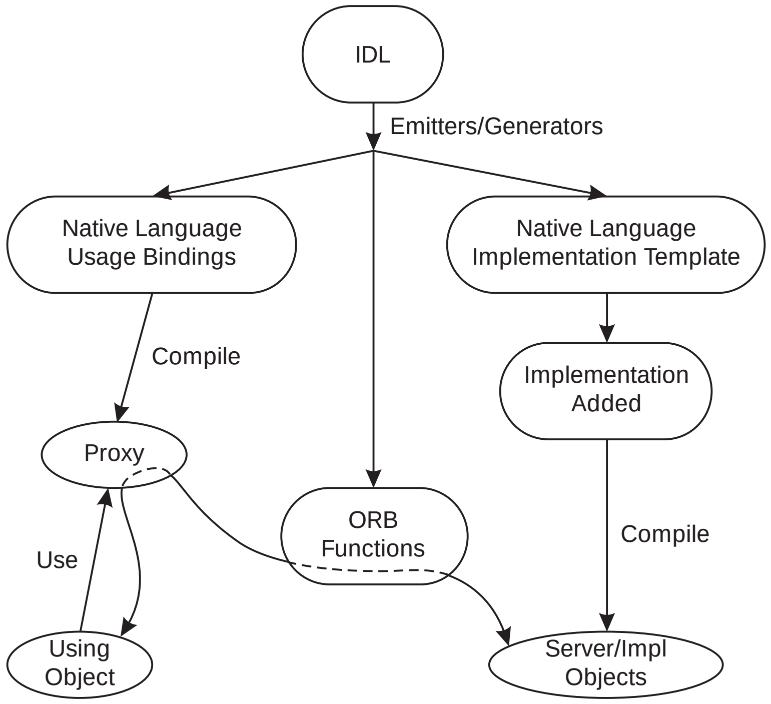


Figure 15. IDL, Usage and Implementation

Figure 15 is an overview of the relationship between IDL and application development languages. Object Providers use IDL to define the interfaces to their objects. The IDL may be directly defined by the Object Provider or may be produced under the covers in application development tools. Code emitters and generators produce the following:

- A *Usage Binding* that provides a native, client language rendering of the IDL. For example as a C++ class or Java Interface. The Usage Binding is also used to generate a proxy object that uses delegation to map the interface onto the server object providing the implementation.
- An *Implementation Template* that provides a native, server language class template into which method behavior can be inserted, for example, by editing the file and adding source code. Adding the behavior to the Implementation Template can be done in a tool.
- Implementation objects such as *skeletons* and *stubs* may also be emitted and compiled by the application development tools, if the client and server are in different processes, or in different languages. These implementation objects provide the functions necessary to make inter-language calls and remote method execution.

One often overlooked point about IDL is that all components in an IDL interface specification are considered to be just that - interfaces. Even attributes in IDL are a shorthand for get and set methods or just get methods for read-only attributes. There is no requirement that an underlying instance variable with the same name must exist.

This separation of interface from implementation is one key to understanding how a language which supports multiple inheritance, a construct not supported in all languages, can be considered to be language neutral.

Java

Java is considered by some to have combined the best features of Smalltalk-like languages (such as a well defined virtual machine interpreter with garbage collection) with those of C++ (the ability to drop out of pure objects for primitives and compiled native methods to get performance).

Component Broker provides support for using Java to develop client applications and server objects, and provides additional value over basic Java by supporting Java as a client application language that has access to Component Broker object functionality from a remote process, and supporting C++ to Java cross-language calls.

C++

Component Broker provides support for developing client and server applications in C++. Component Broker is also extensible, and supports the addition of Application Adaptors and Object Services, as well as tailoring of existing services and application adaptors. C++ is the language initial versions of Component Broker supported for extending Component Broker in these areas.

Other Languages (Windows Only)

Component Broker client applications can be developed in any language for which there are CORBA IDL Usage Bindings. The Component Broker application development tools also emit ActiveX/COM interfaces that wrap the CORBA Usage Bindings, which enables Component Broker client development in other languages such as Visual Basic.

Naming Conventions

Throughout this document and regardless of the language used, the class diagrams and code examples use the VisualAge C++ naming conventions with an additional convention that a class name is assumed to refer to an interface only (not an implementation class), unless it has a suffix of Impl. Therefore, Object as shown in the following figure refers to an interface, but ObjectImpl refers to an implementation (note the use of the interface inheritance symbol):

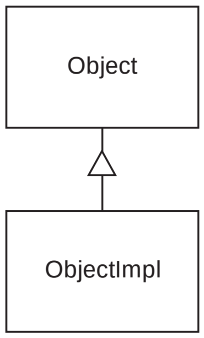


Figure 16. Naming Conventions

The “I” prefacing the interfaces and implementation of Component Broker Objects stands for IBM. All of the interfaces provided as part of Component Broker begin with “I.”

Throughout this book and the rest of Component Broker, there are several conflicting method naming conventions. CORBA uses a convention that separates words with an underscore and uses lowercase while the convention used for Component Broker methods is mixed-case method and class names with no underscores between words. The only exception to this is when Component Broker extends CORBA interfaces with related methods. These are introduced using CORBA conventions.

Coding Conventions

Coding conventions are beyond the scope of this book. However, material related to this topic is presented in Appendix B, “Interface Definition Language” on page 271 and in Appendix C, “C++ CORBA Programming” on page 299.

Another Roadmap

The rest of this book provides you with the information necessary to write applications that use business objects, and to create, test, and install new business objects.

If you need to write C++ CORBA client applications that use a minimum set of the Component Broker-supplied client interfaces to business objects, read Chapter 4, “MOFW Client Programming Model” on page 33. Then see Chapter 6, “MOFW Client Programming Model – Advanced Concepts” on page 83 for additional ways to work with managed business objects in Component Broker.

The basics of developing business objects can be found in Chapter 5, “MOFW Server Programming Model” on page 57. This describes the basic abstractions that are necessary to build business logic and augment it with the necessary logic to be supported by the server.

If you want to develop business objects using Java, see Chapter 10, “Java Server Programming Model” on page 171.

Advanced topics regarding business object development are discussed in Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105.

If the client application requirements specify the use of Java clients, then see Chapter 9, “MOFW - Java Client Programming Model” on page 157. If the client application requirements specify the incorporation of ActiveX components into the client application, then refer to Chapter 8, “MOFW – ActiveX Client Programming Model” on page 143.

When a set of business objects and associated applications have been developed, the next step in the development process is unit testing. This activity is described for client applications and business objects in Appendix D, “Unit Test Environment” on page 333.

The final step is to assemble and install the business objects and associated applications. A detailed description of this process can be found in Chapter 11, “Assembling and Installing Business Objects on AIX and Windows NT” on page 197.

Select and read the chapters pertaining to the roles and responsibilities that you are required to do. Application programmers will take one path, business object builders will take another. Each path includes options depending on how much complexity you desire (or require), and how many Component Broker systems capabilities are needed in your particular solution.

Chapter 2. Personal Life Insurance Application Example

One effective way to illustrate the programming model is through an actual example. For this guide, a Personal Life Insurance Application is used because it is a domain with which most readers are familiar and is complex enough to cover all of the analysis concepts and programming model concepts.

The scenario behind this example is as follows: an insurance company which has been selling business liability insurance decides to go into the personal life insurance market. The company needs new applications to support this new business opportunity, and has decided to develop new object-oriented applications. The new application must leverage a set of existing applications and databases.

The top-down model is presented first, starting with the object-oriented model for the new application and describing the new applications. Then, the constraints imposed by operational reuse of an existing set of applications (CICS Transactions) and databases (DB2 Tables) is described.

The remaining chapters in this document use this example to explain the programming model, as the basis for sample code, and to demonstrate the usage of tools.

The Object Model (Top-down)

Figure 17 on page 16 presents the object model for the new insurance application. The model contains the following classes and relationships:

- The Person object contains information known about people, some of whom may be Customers or Beneficiaries. The insurance company has information about people who are neither Customers nor Beneficiaries. This information was obtained from purchased lists (for example, magazine subscription lists) and other demographic services. The Life Insurance Application ensures that all Customers and Beneficiaries are in the Person database, and updates the database when necessary.
- The Customer object represents information and functions known for Customers of the insurance company. In the scenario, this information was obtained separately from the Person database, and contains separate information. The Customer information was built during years of business, while the Person database was recently acquired. A Customer is associated with an Agent.
- An Agent is associated with zero or more Customers. The Agent is a point of contact for the Customer and receives commission payments.
- A PolicyHolder is both a Customer and a Person, and can have from zero to many Policies. The new object-oriented application enforces the rule that both Customer and Person information must exist for a PolicyHolder. Therefore, when new Policies are created, or existing Policies are modified or have claims processed, any missing information is obtained.
- A Policy is owned by one PolicyHolder; it can have zero or more PayoutFractions, with one per Beneficiary.
- A PayoutFraction is associated with one Beneficiary and one Policy. The PayoutFraction defines the percentage of the Policy's value that is paid to the Beneficiary.
- A Beneficiary is a kind of Person that may hold interests in zero or more Policies (through a PayoutFraction).
- A Claim is associated with a Policy. It represents the request for payment for a Policy, and the state and resolution of this request.

In the terminology of “Three-Tier Architecture Overview” on page 7, the objects described in Figure 17 on page 16 are business objects.

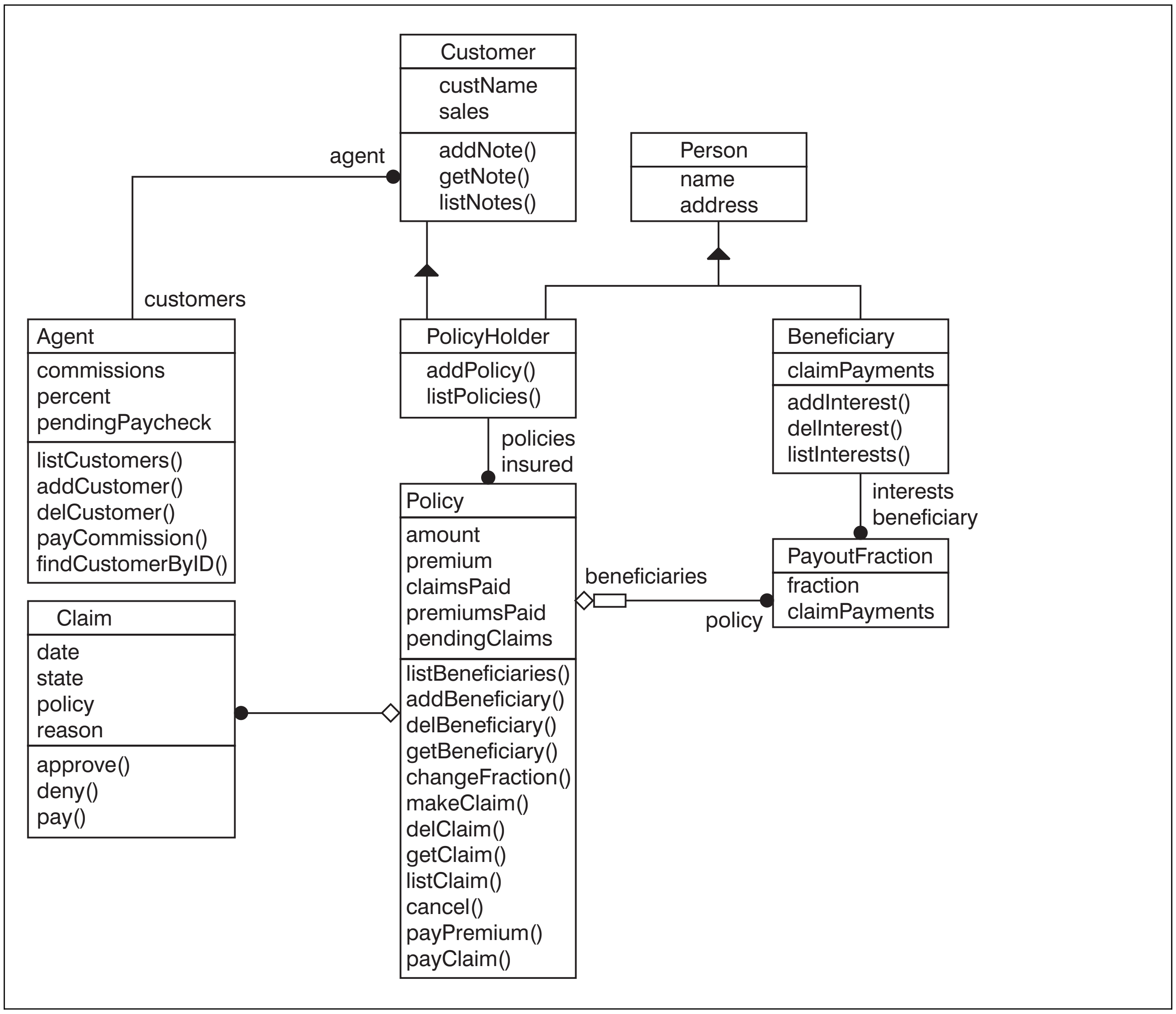


Figure 17. Top-down Object Model

The method and attribute descriptions for each of these objects begins in “Model Details” on page 19.

The Application Model (Meet in the Middle)

In addition to the object-oriented model, the new line of business needs a set of Applications or Processes that implement tasks and functions for users of the Object Model. These Applications implement functions that transform the state of the Object Model by scripting task behavior over multiple business objects.

The Applications or Processes for the scenario are:

CreatePolicy

This application creates a new Policy, and ensures that:

- The Customer exists and is in the Person database.
- All Beneficiaries exist and are in the Person database.
- All PayoutFractions are in the range 0 to 100%, and that their sum is 100%.
- Commissions are paid to Agents.

ModifyPolicy
This application modifies a Policy by modifying:

- Attributes of the Policy, for example, *Value*.
- The PayoutFractions for a Policy, and ensuring that the fractions satisfy the constraints above.
- The Beneficiaries.

CreateCustomer
This application creates a new Customer object, and updates the Person database if necessary. This application associates an Agent with the new Customer.

ModifyCustomer
This application updates information about a Customer.

CreateBeneficiary
This application creates a new Beneficiary, and updates the Person database if necessary.

ModifyBeneficiary
Modifies information about a Beneficiary.

ProcessClaim
This application enters claims, moves a Claim through the states of Entered, Pending and Denied Approved and Paid. When a Claim is Approved, the Policy, Claim and PayoutFractions are marked as Paid.

In the terminology of “Three-Tier Architecture Overview” on page 7, the objects described previously are application objects.

The scripts for these applications are presented in “Applications” on page 26.

Clearly, a truly useful Life Insurance Application must have a richer set of functions, but this set is simple enough for illustrative purposes and covers the main Component Broker programming model concepts.

Design Model (Bottom-up)

Figure 18 on page 18 presents the “Design Model” or constraints on the new object-oriented application. These constraints and design points are:

- The Person class is defined by Operational Reuse of an existing relational database. The Person class is implemented in an object server that front-ends this class.
- There is an existing CICS mainframe based application Customer Management Application that is wrapped by the Customer and Agent classes. These classes are implemented in an object server.
- The Policy, PolicyHolder, Claim, PayoutFraction and Beneficiary classes are new objects whose state data is maintained in a newly defined relational database. In contrast to Person, the state data of the classes defines the database scheme instead of the database scheme defining the state of the classes. There are many possible choices for the persistence mechanism for new classes. Customers, ISVs and SIs are expected to expand the basic Component Broker solution by adding and extending application adaptors.
- The PolicyHolder class is also composed from an instance of Person and an instance of Customer (represented by the dotted line in Figure 18 on page 18).
- The Beneficiary class is composed from an instance of Person.
- The applications CreatePolicy, ModifyPolicy, CreateCustomer, ModifyCustomer, CreateBeneficiary, ModifyBeneficiary and ProcessClaim are implemented on a server.

- Agents and company employees use a GUI (Graphical User Interface) or a Web Browser to view business objects and execute applications. There may also be a set of thick clients that use the Applications and business objects.

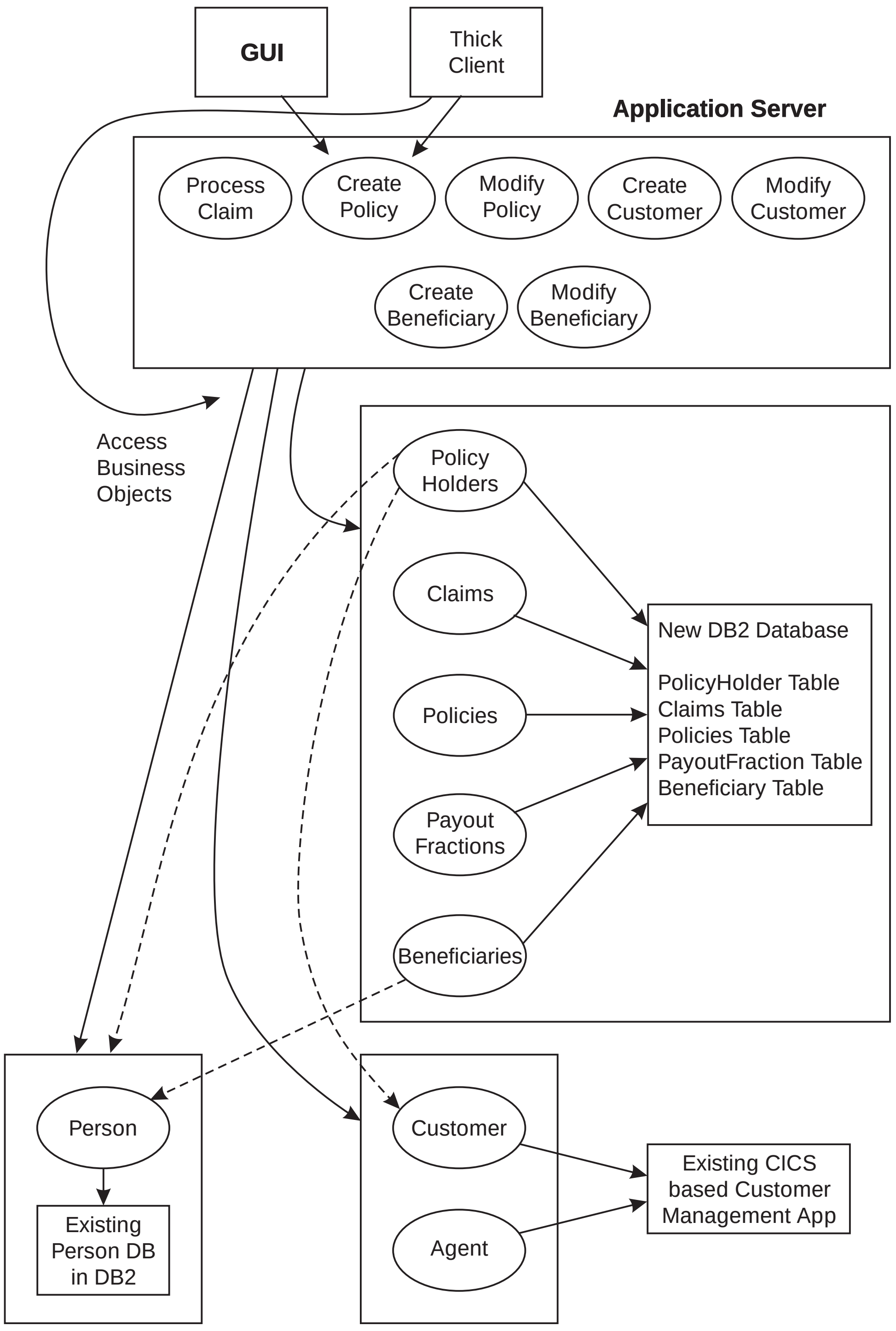


Figure 18. Design Model

The existing applications place some constraints on the object model. For example, both PolicyHolder and Beneficiary subclass from Person. Because there are people who are neither PolicyHolders or Beneficiaries, and because the classes are maintained on different servers, the PolicyHolder and

Beneficiary classes support the Person interface, but implement this interface through composition and delegation onto instances of Person.

In “Design Model (Bottom-up)” on page 17, there are details on the CICS transactions to be reused, and the scheme for the Person database. The scheme for the new databases is defined by the new object model, and the database definition is explained as part of the programming model in Chapter 4, “MOFW Client Programming Model” on page 33 and Chapter 5, “MOFW Server Programming Model” on page 57.

Model Details

This section provides the details of the Object Model, Application Model and the existing CICS Customer Management Application and Person relational database.

Object Model

Object Model includes the following topics:

- “Object Identity”
- “Agent”
- “Beneficiary” on page 20
- “Customer” on page 21
- “PayoutFraction” on page 21
- “Person” on page 22
- “Policy” on page 22
- “PolicyHolder” on page 24
- “Claim” on page 25

Object Identity

One of the most important concepts in an object-oriented model is Object *Identity*. The Identity of an object and its class uniquely identifies an instance of a class. Identity is a critical concept in the Component Broker programming model and for every class there must be a combination of attributes that uniquely identifies an instance of the class. For example, for the Person class, the unique ID is:

```
(ssNo, name)
```

In many cases, the ID must be explicitly defined as an attribute of the object, because no combination of attributes can be guaranteed to be unique. Where a class does not have an explicit ID field, the set of attributes that defines identity is explained.

The Component Broker model does not require that the Identity attributes be a public part of an interface and usable by clients. Identity is needed by Component Broker application adaptors, and to support the OMG Identity Service.

Component Broker introduces the notion of Keys in the programming model to assist in establishing identity and maintaining unique access paths to the business objects. See “Implementing the Primary Key Class” on page 73 for further information.

Agent

The Agent object is reverse-engineered from the existing CICS based Customer Management Application. This object serves as a point of contact for a Customer, and therefore maintains a list of associated customers, as shown in the overview. The following IDL shows the next level of detail:

```
interface Agent
{
       readonly attribute float commissions;
                attribute string name;
       readonly attribute long id;
                attribute  float pendingPaycheck;
                attribute  float percent;

       void addCustomer (in Customer customer, in float bonus);
       void removeCustomer (in Customer customer);
       Customer findCustomerByID (in long customerNo);

       Iterator listCustomers();

       void payCommission (in float amount);
}
```

Attributes	**commissions** The total commissions that have been paid to the Agent this year (does not include *pendingPaycheck*). This is a read-only attribute. **name** The name of the Agent. **id** The ID of the Agent. **pendingPaycheck** The total earned commissions that have yet to be paid to the agent. **percent** The current percentage the Agent receives as a commission.
Methods	**addCustomer** Adds the Customer to the Agent's list of customers and gives a "signing" bonus by incrementing the *pendingPaycheck* attribute. **removeCustomer** Deletes the Customer from the Agent's list of customers. **listCustomers** Lists and iterates through the Customers associated with a given Agent. **findCustomerByID** When a Customer's ID is passed, returns a pointer to the Customer object, if the Agent is associated with this Customer. It returns NULL if the Agent does not manage the Customer. **payCommission** Pays the appropriate commission percentage to the Agent on the amount specified, incrementing the *pendingPaycheck* attribute.

Beneficiary

The Beneficiary is a new object added for this Life Insurance Application. It inherits from Person and adds the association to the PayoutFractions shown in the overview. The IDL follows:

```
interface Beneficiary : Person
{
       attribute float claimPayments;
       readonly attribute long id;

       void addInterest (in PayoutFraction interest);
```

```
        void removeInterest (in PayoutFraction interest);
        Iterator listInterests ();
}
```

Attributes	**claimPayments** The total amount of claims paid to the Beneficiary across all interests in all Policies. **id** The ID.
Methods	**addInterests** Adds the "interest" in the form of a PayoutFraction to the Beneficiary's list of interests. That is, this method updates the Beneficiary to reference the PayoutFraction for a Policy. **removeInterests** Deletes the interest in a Policy. **listInterests** Checks on the interests a Beneficiary may have in various Policies.

Customer

The Customer, like Person and Agent, is an object reverse engineered from an existing application. As with Agent, this class is backed by an existing CICS Customer Management Application.

```
interface Customer
{
   attribute Agent agent;
   attribute string custName;
   attribute float sales;
   readonly attribute long customerNo;
   attribute string ssNo;
}
```

Attributes	**agent** The current Agent in charge of the Customer account. **custName** The name of the Customer. **sales** The total sales accrued by the Customer, that is, the total yearly premiums paid by this Customer. **customerNo** The unique customer ID. **ssNo** The Social Security number of the customer.

PayoutFraction

The PayoutFraction is a new class added especially for this application. The fact that it is attributes only (for a forward-engineered object) is an indicator that it is really more of a “link attribute” (in the manner of Rumbaugh) than a first class object. The IDL follows:

```
interface PayoutFraction
{
   readonly attribute Beneficiary beneficiary;
   readonly attribute Policy policy;
   attribute float claimPayments;
   attribute float fraction;
}
```

Identity	The Object Model allows only one PolicyFraction per pair of Beneficiary and Policy. Thus, the value of these two attributes uniquely identifies the PayoutFraction.
Attributes	**beneficiary** The Beneficiary who is to be paid the specified fraction of the Policy amount. **policy** The Policy in which a Beneficiary has a fractional interest. **claimPayments** The total amount of claim payments made to the Beneficiary for this Policy. **fraction** The fraction of any claim that is to be paid to the specified Beneficiary.

Person

Person represents another reverse-engineered object, this time from an existing relational database. In the example, it is attributes only. However, new business logic could be added to the class.

```
interface Person
{
   readonly attribute string name;
   attribute string street;
   attribute string town;
   readonly attribute string ssNo;
}
```

Identity	A Person is uniquely identified by: `(ssNo, name)`
Attributes	**name** The name of the Person. **street** The street number and name, for example: `30 Saw Mill River Road` **town** The name of the town. **ssNo** Social Security Number.

Policy

The following IDL represents its “static” state (from the Static model), that holds regardless of the dynamic state:

```
interface Policy
{
    readonly attribute PolicyHolder  insured;
    readonly attribute long          policyNo;
             attribute float         amount;
             attribute float         premium;
             attribute float         claimsPaid;
             attribute float         premiumsPaid;
             attribute float         pendingClaims;

    boolean addBeneficiary (in Beneficiary beneficiary, in float fraction);
    void removeBeneficiary (in Beneficiary beneficiary);
    Iterator listBeneficiaries ();
    Beneficiary getBeneficiary (in long id);

    boolean changeFraction(in Beneficiary beneficiary,in float fraction);

    void addClaim (in Claim Claim);
    void removeClaim (in Claim Claim);
    iterator listClaims ();
    Claim getClaim (in long claimNo);

    void cancelPolicy();
    void payPremium(in float amount);
    void payClaim(in float percent);
}
```

Attributes	**insured** The PolicyHolder who is insured by this Policy. This attribute is read-only. **policyNo** The ID of the Policy. **amount** The total amount of the Policy, that is, the maximum amount for which the PolicyHolder is entitled to make claims. **premium** The amount that the Policy Holder is obligated to pay each year in order to keep the Policy In Force. **claimsPaid** The total amount of claims that have been paid so far against the Policy. It is used to determine when the Policy has been paid off, and is also useful in ROI calculations. **premiumsPaid** The total amount of premiums paid against the Policy over its lifetime. This figure is useful for ROI calculations. **pendingClaims** The amount of requested claims still left to be paid to the Beneficiaries if claims are approved.

Methods	**addBeneficiary** Adds a Beneficiary to the Policy with the given fractional amount using a PayoutFraction object. **removeBeneficiary** Deletes the PayoutFraction from the Policy's list of beneficiaries and the Beneficiary's list of interests. **listBeneficiaries** Lists the PayoutFractions that show the Beneficiaries "interested" in this Policy. **getBeneficiary** Gets a Beneficiary by ID, if associated with this Policy. **changeFraction** Changes the PayoutFraction associated with a given Beneficiary. **addClaim** Associates a claim against the Policy by incrementing the *pendingClaim* attribute and creating a linkage between the Claim and Policy. **removeClaim** Removes the association between a Policy and Claim. **listClaims** Enables iteration through a set of Claims. **getClaim** Gets a claim by ID. **payPremium** Pays a specified amount towards premiums by incrementing *premiumsPaid*, either against the Policy In Arrears, or In Force. The Policy remains In Force or is placed In Arrears depending on whether the premium payments are up to date. In any event, premiums are not paid except while the Policy is In Force or In Arrears. **payClaim** Records that a Claim was paid against this Policy. **cancelPolicy** Moves the Policy into the cancelled state.

PolicyHolder

A PolicyHolder represents the Customer of a Personal Life Insurance application. Most of its data comes from the Customer and Person objects, and only the additional behaviors associated with a PolicyHolder are shown here.

```
interface PolicyHolder : Customer, Person
{
   void removePolicy (in Policy policy)
   void addPolicy (in Policy policy);
   Iterator listPolicies (in string query);
   Policy getPolicy (in long policyNo);
}
```

Identity	Policy Holder derives its identity from Customer.

Methods	**removePolicy** Removes a policy associated with a PolicyHolder. **addPolicy** Allows a new Policy to be associated with the target PolicyHolder. At present, the system allows for multiple Policies to be associated with a single PolicyHolder. **listPolicies** Allows multiple Policies associated with a given PolicyHolder to be sequentially accessed through an iterator. **getPolicy** Gets a Policy by its ID, if associated with this Policy Holder.

Claim

A Claim tracks the state and processing of a request for payment on a Policy. Its interface is:

```
interface Claim
{
   readonly attribute Policy  thePolicy;
   readonly attribute long    claimNo;
   readonly attribute string  date;
            attribute enum    state;
            attribute string  explanation;

   void approve (in string explanation);
   void deny (in string explanation);
   void pay ();
}
```

Identify	The claim is identified by the claimNo.
Attributes	**thePolicy** The Policy for the Claim. **claimNo** The unique Claim number. **date** The date the Claim was created. **state** The current state of the Claim. The state transition diagram for the Claim is shown in Figure 19 on page 26. **explanation** Additional information about the Claim.
Methods	**approve** Mark the Claim approved and set an explanation or comment. **deny** Mark the Claim denied and set an explanation. **pay** Mark the claim as Paid.

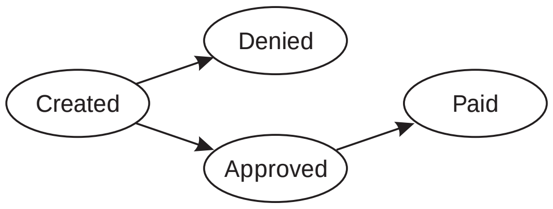


Figure 19. Claim States

Applications

This section provides the pseudo-code and business logic for the applications.

Notes:

1. These applications are conversational, and involve many interactions between the end-user and the running application.
2. These applications are transactions and follow typical transaction guidelines for data integrity. On Commit, all databases are updated. On Abort, all changes are removed.

Create Policy

An Agent uses this application to create a new Policy. This application is defined by the following rules:

Pre-conditions	• The PolicyHolder must exist
Post-conditions of Successful Completion	• There will be a Person object for the Customer. • The Policy exists and has all attributes set to consistent values, that is, policyNo and PolicyHolder.
Invariants	The Customer is unchanged.
Application Logic	1. The PolicyHolder is identified. 2. A Policy is created and the attributes are set.

ModifyPolicy

Pre-conditions	The Policy must exist.
Post-conditions of Successful Completion	• The Policy exists and has all attributes set to consistent values, that is, policyNo and PolicyHolder. • The PolicyHolder exists and is in a consistent state.
Application Logic	1. The Policy is identified. 2. Policy attributes are updated.

Create Customer

Pre-conditions	The Customer must not exist.
Post-conditions of Successful Completion	• Customer exists and has attributes set. • Person exists and has attributes set.

Application Logic	1. The Customer's name is obtained 2. A Customer may or may not have an entry in the Person database. The Customer and Person classes are tied together by a note associated with the Customer through the Customer Management Application. 3. If the Customer or Person is missing, it is created.

Modify Customer

This application enables modification of a Customer and the related Person information. It creates the Person if one does not exist and creates the binding between Customer and Person if one does not exist.

Create Beneficiary

Pre-conditions	The Beneficiary must not exist
Post-conditions of successful completion	• The Beneficiary exists. • The Person exists and is linked with the Beneficiary.
Application Logic	1. The Beneficiary's name is entered. 2. The Beneficiary is created if it does not exist. 3. The Person is created if it does not exist.

Modify Beneficiary

This application updates the Beneficiary and Person objects.

Process Claim

Pre-conditions	• The Policy must exist. • The PolicyHolder exists. • All Beneficiaries exist.
Post-conditions of Successful Completion	• If this is a new Claim: – A Claim is created. – The Claim is associated with a Policy. • If the Claim exists and is in the Created state: – The Claim may be Approved with an explanation. – The Claim may be Denied with an explanation.
Invariant	Existence of objects is unchanged.
Application Logic	1. If this is a new Claim: a. The Policy is identified. b. A Claim is created. c. The Claim state is Entered. d. The Claim and Policy are associated. 2. If an existing Claim is being processed: a. The Claim is retrieved: • If the claimNo is known, the Claim is retrieved by number. • If the claimNo is not known, the Policy is retrieved and the Claim is retrieved by iterating through the associated claims to find the Claim with the desired properties. b. If the Claim is Entered, it may either be Approved or Denied. c. If the Claim is Approved, it may be Paid. d. If the Claim is Paid or Denied, it may only be viewed.

Key Observations

The main points of this example are to demonstrate that:

- Both state data and attributes of objects can be derived from the in/out parameters of existing applications.
- Many object methods can be mapped onto existing applications.
- Reuse of applications may often involve complex mappings between transactions, menu states, and screens. Often, the mapping is simple and almost one-to-one between methods and transactions and attributes and the fields on display or update screens.
- Even if the data is not maintained in an RDBMS, the application usually provides support for limited query through selected attributes and iteration through the results. This is most often accomplished by simple query operators on keys.

Chapter 3. The Managed Object Framework

This chapter introduces the Managed Object Framework (MOFW) and its positioning within Component Broker. A systematic approach to explaining the details of the MOFW follows in succeeding sections.

The MOFW represents the set of interfaces, implementations, and conventions that must be followed in order to create and use business objects in Component Broker. The MOFW provides capabilities above and beyond those present in the basic CORBA ORB and object services defined by OMG. MOFW also provides simplified interfaces to some of the basic CORBA interfaces. The MOFW is not the only set of interfaces supported by Component Broker. Component Broker allows for additional frameworks which can be used by business objects and client programs.

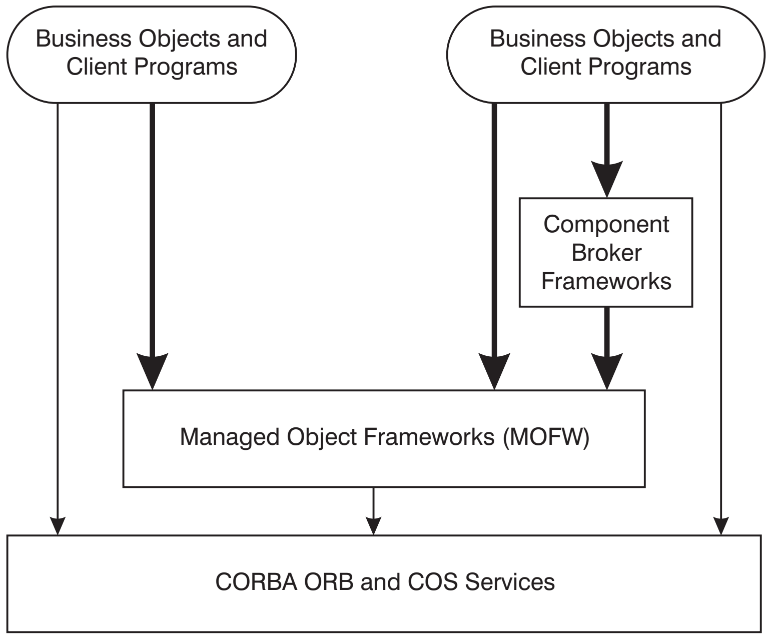

Business Objects and Client Programs
Business Objects and Client Programs
Component Broker Frameworks
Managed Object Frameworks (MOFW)
CORBA ORB and COS Services

Figure 20. Component Broker and MOFW Framework Overview

As Figure 20 shows, business objects and client programs that use business objects can be written directly to the MOFW interfaces. The MOFW is not a complete layer over the CORBA services. It adds usability and function only in those places key to providing an integrated object server. Sometimes the existing CORBA ORB and object services provide what is needed to implement the Component Broker server vision. In some cases, the Component Broker Frameworks provide simpler access to the server. The object provider rather than the applications programmer chooses the interfaces. The Component Broker Frameworks are not, however, a complete encapsulation of the MOFW interfaces. Be careful when mixing these sets of interfaces together. This document provides guidance on which combinations of Component Broker Frameworks and the MOFWs make sense.

Examples of the abstractions found in the Managed Object Frameworks (MOFW) include:

- IManagedClient::IHome
- IManagedClient::IManageable
- IManagedLocal::ILocalOnly
- IManagedLocal::INonManageable
- IManagedLocal::IKey, IUniqueKey and IPrimaryKey
- IManagedLocal::IHandle
- IManagedServer::IManagedObject

- IManagedServer::IManagedObjectWithDataObject
- IManagedServer::IManagedObjectWithCachedDataObject
- IManagedServer::IDataObject
- IManagedCollections::IReferenceCollection
- IManagedCollections::IKeyedReferenceCollection
- IManagedCollections::IIterator and IManagedCollections::IMIterable

Managed and Non-Managed Objects

There are two kinds of objects in Component Broker, those that are managed by a Component Broker server and those that are not. All of the objects that client application programmers and business object builders deal with descend, directly or most often indirectly, from either IManagedLocal::ILocalOnly or from IManagedClient::IManageable. Component Broker has these two specific kinds of objects to ensure a minimum footprint client, separate server-only objects from those that may exist on clients and servers, and, most importantly, simplicity for the programmer. No extra methods need to be used or implemented based on this separation.

Those objects that are to be local-only are descendants of ILocalOnly or INonManageable. Those objects that are to be accessed remotely and managed by a Component Broker server are subclasses of the IManageable interface. Figure 21 shows the basic relationship between these MOFW interfaces and the CORBA object services interfaces.

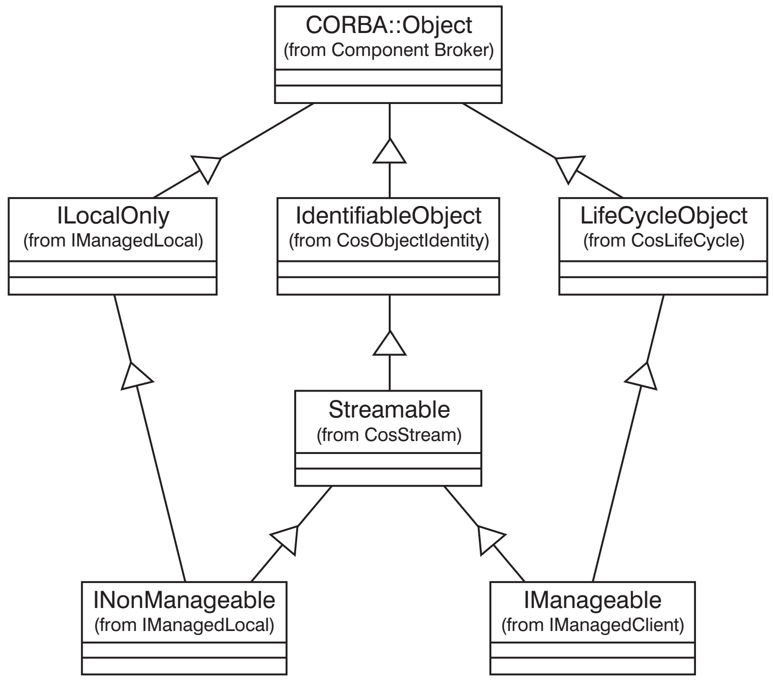


Figure 21. MOFW Basic Abstractions

The INonManageable interface comes from a module named IManagedLocal while the IManageable interface comes from the IManagedClient module. IManagedLocal, is explained in “Configuring Managed Objects into Servers” on page 254. The Local-Only Development Process contains other abstractions that are local-only objects. While these are not to be accessed remotely, Component Broker made these descendents of CORBA::Object and treats them as much like CORBA objects as is practical for local-only objects. INonManageable extends ILocalOnly by introducing methods that assist developers in making “stringified” versions of ILocalOnly objects.

IManagedClient is a set of abstractions that business object clients need to understand to some degree, and interact with when writing applications that use Component Broker business objects. Object providers subclass from, and often implement, abstractions in IManagedClient while client and applications programmers call some of the methods introduced by these abstractions.

Both of the key abstractions shown at the bottom of Figure 21 on page 30 also get some additional interface from CORBA object services. Both INonManageable and IManageable are descendants of CosStream::Streamable, implying that they are also Identifiable objects. IManageable objects are also LifeCycle objects. This is described in “Implementing the IManageable Required Methods” on page 67. The important fact is that there are two kinds of objects to be remembered, and that each has a slightly different ancestry and purpose in Component Broker.

All Component Broker objects inherit from CORBA::Object. This inheritance from here forward is not generally shown. INonManageable has a notable parent ILocalOnly. Component Broker programmers can always tell if a particular object is local-only or accessible remotely. IManageable also implies a lot more. It means that descendants of this base class get not only remote accessibility, but also a full range of services from the Component Broker server. The Component Broker server takes care of the IManageable objects, allowing them to be persistent, accessed securely, participate in transactions, and take advantage of all of the additional Component Broker server features.

You need INonManageable and ILocalOnly because there are some internal Component Broker objects are ILocalOnly subclasses but do not need the additional interfaces implied by INonManageable.

Note: A Component Broker object cannot inherit from both IManageable and INonManageable. It can have only one of these parents.

All Component Broker objects are described in IDL for documentation and consistency. Each programming language that has a usage binding can be used to interact with Component Broker MOFW-based objects. For example, a C++ programmer deals with descendants of IManageable through a C++ usage binding to the IManageable's capabilities.

In addition to these two kinds of objects, any application using Component Broker also uses native language objects. A mixture of native language objects and usage bindings to Component Broker objects make up a particular application. Component Broker supports language usage bindings for Java and C++ and also provides special support for client programmers using ActiveX programming tools.

Using Component Broker MOFW-based objects is done with usage bindings as described previously. Building Component Broker MOFW-based objects is different. Component Broker features the construction of business objects using C++ or Java. The IBM ORB and emitters provide the basis for the implementation bindings that are used to construct implementations for the Component Broker MOFW-based business objects.

Understanding MOFW Objects

All MOFW-based objects, both IManageable and INonManageable, inherit from CORBA::Object, even though this might not be apparent from the IDL.

IManageable objects:

- Are always created through Homes, that are located using Factory Finders.
- Are accessible remotely through client usage bindings in Java and C++.
- Are implemented using C++ or Java.
- Are always managed by an application adaptor and reside on a Component Broker server.

INonManageable objects:

- Are always created using a static create method of the form *className*::_create(), a method generated by the IDL compiler.
- Look and behave a lot like regular native language (Java or C++) objects.
- Are not managed by an application adaptor. They might be used on the server, transiently, but they are not persistent.
- Are written in C++ for use by the server and C++ client. They also must be written in Java if the Java-client feature of Component Broker is to be used.
- Are developed using a special local-only development process. The emitters only generate the subset of the various bindings that are necessary to support local use. (For example, no dispatcher code, no server-side bindings, and so forth.) This is explained in more detail in “The Local-Only Development Process” on page 252.

Chapter 4. MOFW Client Programming Model

This chapter defines the Component Broker Managed Object Framework (MOFW) client programming model. The word *client* generally refers to a particular computer in a distributed environment and the relationship of that computer with other computers. In this context, the word client refers to any program and its relationship to a business object. In other words, a client is any program executing in a process on a client or server computer.

Therefore, the client programming model explains how you can use business objects and how to develop objects that are clients of Component Broker business objects. Application programmers developing *tier-1* (client) or *tier-2* (server) applications use the client programming model when they use a business object in the implementation of a new object. Client programmers use server objects to develop new applications by composing new application objects and scripting behavior over sets of server objects.

A Component Broker server process implements managed objects that are used by client applications to perform business functions. All Component Broker server objects are derived from the Managed Object Framework. The term *managed object* is used because one of the Component Broker server product's main contributions to object-oriented applications is the implementation of run-time management functions used in the implementation of server objects. Some examples of these management services are:

- Persistence, Transactions, and Security.
- Workload management and availability management over multiple servers.
- Object-oriented access to existing databases and applications.

This chapter provides details on a client view of the structure of a Component Broker client/server application.

- "Client View of Component Broker Applications" contains details on how a client object uses a Component Broker business object and explains what occurs if this business object is implemented in a different language on a different system.
- "Client Programming Model: Basic Tasks" on page 35 contains a quick overview of the client programming model. This section explains what you need to know to get started developing a Component Broker client application. Specifically, it explains how to accomplish common programming tasks, and it contains sample code segments.
- "Summary: The Client Programmer's Check List" on page 54 contains a check list of information a client application programmer needs to start developing the client side of a Component Broker application.

Component Broker supports writing client applications in C++, Visual Basic, or Java. This chapter uses C++ for the examples. For information on writing client applications in Visual Basic, see Chapter 8, "MOFW – ActiveX Client Programming Model" on page 143. For information on writing client applications in Java, see Chapter 9, "MOFW - Java Client Programming Model" on page 157.

Client View of Component Broker Applications

Figure 22 on page 34 presents a high-level overview of the client programming model. The way clients deal with business objects at the programming model level is consistent regardless of the underlying support mechanisms used to build those business objects.

The client application accesses *business objects* on the server. Business objects are instances of classes that implement the business logic of the application. Business objects support the interfaces of the Managed Object Framework which allows business objects to be installed onto, and managed by, a

Component Broker server. A business object that is installed on a Component Broker server is referred to as a *managed object* (MO).

If the managed object is in the same process and same language as the client application, the client directly accesses the object. If the object resides in another process or on another machine, the client uses a CORBA *proxy* object. This proxy object:

- Implements the interface of the managed object in the language and process of the client.
- Uses the ORB and CORBA object RPC to relay method calls to the managed object in the remote process.

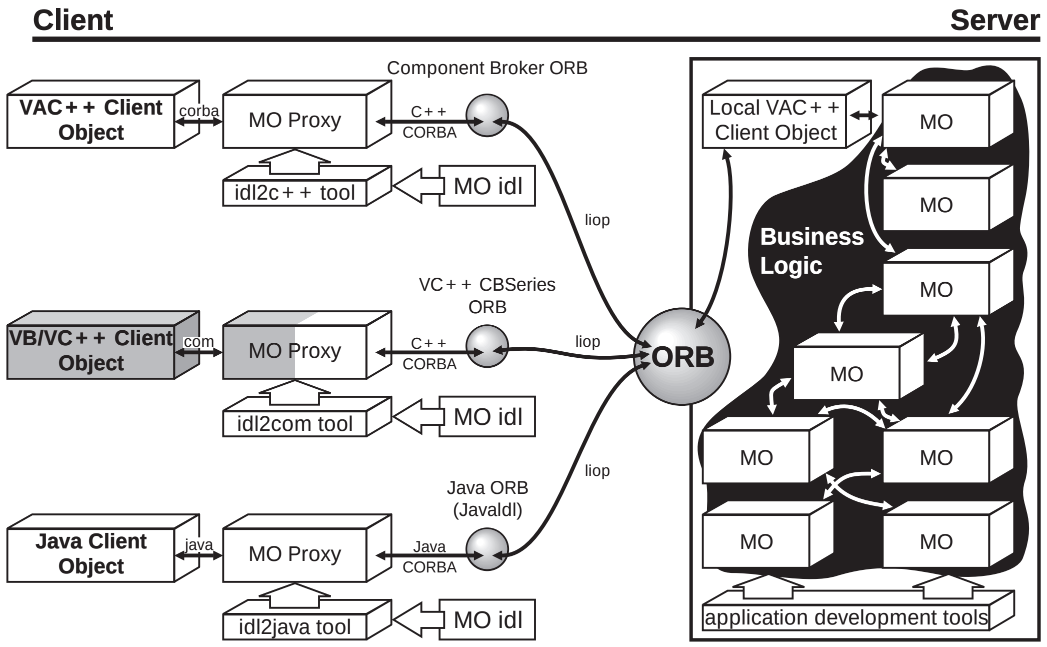


Figure 22. Client Model Overview

Component Broker provides functions that enable the development of ActiveX/COM *wrappers* for Component Broker managed objects. The wrappers implement an ActiveX/COM rendering of the interface of the managed object. Within the COM object is a Component Broker proxy, and this proxy delegates business logic calls from the COM object on the client to the Component Broker managed object on the server.

The *managed object implementor*, sometimes referred to as a *server application*, provides the client application with:

- A set of interface files that define the interface to a managed object and any *helper* class the client uses. Interface files are provided for each client programming language; for example, .hh files for C++ and .java files for Java. In addition, Interface Definition Language (IDL) is provided for the managed objects and helper classes.
- DLL and Java .class files that implement the classes in the interfaces and the helper classes. These include the proxy classes that enable remote/local object use.

To use an object-oriented application (a set of managed objects) you need to understand its object model. Specifically, you need to understand the interfaces and its methods and attributes.

The basic client programming model assumes that you have compile time access to the server interface definitions (.idl files) and has documentation on the class relationships and the semantics of methods and attributes. The artifacts provided by the object provider define the interface syntax. What the interface does is provided in comments and in the documentation.

Component Broker supports a remote/local programming model for managed objects. This is implemented by run-time libraries, proxy classes, and application development tools. The managed object implementor defines the interface to the managed object using CORBA IDL. The Component Broker application development tools emit the necessary language bindings and proxies to support the managed object from remote applications.

The Component Broker Managed Object Framework allows object providers the opportunity to present a number of programming interfaces to clients. Determining which client interfaces to use is an advanced skill in client programming and not critical to getting started. The kinds of client programming interfaces provided and tips on how to select the correct one for a particular application are described in “Expanding the Client Programming Interface” on page 246.

For this chapter, the container-type independent, business logic-only interface is used. This interface is the simplest with which to get started. The examples in this document use interface names like “claim” and “policy.” Using the client bindings for these interfaces provides access to all the methods that the object provider defined for client usage.

Client Programming Model: Basic Tasks

A client application can perform the following tasks:

- Find an object.
- Use an object.
- Create an object.
- Use a set of objects.
- Remember an object.
- Release or delete an object.

The following sections present a quick overview and samples of how a client application performs these tasks. However, before a client application can perform any of these tasks, it must initialize the Component Broker client environment.

Initializing the Client Environment

The Component Broker provides a convenient interface for initializing a Component Broker client. This interface is CBSeriesGlobal.

Initializing a Component Broker client requires only the following single line of code:

```
  CBSeriesGlobal::Initialize();
```

After initialization, the client has access to the static methods CBSeriesGlobal::orb() and CBSeriesGlobal::nameService() that return references to objects of type CORBA::ORB and IExtendedNaming::NamingContext, respectively.

The CBSeriesGlobal interface is provided as a convenience to client programmers. A Component Broker client could define a set of CORBA calls that encapsulate the Initialize() method as follows:

```
 interface CBSeriesGlobal
{
   void Initialize();

   CORBA::ORB orb();
   IExtendedNaming::NamingContext nameService();

};
```

This IDL is directly implemented in each client programming language (C++ and Java). The C++ interface is:

```
class CBSeriesGlobal
{
   public:
      static void Initialize();

      static CORBA::ORB_ptr orb ();
      IExtendedNaming::NamingContext_ptr nameService ();
};
```

Initialization and Object References

CBSeriesGlobal is a convenience interface that is not required for all client programs. However, if the client program uses either a CopyHelper or PrimaryKey that contains an object as one of its attributes, then initializing CBSeriesGlobal is a requirement. This is because the implementation of the CopyHelper and PrimaryKey depend on CBSeriesGlobal:orb() when using the ORB object_to_string() operation.

Note: CBSeriesGlobal was developed as a convenience function and is not coded to the CORBA programming style. This interface must be used carefully. See the MOFW section of the *Component Broker Programming Reference* for more information.

Navigating the Name Space Using the Naming Service

The Component Broker name space is hierarchical and similar in structure to a file system directory tree. As the nodes in a directory structure are files (either directories or leaf files), the nodes of the Component Broker name space are CORBA::objects (either NamingContexts objects or leaf objects). A NamingContext is an object that contains zero or more bindings of string name-object reference pairs. Each object, bound by name into a context, can be a leaf object or a subordinate NamingContext in the tree. Subordinate NamingContexts similarly can contain bindings of other NamingContexts and leaf objects.

Component Broker introduces the interface IExtendedNaming::NamingContext as an extension of the OMG defined CosNaming::NamingContext interface. This interface provides simplified methods for binding, unbinding, and resolving name-object pairs in the name space. Specifically, the NamingStringSyntax::NameString type allows programmers to specify the name space path in a string format similar to the way they use strings for specifying directory paths. For example:

```
"name1/name2/name3"
```

where:

name1	Identifies a NamingContext contained in the current NamingContext (and bound to "name1").
name2	Identifies a NamingContext within *name1*.
name3	Identifies a leaf object or another naming context bound in *name2*.

The interface for IExtendedNaming is:

```
module IExtendedNaming
{
    // This type resolves to a string that maps to a char*
    typedef NamingStringSyntax::NameString Name;

    interface NamingContext : CosNaming::NamingContext
    {
        void bind_with_string (in Name n, in CORBA::Object obj)
            raises (NotFound, CannotProceed, InvalidName, AlreadyBound);

        CORBA::Object resolve_with_string (in Name n)
            raises (Not Found, CannotProceed, InvalidName);

        void unbind_with_string (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
};
```

The OMG Naming Service specification defines only the interface to the Naming Service and does not define any structure for the name space. Figure 23 provides an introduction to the Component Broker name space structure.

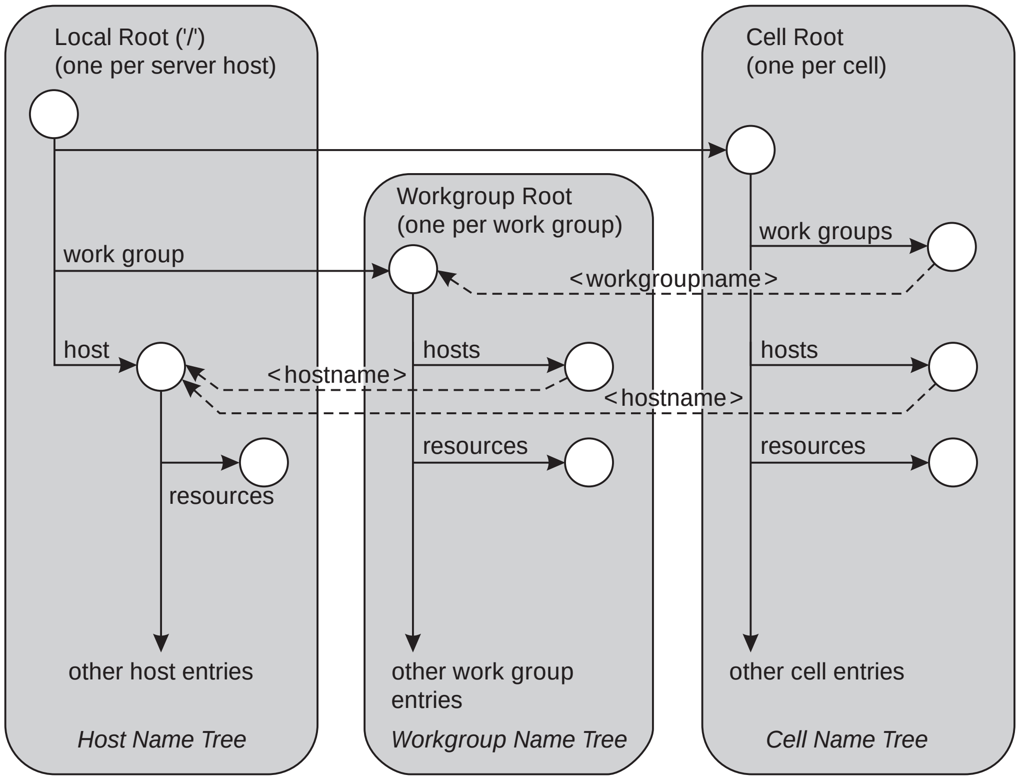


Figure 23. The Name Space Structure

This structure defines the top level (root) naming contexts that exists in an enterprise (cell). Every host system participating in the network has a Local Root NamingContext. This is the NamingContext that is made available to applications through the CBSeriesGlobal::nameService() static method. This context is the anchor from which applications navigate the name space. Each host system belongs to one and only one work group and one cell.

A work group represents a subset of the systems participating in the network. The Workgroup Root NamingContext can be navigated from the Local Root NamingContext by resolving names prefixed with `workgroup/<rest of path>`. The NamingContexts of other systems within the work group can be navigated from the Local Root NamingContext by a path that traverses the work group NamingContext if the hostname of the desired system is known. For example, the name `workgroup/hosts/<hostname>/<rest of path>` provides access to the NamingContext of another host within the work group.

A cell represents the set of all systems and work groups participating in the network. The Cell Root NamingContext can be navigated from the Local Root NamingContext by resolving names prefixed with `.:/<rest of path>`. As can be seen from Figure 23 on page 37, the NamingContexts of any system or work group in the network can be navigated from the Local Root NamingContext by a path that traverses the Cell NamingContext.

The resolve_with_string() method returns a CORBA::Object. Before a client can use the returned reference as a NamingContext or as a specific class of leaf object, the client must narrow the object to the desired class.

Finding a Managed Object

When a client application starts, it has no knowledge of any objects. In a non-distributed environment, the application would create any objects it needs. These newly created objects would reside on the same system as the application. If the application needs these newly created objects to be available the next time the application runs, the application itself is responsible for making these objects persistent.

However, Component Broker is a distributed environment. In this environment, objects are not created on a client system. Objects are created from a client system on a server system. Object persistence is managed with the help of the Component Broker run time.

There are two ways for a client application to find a managed object:

- Use the Naming Service.

  If the object has a *Name*, the client can use the Naming Service to locate the object by its *Name*. In general, the Naming Service only contains a subset of the objects in a distributed system. This subset consists of well-known objects, such as collections of business objects or important objects in the object model.

- Use another object.

  If the object does not have a *Name*, the client can find the object by using the Naming Service to find a well-known object, such as a *factory* or collection, and then use this object to execute methods that return the object as a return value or output parameter.—

Both techniques involve the use of the Naming Service. In the Component Broker distributed environment, all work with objects in a client application must begin by using the Naming Service.

Finding a Managed Object Bound in the Naming Service

Assume that the insurance company application example contains a few important Claim objects in the Naming Service and that the client application is written with the knowledge that these Claim objects can be located at a known path in the name space. The following code segment shows how to find the Claim object, belonging to a customer named “Lou.”

```
{
   CORBA::Object_ptr temp_ptr;
   Claim_ptr louClaim_ptr;
   try
   {
      temp_ptr = CBSeriesGlobal::nameService()->resolve_with_string(
                 ".:/Applications/LifeInsurance/Claim/LouClaim");
      louClaim_ptr = Claim::_narrow(temp_ptr);
      CORBA::release(temp_ptr);
      // use louClaim_ptr in various calls
      CORBA::release (louClaim_ptr);
   }
   catch(...)
   {
      // appropriate error recovery
   }
}
```

The path ".:/Applications/LifeInsurance/Claim/LouClaim" follows a path that traverses a NamingContext bound into the Cell Root NamingContext with a name of Applications and so forth to the desired LouClaim object.

The following code segment shows how the insurance application could have originally bound the LouClaim object.

```
   Claim_var louClaim;

  // Create Lou's Claim and initialize it prior to the following code segment
     ...
  // Try to bind the 'louClaim' object into the name space.

  try
  {
      CBSeriesGlobal::nameService()->bind_with_string(
             ".:/Applications/LifeInsurance/Claim/LouClaim", louClaim);
  }
  catch(...)
  {
      // appropriate error recovery
  }
```

For additional information on the Component Broker Naming Service, see References in the Component Broker Online Documentation.

Finding a Managed Object Using the PrimaryKey Helper Class

What if Lou's Claim is not in the Naming Service? How do you find a specific claim? This section explains how.

Homes are instances of the IHome class. You might decide to implement and provide a tailored subclass of IHome, or you might use an instance of the base class. The relationship between managed objects and collections is explained in "Using Sets of Objects" on page 44. For now, all that is important to know is that a home can be used to find objects that were previously created by that home.

As stated previously, a client application always starts with the Naming Service to find its first object. As such, it might seem reasonable to assume that a client application would find a Home directly in the name space. However, the LifeCycle Service provides a facility for finding Homes called FactoryFinder. Therefore, a client application uses the Naming Service to find a FactoryFinder, which is then used to find a Home. FactoryFinders are discussed again in more detail in "Creating a New Object – Create From Key" on page 42.

To continue the example, the following code segment finds the Home for Claim objects.

```
CORBA::Object_var obj;
IExtendedLifeCycle::FactoryFinder_var ff;
IManagedClient::IHome_var claimHome;
try
{
   obj = CBSeriesGlobal::nameService()->resolve_with_string(
                  "host/resources/factory-finders/host-scope");
   ff = IExtendedLifeCycle::FactoryFinder::_narrow(obj);
   obj = ff->find_factory_from_string("Claim.object interface");
   claimHome = IManagedClient::IHome::_narrow(obj);
   // use claimHome
}
catch(...){
   // appropriate error recovery
}
```

Now you need to find Lou's Claim. Assume that Lou is on the phone and can tell you his claim number. In the example, the Claim's *claimNo* attribute uniquely identifies an instance of the Claim class, and is a primary key into the Claim Home. To facilitate the usage of a generic home, the Component Broker programming model introduces two types of Helper Class: PrimaryKey and Copy. The Copy Helper Class is discussed further in “Creating a New Object – Create From Key” on page 42. Every managed object class has a primary Key Helper Class that lets you find (and create) objects of that type. An instance of a Key Helper Class is always local to the client's process and language. Key Helpers, like all helper classes, are created with a static method on the class named _create(). This static method gets generated in the usage bindings of all interfaces that specify the *localonly* pragma in their IDL files. The same rule is in place for copy helpers and objects of other classes that are described in “Local-Only Development Process” on page 78.

When you create an instance of a primary key, the key must be set by one or more attributes on the primary key object. When all of the key attributes have been set, the primary key object is now usable. The Claim Home uses this primary key to find the previously created Claim object. Remember, the PrimaryKey is on the client system, but the Claim object and the Claim Home are on the server system. If the client passes a primary key object as a parameter to the Home, and the Home is on a remote system, the remote system might get a proxy back to the client's PrimaryKey instance. This would turn the client into a server and unpredictable results could occur. Therefore, the Component Broker programming model uses strings as the method for passing keys to potentially remote objects.

Using a PrimaryKey Helper Class to Find an Object

Continuing the example, the following code segment would find Lou's Claim in the Home (assuming Lou told you his number is 1234).

```
// Create an instance of the Key Helper Class
ClaimPrimaryKey_var claimPrimaryKey = ClaimPrimaryKey::_create();

// Set the claimNo attribute in the key
claimPrimaryKey->claimNo(1234);

// Must convert key object to a string to go over the wire to the server
ByteString_var claimString = claimPrimaryKey->toString();

IManagedClient::IManageable_var  temp_var;

// Call find by key on the Home to find Lou's Claim
temp_var = claimHome->findByPrimaryKeyString(claimString);
```

```
// Narrow type to Claim
Claim_var louClaim = Claim::_narrow(temp_var);

// continue to use louClaim
```

The object provider of a public managed object always provides you with a set of helper classes for using the Homes that contain his managed objects. There is always exactly one primary key helper class. The object provider gives you:

- The interface definitions for the primary key class.
- An implementation of the primary key class.
- Documentation for its use.

The previous code segment creates a new instance of the Key Helper Class ClaimPrimaryKey using the static _create() method, and sets the *claimNo* attribute to `1234`. Then it creates a string version of the key and finds Lou's Claim using the Claim primary key and the Claim Home.

The Component Broker programming model mandates Key Helper Classes for managed objects to make things easier for you. The keys are passed as strings. You need to use the available set*xxxx*() methods on the helper class to prepare the key information. No string manipulation to concatenate pieces of multi-valued keys is necessary. Using a Key Helper Class enables type errors to be detected at compilation time, as well as provides knowledge of the field ordering and algorithm for defining a key.

Finding a Managed Object by Methods on Held Objects

Some business objects have interfaces that include methods or attributes that return other business objects. If this is the case, then when you have an object, you can use its methods to find related objects. Continuing the previous example, after finding Lou's Claim, you can find other objects that the Claim references. The following example returns a reference to Lou's Policy:

```
// Find Lou's Policy

Policy_ptr louPolicy; // declare a local variable
louPolicy = louClaim->policy();
```

Using a Managed Object

When you find a reference to a managed object, you can invoke methods on it. For example,

```
person->name("Lou Smith");
```

calls the name() method on the person object identified by the person object reference to set the value of the *name* attribute. The Component Broker server handles the use of remote objects in a way that is transparent to you.

Creating a Managed Object

Component Broker managed objects can be created in a number of ways. The following sections describe the default ways of creating managed objects.

Creating a New Object – Create From Key

Every Component Broker managed object class has an instance of a Factory associated with it. The Factory provides a set of interfaces for creating instances of a managed object. The Factory gets some of its interface from the base class CosLifeCycle::GenericFactory. The method used in createFromPrimaryKeyString is introduced in the IManagedClient::IHome interface supplied by Component Broker. This interface specializes the COSLifeCycle::GenericFactory interface and plays the role of factory for Component Broker managed objects. Object providers can implement and provide a tailored subclass of this interface, or they can use the implementation of IHome provided.

All you need to know is how to find the right IHome for creation. This is done using a factory finder. The input required for the factory finder is the name of the intrface of the class that you want this factory to make instances of. The following code segment gets a reference to the Claim Factory for the Life Insurance application.

```
CORBA::Object_var obj;
IManagedClient::IHome_var claimHome;
IExtendedLifeCycle::FactoryFinder_var myFinder;

obj = CBSeriesGlobal::nameService()->resolve_with_string(
      "host/resources/factory-finders/host-scope");
myFinder = IExtendedLifeCycle::FactoryFinder::_narrow(obj);

obj = myFinder->find_factory_from_string("Claim.object interface")
claimHome = IManagedClient::IHome::_narrow(obj);
```

Notice the usage of both the IExtendedNaming::NamingContext interface, the resolve_with_string() method and the IExtendedLifeCycle::FactoryFinder interface, the find_factory_from_string() method. These interfaces are extensions to the corresponding CORBA interfaces, which should be easier to use. The IExtendedNaming::NamingContext interface is described in “Navigating the Name Space Using the Naming Service” on page 36. The IExtendedLifeCycle::FactoryFinder interface introduces the previous methods beyond the CosLifeCycle::FactoryFinder interface, providing simpler and more flexible ways to find a factory:

```
module IExtendedLifeCycle
{
   interface FactoryFinder : CosLifeCycle::FactoryFinder
   {
      CosLifeCycle::Factory find_factory (in
                              CosLifeCycle::Key factory_key)
         raises (CosLifeCycle::NoFactory);

      CosLifeCycle::Factory find_factory_from_string (in
                              FactoryKeyString factory_key)
         raises (CosLifeCycle::NoFactory,
            NamingStringSyntax::IllegalStringSyntax,
            NamingStringSyntax::UnMatchedQuote);

      CosLifeCycle::Factories find_factories_from_string (in
                              FactoryKeyString factory_key)
         raises (CosLifeCycle::NoFactory,
            NamingStringSyntax::IllegalStringSyntax,
            NamingStringSyntax::UnMatchedQuote);
   };
};
```

Having found an IHome by a FactoryFinder, you must provide the IHome with information necessary to manufacture a new object instance. At a minimum, the primary key must be provided. An example of creating a new Claim with a *claimNo* of 1234 is shown in the following example:

```
// Create an instance of the Primary Key Class
ClaimPrimaryKey_var claimKey = ClaimPrimaryKey::_create();

// Set the claimNo attribute in the key
claimKey->claimNo(1234);

// Call createFromKey on the Factory to create Lou's Claim
IManagedClient::IManageable_var theMO;
Claim_var theClaim;

ByteString* claimString = claimPrimaryKey->toString();
theMO = claimHome->createFromPrimaryKeyString(*claimString);
theClaim = Claim::_narrow(theMO);
```

This example is almost identical to the example in “Using a PrimaryKey Helper Class to Find an Object” on page 40. First, create a primary key object to define the identity of the object that will be made. Then, call createFromPrimaryKeyString on the Home to pass the Key as a string.

The createFromPrimaryKeyString method is defined by the IHome class, and all business objects can be created by this method. An object provider provides you with a subclass that introduces other, easier to use creation methods. Additional object creation techniques are described, with examples, in Chapter 6, “MOFW Client Programming Model – Advanced Concepts” on page 83.

Creating a New Object – Create from Copy

Setting and getting the attributes of a managed object can be expensive. There are several reasons for this. First, if the managed object is implemented in another language, each get or set method is actually a cross-language call. Cross-language calls are more expensive than simple, same-language calls. The get and set overhead is even worse if the managed object is remote because each call is actually a remote procedure call and involves significant overhead. Consider the following code fragment:

```
// Let's assume that 'theClaim' is declared and created as in the
// previous code snippet.

// Creating 'theClaim' above required one RPC. Setting the rest of the
// object's attributes involves one RPC per attribute. The following
// lines of code show four such RPCs. This could, of course, be any
// number of RPCs, depending on the complexity of the object.

theClaim->date("10/14/96");
theClaim->state(entered);
theClaim->reason(accident);
theClaim->description("Side-swiped by teenager in a red convertible.");
```

This fragment could involve the following remote method calls:

- The client to Home of Claims to create the Claim.
- The client to Claim MO to set its *date* attribute.
- The client to Claim MO to set its *state* attribute.
- The client to Claim MO to set its *reason* attribute.
- The client to Claim MO to set its *description* attribute.

These calls could be reduced to a single remote method call by using the createFromCopyString() method on an IHome instead of createFromPrimaryKeyString(). In order for you to use the createFromCopyString() method, the object provider must provide you with a Copy Helper Class. The following code segment rewrites the previous example using this design pattern.

```
// Create a new "local" Claim in my process and language.
// Use a Copy Helper Class that the Claim MO provider gave me.
ClaimCopy_var claimCopy = ClaimCopy::_create();

// Now initialize the Claim's attributes. Note that these methods
// execute locally, within the same language.
claimCopy->date("10/14/96");
claimCopy->state(entered);
claimCopy->reason(accident);
claimCopy->description("Side-swiped by teenager in a red convertible.");

// Pass this local copy to the Home and have it return a new Claim MO
// whose attributes are initialized from the local copy's values. Because
// not all ORBs support Pass-By-Value, we first convert the local copy
// helper object to a string.
IManagedClient::IManageable_var theMO;
Claim_var theClaim;

ByteString* claimString = claimCopy->toString();
theMO = claimHome->createFromCopyString( *claimString );
theClaim = Claim::_narrow(theMO);
```

Like a Key Helper Class, a Copy Helper Class instance is always local to your process and implemented in the language you are using. The helper class interface and implementation is given to you by the object provider.

Copy Helper Classes are especially useful if the client application needs to interact with an object during initialization, and then create a managed object from the attributes. A common scenario for this is entering data for the object from a GUI. The GUI updates the local copy helper object, and then the createFromCopyString() method is called when the **Do** push button is pressed on the end user interface (EUI) to do the action.

Note: All of the attributes that make up the Primary Key must be set in the Copy Helper instance before using it for creation.

Using Sets of Objects

An IHome represents a set of managed objects, all of the same type, whose relationship to one another is defined by the object provider, and maintained by the fact that they were all created in the same home. Sometimes an application needs to define (and maintain) the relationships between managed objects, based on the particular business task at hand. This might even include relationships between managed objects of different types (for example, Policy Holder and Beneficiary). This can be done using an IManagedCollections::IReferenceCollection, as shown in the following code segments.

First, create a Reference Collection. The following code shows how to do this from a client. Notice that because a Reference Collection is itself a managed object, it is created using a Home. Component Broker provides a specialized home which makes it easy to create a Reference Collection. Specialized homes are described in greater detail in “Creating Specialized Homes” on page 133.

```
CORBA::Object_var obj;
  IManagedCollections::ICollectionHome_var rcHome;
  IManagedCollections::ICommonCollection_var cc;
  IManagedCollections::IReferenceCollection_var rc;

  obj = myFinder->find_factory_from_string(
        "IManagedCollections::IReferenceCollection.object
           interface/PersistentReferenceCollectionFactory.object home");
  rcHome = IManagedCollections::ICollectionHome::_narrow(obj);
  cc = rcHome->createCollection();
  rc = IManagedCollections::IReferenceCollection::_narrow(cc);
```

Note: The collection created in the previous example is capable of containing objects of any IManagedClient::IManageable subclass. If the collection is to contain elements of a specific type (for example, PolicyHolder), and you would like the collection to enforce this, you can use the createCollectionFor call instead of createCollection.

Having created the Reference Collection, it is now ready to be used. The following code segment shows how a client application adds a PolicyHolder object to the Reference Collection:

```
PolicyHolder_var policyHolder;
// ...
// Find or create the PolicyHolder object to add to the collection.
// ...
try
   {
      rc->addElement(policyHolder);
   }
   catch (ICollectionsBase::IInvalidElement &ex)
   {
      cout << "ERROR: Caught Exception: " << ex.id() << endl;
      cout << "ERROR: Trying to add object to a Reference Collection";
   }
```

Having possibly added many interesting business objects to the Reference Collection, it seems reasonable that at a later time it might be necessary to iterate over every object in the Reference Collection to perform some action. The following code shows one way of doing this:

```
IManagedCollections::IReferenceCollection_var theMixedCollection;
IManagedCollections::IIterator_var theIterator;
IManagedClient::IManageable_var theBO;

// Create an iterator on the reference collection that was found above.
// Note that when an iterator is created, it is automatically positioned
// preceding the first element.

theIterator = theMixedCollection->createIterator();
try
  {
     // Loop through the collection. The "nextElement" method advances
     // the iterator to the next element (on the first invocation, this
     // will advance to the first element) and then return the element
     // pointed to by the iterator.

     while ( theIterator->more() )
      {
         theBO = theIterator->next();
         if ( theBO->is_a("PolicyHolder") )
         // Send him a bill
         if ( theBO->is_a("Beneficiary") )
         // Send him a check
      }
      catch (...)
      {
        cerr << "ERROR: Problem occurred using iterator." << endl;
      }

      // After iterating over the entire collection, the iterator is no
      // longer needed. As such, it must be removed.

      theIterator->remove();
```

The combination of the IManagedCollections::IReferenceCollection and the IManagedCollections::IIterator are used in the above code segment. An IManagedCollections::IReferenceCollection is a generalized collection of object references that can be iterated. IManagedCollections::IIterator supports advancement of the iterator and retrieval of elements by the next() method. IManagedCollections::IReference Collection supports adding and removing elements by addElement() and removeElement(). IManagedCollections::IManagedReferenceCollection is the most basic kind of collection supported in Component Broker. Combining this with the capabilities of IHome provides the basis for writing simple applications and the foundations for the more advanced query and collections capabilities provided by Component Broker. For more information on collections and query capabilities, see Chapter 6, "MOFW Client Programming Model – Advanced Concepts" on page 83.

The try/catch block in the previous code segment is created after the iterator is created. The purpose of the try/catch block is to ensure, even if an exception is thrown during iteration, that the iterator that was created is removed when it is no longer needed. Without this try/catch block, if an exception is thrown during iteration, the remove() method is not invoked and the storage associated with the iterator on the server is not deallocated.

Note: The while loop in the example of iterating through the collection is shown as follows:

```
while ( theIterator->more() )
{
   theBO = theIterator->next();
   ...
```

Although this works, it is inefficient because it requires two calls to the server for each element that is retrieved. It would be better to use a more efficient approach when possible:

```
while ( theBO = theIterator->next() ) ...
```

Transient Sets

Component Broker supports transient collections. Transient collections do not require transactions; they reside in transient containers and thus have no dependency or overhead on database connections. DB2 and Transient collections provide identical programming interfaces. They differ only in their persistence characteristics. Users can find transient collections by passing an interface string to the factory finder as defined in the next section.

Specifying Reference Collection Interfaces

A variety of interfaces for use with factory finders are provided for specifying the type of reference collections that you want to use.

For example, calling find_factory_from_string passing the argument `IManagedCollections::IReferenceCollection.object interface/PersistentReferenceCollectionFactory.object home`, as in the previous example, creates a DB2 backed reference collection. Using the generic string `IManagedCollections::IReferenceCollection.object interface` returns whichever collection is configured as the default.

Transient Collections

IManagedCollections::IReferenceCollection.object interface/
TransientReferenceCollectionFactory.object home

Transient Collections Keyed

IManagedCollections::IKeyedReferenceCollection.object interface/
TransientKeyedReferenceCollectionFactory.object home

Persistent Collections

IManagedCollections::IReferenceCollection.object interface/
PersistentReferenceCollectionFactory.object home

Persisent Collections Keyed

IManagedCollections::IKeyedReferenceCollection.object interface/
PersistentKeyedReferenceCollectionFactory.object home

Remembering your Favorite Objects

Assume you need to remember an interesting or important managed object instance. Ideally, you need to save the reference or pointer to the object. There are several complexities, however.

- The object might be implemented in another language or on a remote system.
- If the client application shuts down and restarts later, the object may have been removed from memory because of a lack of use. If the object is restarted, or reactivated, it probably will not be located in exactly the same place in memory.

Component Broker solves these problems, as does CORBA, by introducing the concept of an *object reference*. An object reference is opaque and you cannot set its internal structure. However, a reference always and uniquely refers to a managed object regardless of where it resides in the network. Continue the example from the previous section, and assume that you run the following code segment:

```
ofstream fout("SOMEFILE.DAT");

// Get a "string" version of my reference to Steve
// steve points to Steve or a proxy to Steve
char* steveStringifiedReference = CBSeriesGlobal::orb()->object_to_string(steve)

// Save the string to a file using a "pseudo" file routing
fout << steveStringifiedReference;
fout.close();

// I do not need Steve anymore
```

```
steve->release();
```

You saved a reference to Steve as a stringified object reference, and can use this string to re-access Steve at a later time. The following code segment is an example of re-accessing the Steve object.

```
char* infile = "SOMEFILE.DAT";
ifstream fin(infile);

char steveStringifiedReference[1024];
memset(steveStringifiedReference, 1024, '\0');

// Get back the string I saved

fin >> steveStringifiedReference;

// Make an object reference for Steve

CORBA::Object_var obj;
PolicyHolder_var steve;

obj = CBSeriesGlobal::orb()-> string_to_object(steveStringifiedReference);
steve = PolicyHolder::_narrow(obj);

// I can now work with Steve.
steve->name("Steven");
```

Releasing and Deleting Objects

Eventually, you no longer need to use an object that you found or created. Component Broker supports two interpretations of “no longer needs.”

- The remove() method deletes the object and its persistent instance data.
- The release() method informs the object that the client application no longer plans to reference the object. The object still exists in the server, and other applications may be using it, but the client calling the release() method will no longer use the object.

In order to better understand what happens as a result of a remove() or release() request, and the difference between the two, refer to “Finding a Managed Object” on page 38 and “Creating a Managed Object” on page 41. To illustrate this, use an example which has a relational database table of information about people, the policy holders from the Life Insurance application. This table persistently stores the state data for objects of the PolicyHolder class.

As described in “Finding a Managed Object” on page 38, managed objects may be found by invoking the findByPrimaryKeyString() method on a home. The result of this operation is two-fold:

- If an object matching the primary key is not currently active in the server, that is, in use by some other user or application, then one is activated and brought into memory on the server. Figure 24 on page 49 shows the case where the Robert object is currently not active.
- An object reference, also referred to as a *proxy*, to the object on the server is returned to the client. In CORBA, a proxy is itself an object: it is an object on the client which identifies an object on the server, and has the same interface as the object on the server.

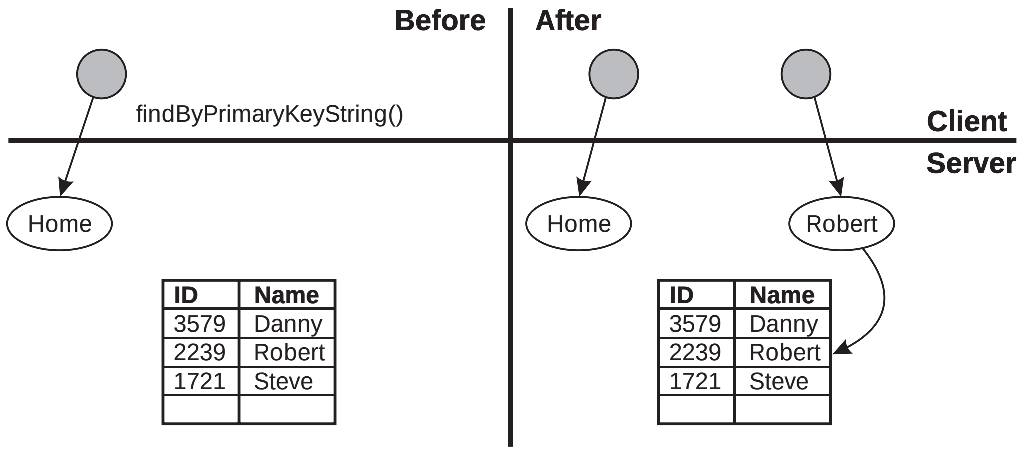


Figure 24. Finding Managed Objects

When creating a new managed object, there is no object on the server, active or otherwise, with a matching primary key. If there is, the creation will fail. A new object is created in memory on the server, a new row is added to the database table, and a proxy is created on the client as shown in Figure 25.

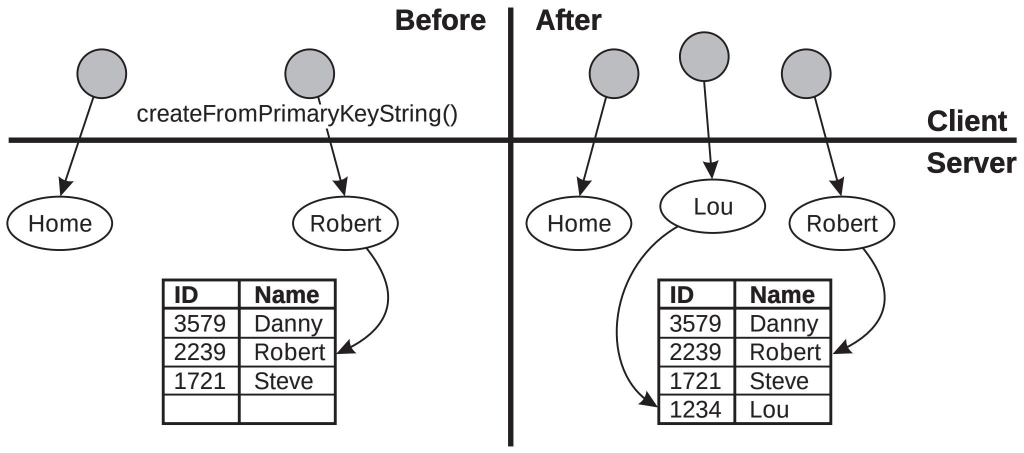


Figure 25. Creating Managed Objects

Having found or created a managed object, the client application has a pointer to a proxy, not to the actual managed object. (A memory pointer to the actual managed object out in some server on the network has no meaning in the client's address space.) However, because the proxy has the same interface as the managed object on the server, the client application uses it just like the object itself. The Component Broker ORB and server take care of routing all method calls and parameters from the client proxy to the server object.

Figure 26 on page 50 shows how a client application changes the *name* attribute of an object on the server.

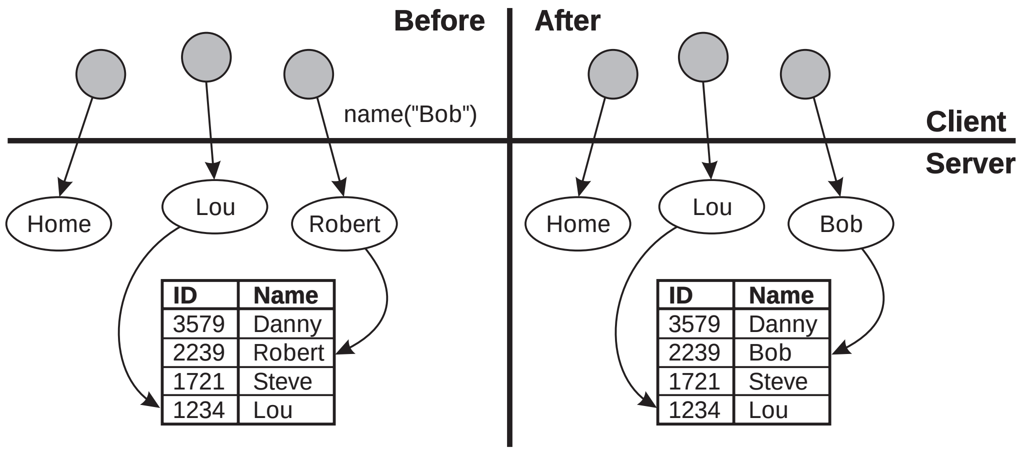


Figure 26. Using Managed Objects

Now, assume that Robert or, as he prefers to be called, Bob calls up and says he would like to cancel his only insurance policy. At this point, the client application concludes that Bob is no longer a policy holder, and proceeds to delete the Bob object. The method for deleting an object in Component Broker is called remove(). Invoking the remove() method on a managed object has the following effect:

- The managed object's persistent state is removed from the database table.
- The managed object is removed from memory on the server.

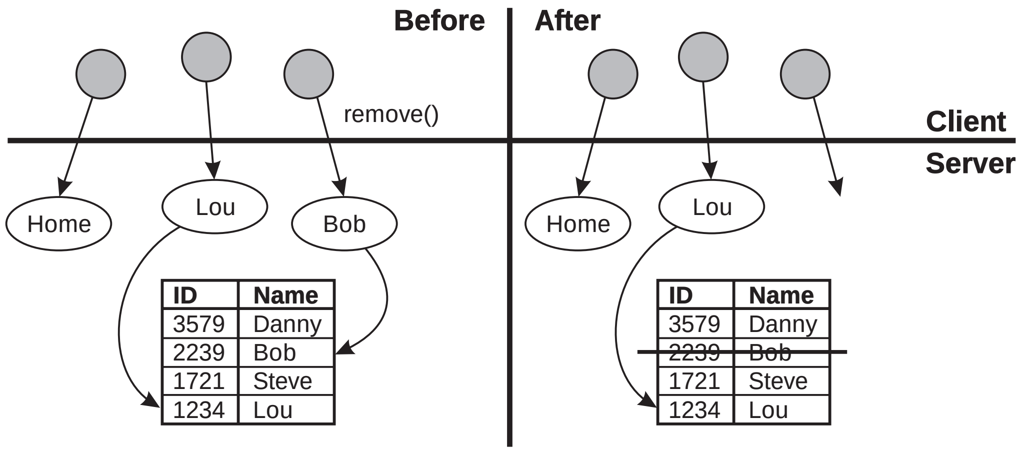


Figure 27. Removing Managed Objects

There are one important thing to take note of here:

- The proxy (object) is not removed from memory on the client as a result of invoking remove() on the proxy. At this point, the proxy is not pointing to a valid object on the server. In order to delete the proxy (object) from memory, you must explicitly invoke CORBA::release(), passing it the proxy (object).

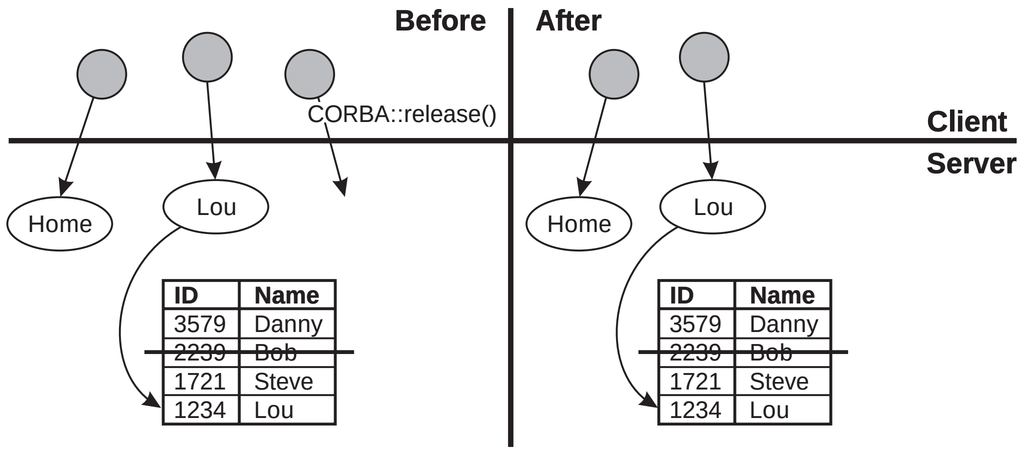


Figure 28. Releasing (Removed) Object References

Figure 28 shows the result of invoking CORBA::release() on the object reference which was removed. Of course, a client application will also want to release references to objects which it is not removing. This would be the case if customer Lou called up to increase his insurance coverage. In this scenario, the client application performs the following steps:

1. Invoke the findByPrimaryKeyString() method to find Lou's policy.
2. Invoke the amount() method on the policy to increase the coverage.
3. Release (not remove) the policy object after calculating the new premium.

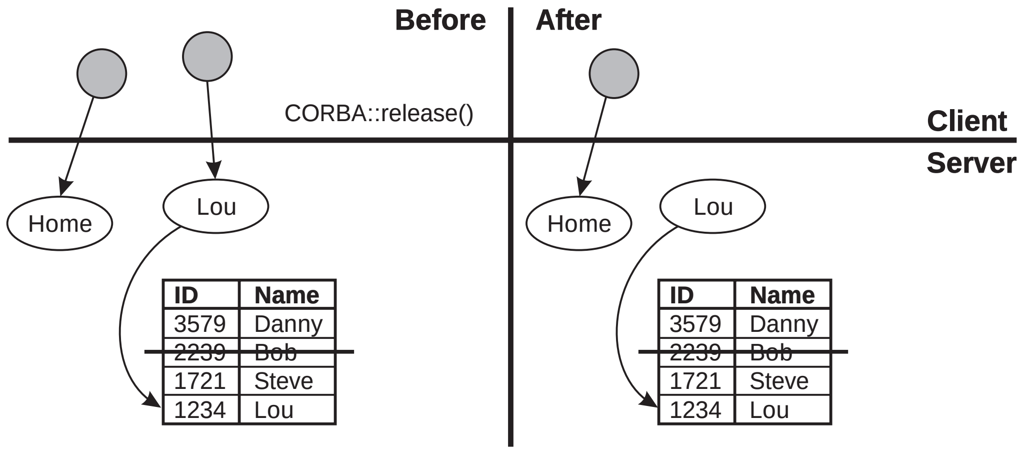


Figure 29. Releasing Object References

Figure 29 shows that the proxy is deleted from the client's memory space, but the object remains on the server. Eventually, if there are no other references to the object within the server, the server deletes the object from the server's memory space but not from the database table. Note that the client application has no control over when the server decides to remove an active object that is, one that has not been deleted by the remove() method. In fact, if the client application is waiting for user input, a server may decide to delete from memory an object to which the client application holds a reference. When the client application finally receives input from the user and attempts to use the object, the server reactivates the object by putting it back in memory.

The remove() method is essential for supporting application logic. Client applications need a way to destroy and delete objects and their instance data. In the previous example, Bob no longer exists as far as the applications are concerned. The release() method on the other hand, helps implement memory management. In a client/server system, applications span multiple servers, processes, and languages.

Traditional memory management techniques such as garbage collection or relying on clean-up when a process terminates do not work. The client garbage collector does not see the server memory, and the server does not contain references to the objects because they are on the client. Therefore, the server needs to be told when clients no longer reference objects.

Because the server does not maintain a list of every object reference that every client application has requested, you might try to use an object reference to an object on which another client application has already invoked remove(). If this happens, an exception is triggered. The same thing would happen if a client application tried to invoke a method on an object on which it had previously invoked the release() method.

Coding Tips for proper CORBA Memory Management

The rule for proper CORBA memory management is the following: The caller owns all storage.

The general model on the client is to use _var objects. This means that when an _ptr is returned, it should be placed into an _var by the client. The _var assumes responsibility for the storage pointed to the _ptr that is placed into the _var. The _var is a class and its destructor runs when the _var goes out of scope.

The other option for clients is to use the duplicate() and release() methods. The duplicate() method is available for making a copy of a proxy, while the release() method is used to free the local memory used by a pointer.

None of this should be confused with the LifeCycle Service remove() method. This has different semantics. It involves actually deleting the server object that is at the other end of a proxy.

More information on CORBA coding style and conventions, especially as they pertain to memory management, can be found in Appendix C, “C++ CORBA Programming” on page 299.

Using Object References

Managing the storage of object references is one of the areas where proper memory management is required. You must use the _duplicate and release operations as described above or use _var variables.

There are also special considerations when passing object references as parameters. The caller is always responsible for allocating storage for object references. The caller is also responsible for releasing of all *inout* and returned object references.

For *inout* parameters the caller provides an initial value. If the callee wants to reassign the *inout* parameter, it must first all the *release()* operation on the initial input value. To continue to use an object reference passed as an *inout*, the caller must first duplicate the reference.

See the section titled "Storage Management and _var variables" in Appendix C, “C++ CORBA Programming” on page 299 for details on memory management.

Commonly Used CORBA Interfaces

References have already been made to interfaces that are not directly defined by the Programming Model that is provided by Component Broker, for example _narrow(). This is because the Programming Model is built on top of CORBA and relies on the interfaces that are already defined by CORBA. This section describes the most commonly used CORBA interfaces that Component Broker developers will use.

For further information on these and other operations defined by these interfaces, see the appropiate section in the Object Request Broker section of the *Component Broker Programming Reference*.

CORBA Class Interfaces

The CORBA interface also provides some class operations that are commonly used. These are used like a C++ class reference (e.g. CORBA::is_nil(somePointer);)

is_nil
This operation returns a boolean that indicates if the input object reference is nil. This is useful for many operations involving object references, including those operations that do not throw exceptions when they fail - for example CORBA::Object::_narrow().

release
This operation releases resources associated with an object or pseudo-object reference.This operation may or may not perform a C++ delete operation. A reference count is used by this operation and CORBA::Object::_duplicate(). When the reference count reaches zero then the appropriate delete operations are performed. Care must be taken when using the release and _duplicate operations to ensure that objects are neither inadvertently deleted or are leaked. Alternatively use the _var technique described in the next section.

string_dup
This operation copies a string. The resulting string should be subsequently freed using the CORBA::string_free operation or assign the string to a _var variable which will free the string appropriately. Strings and wide strings, unlike the other basic CORBA types, have associated allocated memory. Care must be taken when using these variables. A common example is when returning a string from an operation.

CORBA::Object Interfaces

_duplicate
This operation duplicates an object reference. This is particularly useful when passing references to objects to resolve memory ownership issues. For every _duplicate that is performed on an object an equal number of release() must also be performed for proper memory management. An alternative to the _duplicate() and release() logic is to use _var support as described in the next section of this chapter.

get_current
This operation returns the CORBA::Current. This object reflects the execution context of the currently executing thread. Information about the Security, Sessions and Transactions services can be retrieved using this interface. For example, see “Transactions” on page 83.

_is_a
This operation is used to determine whether an object reference supports a given IDL interface. If the object supports the interface the _narrow operation can be successfully performed.

_is_equivalent
This operation is used to determine whether two object references refer to the same object.

_narrow
This operation is used to narrow a more generic interface to a more specific interface. This operation will return an empty pointer without throwing an exception if the interface cannot be narrowed to the requested type. Care must be taken to check the returned value before using it.

_nil This operation returns a nil CORBA::Object. This object could be used for comparison operations.

_non_exsistent
This operation determines whether an object reference refers to a valid object. This will result in verification of the object reference only, no other operations are performed on the requested object.

object_to_string

This operation converts an object reference to an external form that can be stored for later use or exchanged between processes. The string_to_object operation can be used to reconstruct the object reference.

string_to_object

This operation converts an stringified object reference to an reconstructed object reference. The object_to_string operation must have been used to create the input stringified data.

Note: Although object_to_string is the way to save object references for future usage, the returned data should only be used with string_to_object to reconstruct that object reference. Do not use the string for comparing equivalence of object references.

The string is an IOR which can composed of several pieces of data - some the ORB generates and some contributed by other components. The object_to_string operation may return different values at different times because various Object Services may be adding information to this IOR.

Summary: The Client Programmer's Check List

When using a set of managed objects to implement an application, you need to know how to use the following things about managed objects and the Managed Object Framework which encapsulates and interoperates with the managed objects. Table 1 is organized around the tasks outlined in “Client Programming Model: Basic Tasks” on page 35. Note that *xxx* is used to represent the name of the managed object. For example, Claim and Policy were used in this chapter as examples of domain-specific business objects. This also provides a good clue as to whether or not the classes were constructed by the business object provider or whether they came as part of the Component Broker system.

Table 1 (Page 1 of 2). Summary of Interfaces

Task	Class Types	Methods	Purpose
General Use	StandardSyntaxModel	stringToName	Converts strings to CORBA name types.
Find an object	NamingContext	resolve	Finds an object in the namespace
	IHome	findByPrimaryKeyString	Finds an object in a home
	*xxx*PrimaryKey	set*xxx* methods and toString	Sets the key value
Create or delete an object	FactoryFinder	findFactory	Finds the factory object.
	IHome	createFromKeyString	Creates an object with a key only.
	*xxx*PrimaryKey	set*xxx* and toString	Passes information on key to the server.
	*xxx*CopyHelper	set*xxx* and toString	Passes copy information to the server.
	any *xxx* ManagedObject subclass	remove and release	Deletes or releases object from memory.
Use sets of objects	IReferenceCollection	createIterator	Creates an iterator object.
	IIterator	next	Iterates over a collection and gets the objects.

Table 1 (Page 2 of 2). Summary of Interfaces

Task	Class Types	Methods	Purpose
Remember interesting and important objects	NamingContext	bind	Binds an object into the name space with a name.
		resolve	Finds an object in the name spece by its name.
	theORB	object_to_string	Creates a stringified form of an object reference.
		string_to_object	Creates an object reference from a string.

This is the basic set of methods needed to work with Component Broker managed objects. More advanced methods for a C++ client are discussed in Chapter 6, "MOFW Client Programming Model – Advanced Concepts" on page 83.

Chapter 5. MOFW Server Programming Model

This chapter describes the interfaces and processes you follow to create a business object based on the Managed Object Framework (MOFW) interfaces. The first section introduces these abstractions and the relationship that they have to the domain-specific interfaces and implementations that make up the business logic and business data inherent in a business object.

After this brief overview, the basic steps involved in developing a business object are described.

Business Object Basics

Each business object in Component Broker consists of a number of pieces. The primary interface to the business object and its implementation are shown in Figure 30.

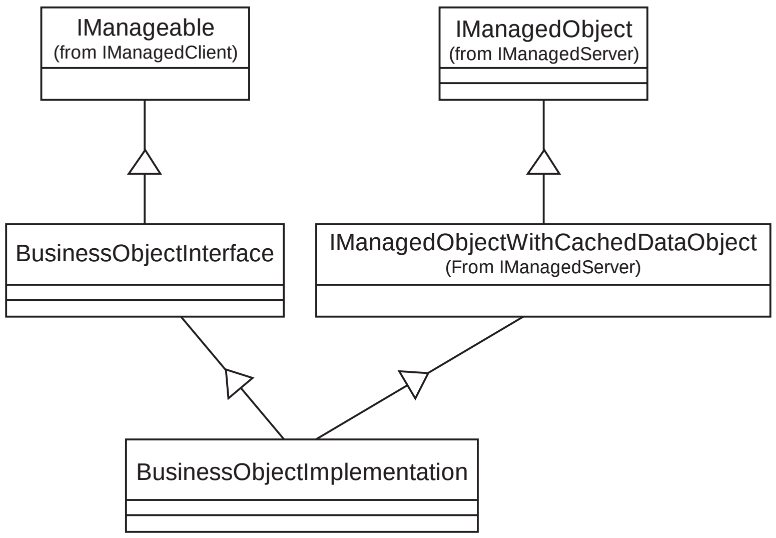


Figure 30. Business Object Basic Structure

The BusinessObjectInterface abstraction in Figure 30 represents the interface as specified by the domain expert involved in the object-oriented analysis and object-oriented design (OOA/OOD) activities for the system under construction. This is represented by IDL and is an interface-only class from a run-time perspective. It is also the interface that you are most likely to interact with this business object. These types of interfaces always inherit from the IManageable interface in the IManagedClient module.

The BusinessObjectImplementation abstraction is an IDL file which is used to generate implementation bindings that are for the business logic that implements the BusinessObjectInterface described previously. The BusinessObjectImplementation inherits from one of the abstractions contained in the IManagedServer module. The example shows inheritance from the IManagedObjectWithCachedDataObject interface. This name indicates a number of things about how the BusinessObjectImplementation object will appear. The IManagedServer module interfaces prescribe additional responsibilities that must be met in the BusinessObjectImplemetation that deal with Component Broker and the infrastructure.

The two key modules involved are IManagedClient and IManagedServer. IManagedClient represents those abstractions that are generally accessible to clients, and subclassed by business object builders when they want to expose an interface to client programmers. As described in Chapter 2, “Personal Life Insurance Application Example” on page 15, this includes abstractions such as IHome, IHandle, and IManageable.

IManagedServer is a module that contains interfaces used only by object providers and customizing applications to run in particular environments. The abstractions in IManagedServer enable business objects to run in the Component Broker server and properly take advantage of the Component Broker server capabilities. Examples of this are the IDataObject abstraction and the abstractions that deal with enabling the basic business object for a specific data object pattern. Understanding these two modules and the different purposes they serve is important.

Before a business object can be unit tested, you need to complete the construction of its classes, namely primary keys and copy helpers. The additional steps involved in preparing a business object to be installed in a Component Broker server for further testing, and eventual deployment, are discussed in Chapter 11, “Assembling and Installing Business Objects on AIX and Windows NT” on page 197. These are found in the module named IManagedLocal, and are the *local-only* objects which assist in making MOFW work across clients and servers.

There are also other considerations with respect to factoring of the interfaces for various clients. Before proceeding with the steps to develop a simple business object, it is important to establish some common terminology.

Business Object State

In object-oriented programming, an object has both behavior and state. An object's behavior is manifested in the implementations of the methods on the public and private interfaces of the object. The state of an object, on the other hand, is predominantly manifested in the public and private data members of the object. The state can be divided into categories of *non-essential* or *essential*.

An object's non-essential state consists of the subset of the state that can be calculated or derived from other state, and the subset of the state which is transient and does not need to be persistent because it can be recreated as necessary. A state that is not non-essential is considered to be essential state. To illustrate, assume a Policy business object consists of the following:

- Policy number
- Amount (of coverage)
- Premium
- Insured (policy holder)
- Comment
- Risk calculator (helper object)

Furthermore, assume that the premium can be calculated by dividing the amount by 100. The risk calculator is a helper object which itself has no state, and a new one can be created if the Policy business object is passivated and subsequently reactivated. The insured, on the other hand, is another business object of type PolicyHolder. If the Policy business object is passivated and reactivated, the Policy business object must find the same PolicyHolder as it had before passivation.

Given the previous assumptions, Figure 31 on page 59 shows that the policy number, amount, insured, and comment are part of the essential state of the Policy object, whereas the premium and risk calculator are part of the non-essential state.

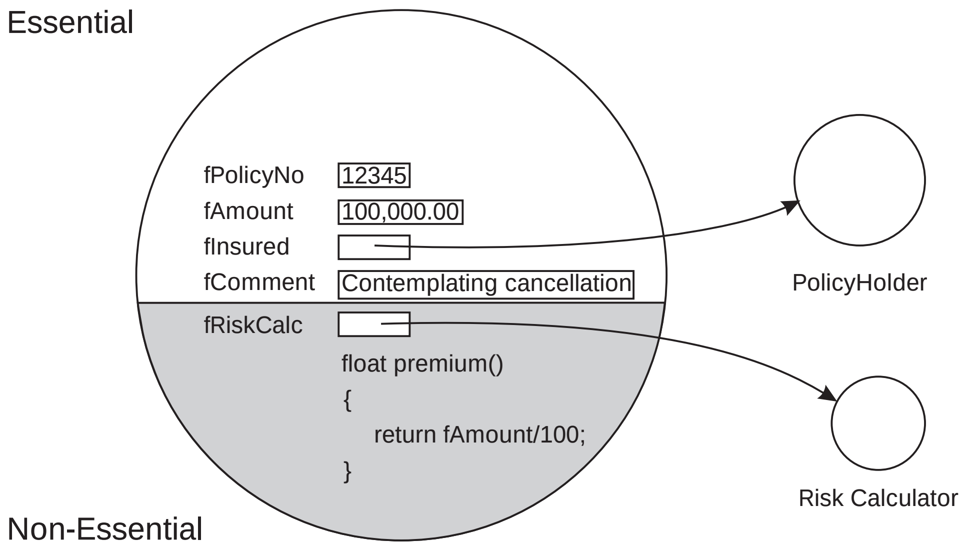


Figure 31. Essential and Non-Essential States

Business Object Attributes

In C++, the term *attribute* is often used synonymously with the term *data member* to describe a piece of data which is part of an object's state. In CORBA or, more precisely, in IDL, attribute has a different meaning altogether. In CORBA, IDL describes the interface of an object. Because CORBA is all about distributed objects, and because there is no way to directly address data inside an object from across a network, IDL does not support the notion of having data as part of an object's interface. Thus, in IDL, attribute refers to a pair of methods for accessing and changing the value of a specific type. For instance,

```
  attribute string comment;
```

defines the following methods in C++

```
  virtual char* comment ();
  virtual ::CORBA::Void comment (const char* comment);
```

The first of these two methods retrieves the value of the comment. The second allows the client to change it. A business object often has a state which is necessary to support the implementation of the business logic, but which is not directly accessible by an attribute on the object's interface. In object-oriented design (OOD), this is referred to as *encapsulation*.

Now you should be able to go through the steps necessary to specify and implement a business object based on the MOFW set of Component Broker interfaces. The goal of this chapter is to present enough material for you to create the minimum set of artifacts necessary to support the basic client programming model as described in “Client Programming Model: Basic Tasks” on page 35.

Developing a Business Object

The minimal set of required activities for developing a business object is described in this section. The real business objects that you deploy will probably leverage some of the additional interfaces and capabilities of Component Broker described in subsequent sections. This section helps you get started and understand the basics involved in constructing a business object. This section describes more details about the abstractions of the MOFW, how they are implemented, and what options exist when constructing business objects based on the MOFW.

The minimal set of steps is:

1. Develop an interface to the business object.
2. Choose a pattern for handling essential state.
3. Implement the business object methods.
4. Implement the methods required by the MOFW interfaces.
5. Implement the necessary primary key class.
6. Implement the optional copy helper class. Implementation of a copy helper class is optional, but strongly recommended. Copy helper objects can greatly improve the performance of creating a business object in a server from a client application.

At the end of these tasks, the business object should be ready for unit testing. Details on how to unit test business objects can be found in Appendix D, “Unit Test Environment” on page 333.

Developing an Interface to the Business Object

In this and the following sections, the Policy business object is used as an example. The Policy business object has the following interface:

```
#include <IManagedClient.idl>
#include "Beneficiary.idl"

exception InvalidAmount {};

interface Policy : IManagedClient::IManageable
{
   void addBeneficiary(Beneficiary benRef);
   void delBeneficiary(Beneficiary benRef);

   readonly attribute long policyNo; // Primary key for 'Policy'

   void changeAmount(float newAmount) raises(InvalidAmount);
   float getAmount();

   attribute string comment;
   readonly attribute float premium;
}
```

This represents an interface that clients of the Policy business object can look at, understand, and design to language bindings and is used for compiling client applications.

Note: The previous interface would be contained in a file named Policy.idl.

This is a standard looking IDL file. It declares some methods and attributes. What makes this a Component Broker MOFW-specific interface is the fact that it inherits from IManagedClient::IManageable, which is included from the IManagedClient.idl file.

This interface is of particular interest from a Component Broker MOFW perspective because of the read-only attribute called *policyNo*. “Implementing the Primary Key Class” on page 73 describes how to develop a primary key class for the Policy business object. However, even as early as this, the first step in developing a business object, you should already be thinking about primary keys.

A primary key class encapsulates the data that makes one instance of a class of objects unique from other instances of the same class of objects. Occasionally, the data that uniquely identifies an object is not part the object's state. In such cases, the data which uniquely identifies an object would probably be something like a Universally Unique IDentifier (UUID). However, typically, it is a subset of the state of an object which makes the object unique. In the Policy business object example, the policy number is used to distinguish amoung seperate instances.

As described earlier, attributes on an object's interface define methods for accessing the state of an object. For example, `attribute long policyNo;` defines the following methods in C++:

```
virtual ::CORBA::Long policyNo ();
virtual ::CORBA::Void policyNo (::CORBA::Long policyNo);
```

The first method retrieves the value of the policy number. The second allows the client to set it. A business object gets its identity set in its data object from its primary key as part of creation or reactivation. For more information, see “Implementing the Primary Key Class” on page 73. Changing an object's identity after creation by setting one of the attributes which make up the key to a new value is not allowed in Component Broker. Applying the *readonly* qualifier to the *policyNo* attribute prevents the generation of the set method shown in the previous example, which means a client application has no interface for changing a Policy object's identity.

Changing an object's identity after creation requires the following multi-step process:

1. Create a new object with the new identity.
2. Copy the state of the old object into the new object.
3. Delete the old object by invoking the remove() method.

This allows the application adaptor to properly keep track of, and manage, instances of business objects on a Component Broker server.

Module Scoping

Although the examples used in this book do not show it, wherever possible, IDL interfaces and types should be enclosed inside a module scope. IDL declared outside of a module scope takes up name space in the global IDL name space and risks having name collisions with names declared by other IDL developers.

For example, if two groups in an organization write IDL interfaces without placing them in different modules, and happen to choose names for their interfaces that overlap with each other, the result is name collision when the IDL is loaded into a shared Interface Repository. However, if each group chooses a unique module name, perhaps based on a combination of company name and group name, each group's IDL is able to co-exist with all other IDL interfaces.

This example is even more useful when Java and the Internet are considered. In the IDL-to-Java mapping specification adopted by the Object Management Group (OMG), IDL module names are mapped directly into Java package names. IDL declared outside a module is mapped into classes and interfaces that are not contained in any package. Because one of the purposes of packages in Java is to partition the Java name space for the entire worldwide Internet, it is particularly important that IDL that might be mapped into Java be contained within a module. Similar reasoning applies to C++, but because C++ classes are typically not downloaded and shared among all users of the Internet, the risk is somewhat lessened.

For example:

```
module XYZCompany_Finance
{
   interface Receivable
   {
      attribute long customerID;
      attribute long amountCents;
   };
};
```

The Receivable interface has the fully-scoped name XYZCompany_Finance::Receivable.

Design Tips for Business Objects

As discussed in “Business Object Attributes” on page 59, in C++, the term attribute describes a piece of data which is part of an object's state. Good C++ object-oriented design (OOD) principles discourage putting attributes on the public interface of an object. Public attributes allow the user of a C++ object to directly manipulate the state of the object, without any control by the object itself. C++ OOD uses the term *encapsulation* to mean putting an object's data members in the protected or private sections.

Because CORBA and IDL do not allow specifying data as part of an object's interface, public data is not a problem. In CORBA, attributes on an object's interface (as specified using IDL) define methods for accessing the state of an object. Because IDL does not allow the specification of exceptions to be thrown as the result of an attribute, IDL attributes do not provide the same level of protection as encapsulation in C++. To see why, continue with the Policy example. For illustration purposes, assume that the insurance company does not want to issue a policy unless the amount is over $10,000.00. As shown previously, if *amount* had been defined as an attribute, it would result in the following two methods being declared:

```
virtual ::CORBA::Float amount ();
virtual ::CORBA::Void amount (::CORBA::Float amount);
```

Look at the implementation of the method for setting the attributes value:

```
virtual ::CORBA::Void amount (::CORBA::Float amount)
{
   if ( amount > 10000.00 )
      fAmount = amount; // save value in private data member
   else
      // What shall we do here...?
};
```

IDL does not allow us to raise an exception on an attribute. The method signature, as emitted by the IDL compiler, has no return value or output parameters. All you can do is *not* save the new value, return to the caller, and hope that the caller notices that the value was not changed. A better way to truly encapsulate the amount is to not make it an attribute in IDL, but rather a pair of get and set methods. This lets the business object do constraint checking and communicate error conditions to the client of the business object.

On the other hand, this needs to be balanced against performance. The Query Service of Component Broker performs best when a query is specified in terms of attributes, and furthermore, that the attributes of the business object map one-to-one to the attributes of the business object's data object. Doing so lets the Query Service of Component Broker push down the query to the underlying resource manager. Resource managers such as relational database managers perform queries much faster than Component Broker can without any other help.

Module and interface names must be different. Although identical names are valid IDL syntax, it gets mapped to nested classes by the C++ emitter, and the C++ compiler does not allow the name of a Container class to be the same as a Contained class.

Selecting a Pattern for Handling Essential State

This step introduces another interface that looks like this:

```
#include "Policy.idl"
#include <IManagedServer.idl>

interface PolicyBO : Policy,
   IManagedServer::IManagedObjectWithDataObject
{
};
```

Note: This interface is contained in a file named PolicyBO.idl.

This interface is small and straightforward. By inheriting from IManagedServer::IManagedObjectWithDataObject a decision has been made about the pattern to be used for handling the essential state of this business object. There are two interfaces in IManagedServer that correspond to the two patterns that the Component Broker MOFWs support for handling the essential state of a business object.

IManagedObjectWithDataObject
This interface implies that this business object has its state managed by a data object. Because this business object has a data object, and no implied caching as in the next interface, this is called the *delegating pattern*. There are implications here for the implementor of the business logic methods. These are described in "Implementing Business Object Methods and Attributes" on page 64.

IManagedObjectWithCachedDataObject
This interface implies the presence of a data object and support from Component Broker for fetching this data into a cached set of state data maintained by the business object at the appropriate times throughout the lifetime of a business object. This is called the *caching pattern*, because the business object is caching the state being maintained by the data object.

The Data Object

For the patterns described previously, a data object is necessary. A data object manages the essential state of a business object. For the working example, assume that the data object for the Policy is as follows:

```
interface PolicyDO
{
   attribute long policyNo;
   attribute float amount;
   attribute string comment;
   attribute PolicyHolder insured;
}
```

The interface for the data object contains one attribute for each piece of the business object's essential state, as shown in Figure 31 on page 59, and that there are no attributes for the business object's non-essential state. Also notice that this interface is not a one-to-one mapping of the attributes on the business object. The *premium* attribute from the business object is not present here because it is not part of the essential state. Also, although *amount* and *insured* were not attributes on the business object, they are attributes on the data object because they are part of the business object's essential state.

This interface is enough to let you develop the object's business logic, without having to actually implement the underlying data access methods. However, before the business logic can be tested, the data object interface must in fact be implemented. "Implementing the Data Object" on page 337 describes the implementation of the data object interface to unit test an object's business logic. "Data Object Customization" on page 204 describes how to customize the data object to allow the business object to be installed into a particular application adaptor.

Design Tips for Data Objects

There are several considerations when defining and naming the data object and its attributes. First, be careful when naming the attributes for the data object. The Query Service of Component Broker operates more efficiently if the attribute names on the business object are the same as the attribute names on the data object. Second, while attributes on the business object's interface may not be a good idea, depending on the amount of encapsulation desired, they are a convenient way of declaring get and set methods on the data object. Because the data object is used only by the business object, and by the application adaptor, encapsulation in the data object is not a consideration.

Implementing Business Object Methods and Attributes

In this step, the implementation bindings for PolicyBO are filled in with the business logic required to support the Policy interface specified in the Policy.idl file in C++ code. The implementation bindings generate a class called PolicyBO_Impl in this case. The emitted files, named PolicyBO.ih and PolicyBO_I.cpp, do *not* have any of the methods that are introduced by the parents of PolicyBO, but which need to be implemented by PolicyBO.

Note: This is a limitation of IDL as defined by CORBA: there is no way to specify in IDL whether an interface has been implemented. Without this knowledge, an IDL compiler must make an assumption. The IDL compiler provided with Component Broker assumes that every interface is implemented.

These methods must be introduced to PolicyBO.ih and PolicyBO_I.cpp after emitting.

The Object Builder is going to ensure that the .ih and _I.cpp files at the business object level get the method declarations and implementation stubs that are introduced at the level above. Additional C++ methods and data can be introduced as needed to meet the requirements of the implementation. However, anything additional that is added is not accessible remotely from clients because the interface used by clients is described in the client bindings. For C++, the client usage bindings are specified in .hh files according to the CORBA 2.0 C++ mappings.

A sample implementation of the addBeneficiary() method follows:

```
// somewhere at the top
#include "PolicyBO.ih"

// a business logic method implementation
::CORBA::Void PolicyBO_Impl::addBeneficiary(Beneficiary benRef)
{
   // Assume that beneficiaries was declared as a private C++ data
   // member of type IManagedCollections::IReferenceCollection_var.
   beneficiaries->addElement(benRef);
}
```

Each of the business logic methods needs to be implemented in a similar fashion. Handling of attributes from the interface is more involved, and is described in the following sections.

Using C++ 'this' References in Business Objects

Care must be taken when programming all methods in business objects that use references to themselves when communicating with other objects. Methods must use the programming model as described in this section when using these self references. The technique of using “this” is no longer supported in the programming model in these circumstances.

A local proxy class is created for each interface defining the managed object implementation, the managed object interface, the business object interface, and every other interface that they may support. Only instances of the local proxy of the managed object implementation are instantiated and these proxies must be used for self references. The _this() method can be used to access this proxy in the business object.

The home provides a copy reference to a local proxy for the create() and findBy() methods that return object references. The following example shows the set of rules to follow when an object passes itself as an argument or returns itself as a return argument:

```
I_ptr I::foo(Bar_ptr bar)
{

   // When using normal sequences and structs
   sequence[1] = _this();
   struct.i = _this();

   // We recommend using sequences that release object references automatically.
   // If you must use sequences that do not automatically release their
   // references, there must be another mechanism to free them.
   I_var objref = _this();        // The var will release the objref upon exiting
                                  // from this scope. This is just one of many
                                  // possible ways to release the object reference.
   specialSequence[1] = objref;  // Sequence instantiated with no release indication.

   // return value
   return _this();
}
```

For backwards compatibility, the _self() method is retained in this release of Component Broker. The following example shows the set of rules to follow when using _self():

```
I_ptr   I::foo(Bar_ptr bar)
{
   // When using normal sequences and structs the reference must be
   // duplicated because these structures own the objects stored in them.
   sequence[1]= I::_duplicate(_self());
   struct.i = I::_duplicate(_self());

   // For sequences which do not release the objects within it.
   specialSequence[1] = _self();  // Sequence instantiated
                                  // with no release indication

   // The reference must be duplicated for returning the value
   return (I::_duplicate(_self()));
}
```

Warning: We will be deprecating the **_self()** method in a future release of Component Broker in order to maintain CORBA compatibility via the **_this()** method. We encourage the user to migrate existing code, and to start using the _this() method in any new code.

Option 1 – Patterns for Handling State (Caching)

If IManagedObjectWithCachedDataObject was chosen as the data object pattern, then all essential state, and all non-derived non-essential state, is stored or cached in the business object. To store this state in the business object, protected or private data should be declared inside of the implementation binding header (.ih) file. For Policy and PolicyBO previously mentioned, add the following in the PolicyBO.ih file:

```
class PolicyBO_Impl : public virtual ::PolicyBO_Skeleton
{
   ...
   // methods being implemented go here
   ...
      protected:
         ::CORBA::Long fPolicyNo;
```

```
        ::CORBA::Float fAmount;
        PolicyHolder _var fInsured;
        ::CORBA::String_var fComment;
        RiskCalculator * fRiskCalc; // C++ helper object
};
```

Attributes: The implementation bindings *getter* and *setter* methods for each of the attributes specified in an interface. In the PolicyBO_I.cpp file, the getter and the setter for the *comment* attribute are implemented as follows:

```
::CORBA::Void PolicyBO_Impl::comment (const char* comment)
{
   fComment = comment; // ::CORBA::String class will make copy
}

char* PolicyBO_Impl::comment ()
{
   return ::CORBA::string_dup(fComment);
}
```

The previous example is a simple implementation of getters and setters. More complex logic and exception handling may be required. In fact, getters and setters might actually have logic in them to write to resource managers or get data from resource managers. However, as described previously under "Design Tips for Business Objects" on page 62, if the logic required in the implementation of getters and setters includes any sort of bounds checking or error checking, then the attribute should be changed to a pair of methods in the IDL file.

Other Methods: Other business logic methods which access state data do so similarly to the *comment* attribute mentioned previously: they use the data that is cached in the business object.

Option 2 – Patterns for Handling State (Delegating)

If IManagedObjectWithDataObject was chosen as the data object pattern, then all essential state is accessed by the data object: when the business logic needs to get or set its essential state, it does so by using the attributes on the data object. This is called the delegating pattern because the maintenance of the business object's state is being delegated to the data object. All non-derived non-essential state is still cached in the business object. Using this pattern, the implementation binding header (.ih) file for PolicyBO (PolicyBO.ih) would look something like this:

```
class PolicyBO_Impl : public virtual ::PolicyBO_Skeleton
{
   ...
   // methods being implemented go here
   ...
   protected:
   PolicyDO_ptr fDataObject;
   RiskCalculator * fRiskCalc; // C++ helper object
};
```

Attributes: The implementation bindings generate getter and setter methods for each of the attributes specified in an interface. In the PolicyBO_I.cpp file, the getter and setter for the *comment* attribute would be implemented as follows:

```
::CORBA::Void PolicyBO_Impl::comment (const char* comment)
{
   fDataObject->comment(comment);
}
```

```
char* PolicyBO_Impl::comment ()
{
   return ::CORBA::string_dup( fDataObject->comment() );
}
```

Other Methods: Other business logic methods which access essential state data do so similarly to the *comment* attribute mentioned previously: they use the general fDataObject->datamember pattern.

An advantage of this programming model is that the business logic can be written independent of the exact implementation of the DataObject. At this point, you only know which data is to be persistently stored, not where it is stored or how it is accessed. This encapsulation of information inside of the DataObject makes business logic more stable.

Implementing the IManageable Required Methods

You need to implement the following methods, which are described in the IManagedClient::IManageable interface and its parents.

IManageable::getPrimaryKeyString Method

This method returns a ByteString that contains the contents (in the form of a string) of the IManagedLocal::IPrimaryKey subclass for the business object type being implemented. This could be used to extract the identity of a business object for the purpose of using it at a later time to locate the same business object using a findByPrimaryKeyString() method on a Home.

Note: A ByteString created from an object on one machine may not bytewise compare equal to a ByteString created from the same object on a different machine.

You have several options in implementing this method. The best way to implement this method is to create an instance of the IManagedLocal::IPrimaryKey subclass, set the values into this object, and return the ByteString value. Here's an example of how this would be done for the Policy example:

```
::ByteString* PolicyBO_Impl::getPrimaryKeyString()
{
   //Insert Method modifications here
   PolicyKey_var policyKey = PolicyKey::_create();
   policyKey->policyNo(iDataObject->policyNo());
   return policyKey->toString();

   //End Method modifications here
}
```

The previous example assumes that the pattern for dealing with data is either caching, or that there is no data object. A delegating pattern would cause the `k->policyNo(fPolicyNo);` statement to change to a call to the data object. This method must be overridden. No default implementation is provided by the MOFW.

IManageable::getHandleString Method

CORBA provides the ability to invoke object_to_string() on any object. This might be useful for persistently storing a reference to an object. Component Broker introduces the concept of a handle. Handles provide a way to keep track of an object using a mechanism different than that of a stringified object reference, if this is desired.

You are not required to implement handles. The Component Broker default implementation of this method uses a stringified object reference.

There are many ways to define a more stable handle. The following code is an example of an implementation of the getHandleString method for a handle that combines the primary key and the name of the home into a handle.

```
ByteString* Policy BO_Impl::getHandleString
{
   PKHomeHandle_var myHandle = PKHomeHandle::_create();
   myHandel->home(getHome()); // set value of home into handle
   myHandel->primaryKey(getPrimaryKeyString()); // set value of primary key into handle
   return myHandle->to String();
}
```

CosStream::Streamable::externalize_to_stream Method

Component Broker managed objects are always streamable. Streaming can be the basic mechanism used for copying and moving objects.

The externalize_to_stream method is the CosStream::Streamable method that writes the state data of an object into a stream. For business objects that are not using a data object or for those that use a cached data object, streaming is done as follows:

```
::CORBA::Void PolicyBO_Impl::externalize_to_stream
               (::CosStream::StreamIO_ptr targetStreamIO)
{
   targetStreamIO->write_long( fPolicyNo );
   targetStreamIO->write_float( fAmount );
   targetStreamIO->write_string( fComment );

   CORBA::String_var stringifiedInsured =
      CBSeriesGlobal::orb()->object_to_string( fInsured );
   targetStreamIO->write_string( stringifiedInsured );
}
```

For business objects that use a cached data object, use the data in the business object's cache instead of the data in the data object, because the data in the cache is likely to be more up-to-date.

Do not externalize the pointer to the data object that exists in the private data of the implementation interface in the cases where a data object is being used.

If the business object is delegating its state to a data object, it is the data in the data object that is extracted and placed into the stream. The implementation of this for the Policy example is as follows:

```
::CORBA::Void PolicyBO_Impl::externalize_to_stream
               (::CosStream::StreamIO_ptr targetStreamIO)
{
   targetStreamIO->write_long( fDataObject->policyNo() );
   targetStreamIO->write_float( fDataObject->amount() );
   targetStreamIO->write_string( fDataObject->comment() );

   CORBA::String_var stringifiedInsured = CBSeriesGlobal::orb()->
      object_to_string( fDataObject->fInsured() );
   targetStreamIO->write_string( stringifiedInsured );
}
```

If the data for an object involves references to other objects, there are two possible approaches: *shallow* and *deep*. In the shallow approach to streaming, references to contained objects are stringified, and then the string is written out to the stream. This is the approach shown in the previous two code segments. In the deep approach, contained objects are asked to stream themselves into the same stream as the rest of the object's attributes by invoking externalize_to_stream() on the contained objects.

You should use the shallow approach. If a contained object has many attributes, such as lots of state data, and if it contains references to other objects, it is likely that the deep approach is slower and results in a larger stream (object). In addition, the shallow approach makes internalization more straightforward, as discussed in the following section.

CosStream::Streamable::internalize_from_stream Method

The internalize_from_stream method on CosStream::Streamable needs to be written so that it can read the values that the externalize_to_stream method placed into the stream. The values must be read in the same order that they were written. The example for the caching case and the case when no data object is present follows:

```
  ::CORBA::Void PolicyBO_Impl::internalize_from_stream(::CosStream::StreamIO_ptr
                                                         sourceStreamIO, ::CosLifeCycle
                                                         ::FactoryFinder_ptr, there)
  {
     ::CORBA::Long tempPolicyNo = sourceStreamIO->read_long();
     if ( fPolicyNo == tempPolicyNo )
     {
        fAmount = sourceStreamIO->read_float();
        fComment = sourceStreamIO->read_string();

        CORBA::String_var stringifiedInsured =
           sourceStreamIO->read_string();
        CORBA::Object_var obj = CBSeriesGlobal::orb()->
           string_to_object( stringifiedInsured );
        fInsured = PolicyHolder::_narrow(obj);
     }
     else
        // throw an exception which says that this is the
        // wrong object to load this stream into
  }
```

Note: Do not change the key attributes of a Component Broker business object.

In the delegating data object case, the code for the Policy example is as follows:

```
  ::CORBA::Void PolicyBO_Impl::internalize_from_stream(::CosStream::StreamIO_ptr
                                                         sourceStreamIO, ::CosLifeCycle
                                                         ::FactoryFinder_ptr,
there)  {
     ::CORBA::Long tempPolicyNo = sourceStreamIO->read_long();
     if ( fDataObject->policyNo()==tempPolicyNo)
     {
        fDataObject->amount(sourceStreamIO->read_float());
        fDataObject->comment(sourceStreamIO->read_string());

        CORBA::String_var stringifiedInsured =
           sourceStreamIO->read_string();
        CORBA::Object_var obj = CBSeriesGlobal::orb()->
           string_to_object( stringifiedInsured );
        fDataObject->insured(PolicyHolder::_narrow(obj));
     }
     else
        // throw an exception which says that this is the
        // wrong object to load this stream into
  }
```

If the data in the stream is a stringified object reference, as it would be using the shallow approach described in the previous example, then run the string_to_object method on the ORB with the string that is read in from the stream. This is shown in the previous two code segments.

If, however, the data in the stream represents the entire contents of a contained object, as would be the case in the deep approach mentioned previously, then the internalize_from_stream() method would have to do the following:

- Read all of the contained object's attributes from the stream.
- Create a primary key object for the contained object type.
- Initialize the primary key with data from the stream.
- Find a home for the contained object type.
- Find the contained object by invoking findByPrimaryKeyString() on the home.

The extra work involved in the deep approach could be minimized by writing the contained object's primary key string into the stream instead of its contents, but it would still not be as simple as the shallow approach, and would no longer really be a deep approach, but an alternate shallow approach.

The assumption for the internalize_from_stream method is that the object was created before reading in the stream. This also implies that the initForCreation() method was run on the business object.

Summary of IManageable Methods

IManageable and its ancestors introduce a number of methods into the interface of a business object. Table 2 summarizes these methods and the implementation of each. Only the methods that you must implement or may choose to implement are enumerated here. Other methods exist on these interfaces which are not intended to be overridden by the object provider. This table provides a summary:

Table 2. IManageable Method Implementations Table

interface::MethodName	Unit Test Environment	Production
IManageable::getPrimaryKeyString	object provider	object provider
IManageable::getHandleString	default provided by Component Broker	default provided by Component Broker
IManageable::getHome	object provider	default provided by Component Broker
Streamable::externalize_to_string	object provider	object provider
Streamable::internalize_from_stream	object provider	object provider

This might seem like a lot of methods to implement but they are simple methods to implement. Tools can assist in implementing all of the methods described in Table 2. The tool-generated implementation of these methods is sufficient to begin testing in most cases, and can be customized as needed for specific business reasons.

Implementing IManagedObject Required Methods

In addition to the business logic methods, there are MOFW-required methods that must be implemented regardless of which kind of IManagedObject class is chosen as the base class.

initForCreation() Method

The Managed Object FrameWork invokes this method on every managed object when the object is initially created. This method is the MOFW equivalent of a C++ constructor. Unlike a C++ constructor, by the time this method is called, the object's essential state has already been initialized. In Component Broker, an object's state is initialized by a primary key on a createFromPrimaryKey() method, or a copy helper object on a createFromCopy() method.

In addition, if you are inheriting from the IManagedObject interfaces that have a data object (IManagedObjectWithDataObject and the IManagedObjectWithCachedDataObject), then you must get the input parameter that is provided on this method. It is the first thing you should do in the method. The ::_narrow is used so that the business object has access to specific methods introduced for dealing with domain dependent state data.

```
::CORBA::Void PolicyBO_Impl::initForCreation(
::IManagedServer::IDataObject_ptr theDO);
{
   // keep the data object for later use
   fDataObject = PolicyDO::_narrow(theDO);
   // then put other initForCreation code here
   ...
}
```

If you chose a pattern for handling essential state that requires a DataObject, then you need to have a data member in the .ih file (for example, PolicyBO.ih) to hold the pointer to the data object. This is what is set in the previous code segment.

```
PolicyDO* fDataObject;
```

The data object can be used for any initialization tasks that need to be done. There is no guarantee as to the number of attributes in the data object that are validly set. That would be based on whether a copy helper or primary key was used to do the create. Caching business objects should take this opportunity to load the cache for the business object with any relevant values from the data object. At this point, the data object has values for any key or copy helper data that was passed along during the create. In fact, the primary key information must be extracted from the data object and placed into the cache of the business object.

Given that the data object has a default constructor with reasonable initialization, as recommended and as generated by the tools, these lines of code should be in the initForCreation() method:

```
fPolicyNo = fDataObject->policyNo();
fAmount = fDataObject->amount();
fComment = fDataObject->comment();
fInsured = fDataObject->insured();
```

uninitForDestruction() Method

The Managed Object FrameWork invokes this method on every managed object when the object is being destroyed. This method is the MOFW equivalent of a C++ destructor. If the managed object were managing its own resources, this is where it would remove them.

The data object can be used for anything that is needed. Information can be pulled out of the data object at this point. This is also the opportunity to enforce any referential integrity constraints. For example, this might mean removing an object to which it is pointed.

In addition to initForCreation() and uninitForDestruction(), the IManagedObjectWithDataObject and the IManagedObjectWithCachedDataObject interfaces introduce the initForReactivation() and uninitForPassivation() methods. These are described next.

initForReactivation() Method

The Managed Object FrameWork (MOFW) invokes this method on every managed object when the object is being recreated in storage. This is the MOFW equivalent of an operating system paging storage in from disk. Because the managed object is being put into storage, it needs to make sure that any resources that it was managing are ready to be used. For example, if the managed object were managing its own network connection, the implementation of this method would re-establish the connection.

In addition, if you are inheriting from the IManagedObject interfaces that have a data object (IManagedObjectWithDataObject and the IManagedObjectWithCachedDataObject), then you must get the input parameter that is provided on this method. This is the first thing you should do in the method. The ::_narrow is used so that the business object has access to specific methods introduced for dealing with domain dependent state data.

```
::CORBA::Void PolicyBO_Impl::initForReactivation(
   ::IManagedServer::IDataObject theDO);
{
   // keep the data object for later use
    fDataObject = PolicyDO::_narrow(theDO);
   // then put other initForReactivation code here
   ...
}
```

While it is necessary to get the pointer to the data object in this method, the data object should not be used in any way during this method. This method has no access to the essential state stored in the data object.

uninitForPassivation() Method

The Managed Object FrameWork invokes this method on every managed object when the object is temporarily removed from storage. This is the MOFW equivalent of an operating system paging storage out to disk. Because the managed object is removed from storage, it clearly is not using any resources that it is holding. You have the choice of releasing any held resources at this time, or continuing to hold on to them. If the managed object is not managing any resources, or if you choose to hold onto those resources, then no implementation for this method is required.

The data object cannot be used or accessed during the execution of this method.

In addition to the methods already described, IManagedObjectWithCachedDataObject introduces two additional methods. These are the syncToDataObject() and syncFromDataObject() methods.

syncFromDataObject() Method

The purpose of syncFromDataObject() is to load the business object with the data that is contained in the data object. In “initForCreation() Method” on page 71 and “initForReactivation() Method,” a pointer to the data object fDataObject is set. In this method, the data object must be accessed to load up the local state data into its cache, as the name of the base class for this business object implies (IManagedObjectWithCachedDataObject). The following code is placed in the syncFromDataObject for the Policy example:

```
  fPolicyNo = fDataObject->policyNo();
  fAmount = fDataObject->amount();
  fComment = fDataObject->comment();
  fInsured = fDataObject->insured();
```

Look in the implementation interface for the business object being constructed and ensure that all of the data is properly loaded from the data object. This method is called by the Component Broker server, but must be implemented by the object provider.

MOFW does not let you invoke other methods on the business object's public interface from within the implementation of this method.

Observe that the following lines of code are the same in initForCreation() and syncFromDataObject().

```
  fAmount = fDataObject->amount();
  fComment = fDataObject->comment();
  fInsured = fDataObject->insured();
```

To avoid duplicating this code, introduce a method in your business object called initializeState() that is called from both initForCreation() and syncFromDataObject().

syncToDataObject()

The purpose of syncToDataObject() is to flush the data from the business object back to the data object. This puts the data object in a state in which it can deal with the underlying resource manager, and ensure that this data is properly stored persistently, using the right transaction interfaces to the underlying resource manager. The syncToDataObject() method for the Policy example would look like:

```
  fDataObject->policyNo(fPolicyNo);
  fDataObject->amount(fAmount);
  fDataObject->comment(fComment);
  fDataObject->insured(fInsured);
```

This method is called by the Component Broker server, but must be implemented by the object provided.

The syncFromDataObject() and syncToDataObject() methods are called by the Component Broker server at the appropriate times. They should not be called by the business object in any of its methods. The server calls syncToDataObject() at all appropriate times before passivation to ensure the data is properly stored back to the underlying resource manager. It calls syncFromDataObject() at all appropriate times after reactivation to properly load the business object's cache. These methods should only contain logic that pertains to the values of the business object's cache.

Summary of IManagedObject Methods

All of the methods defined in the IManagedObject subclass chosen must be implemented. They have been described in the previous sections. These methods should be called only by the Component Broker server. See “Sample Framework Flows” on page 203 for further information.

Implementing the Primary Key Class

Each business object must have an associated primary key class to be used in Component Broker. There are several ways to develop a key class. The most prevalent way is to create a key class based on the MOFW base class IManagedLocal::IPrimaryKey.

The IManagedLocal.idl file provides a set of interfaces for keys. IPrimaryKey is the class that you want to subclass from in order to create a key class that works with your business object class. For example, in the insurance policy case, a key class like this:

```
interface PolicyKey : IManagedLocal::IPrimaryKey
{
   attribute long policyNo;
   #pragma meta PolicyKey localonly
}; // end interface PolicyKey
```

In this example, the Primary Key of the Policy object consists of a single attribute, *policyNo*. The Primary Key of an object can consist of one or more public attributes. Protected or private attributes cannot be used as part of a primary key. For example, the Primary Key of a Beneficiary object consists of both the name of the beneficiary and the policy number of the policy for which the person is a beneficiary.

To accommodate the key attributes, protected or private data should be declared inside of the implementation binding header file (.ih). For the PolicyKey in the previous example, put the following in the PolicyKey.ih file:

```
::CORBA::Long      fPolicyNo;
```

In the PolicyKey_I.cpp file, the attributes would be implemented as follows:

```
::CORBA::Void PolicyKey_Impl::policyNo(::CORBA::Long policyNo)
{
   fPolicyNo = policyNo;
}

::CORBA::Long PolicyKey_Impl::policyNo()
{
   return fPolicyNo;
}
```

CosStream::Streamable Methods

The other main requirements for the implementation of this class are the streaming methods as shown in the following example:

```
::CORBA::Void PolicyKey_Impl::externalize_to_stream
      (::CosStream::StreamIO_ptr targetStreamIO)
{
   // Insert Method modifications here
   targetStreamIO->write_long(fPolicyNo);

   // End Method modifications here
}

::CORBA::Void PolicyKey_Impl::internalize_from_stream
      (::CosStream::StreamIO_ptr sourceStreamIO,
       ::CosLifeCycle::FactoryFinder_ptr there)
{
   // Insert Method modifications here
   fPolicyNo=sourceStreamIO->read_long();

   // End Method modifications here
}
```

Although the CosStream::Streamable methods and the toString() and fromString() methods introduced by Component Broker might appear to work independently, they actually work well together. Object providers generally implement the internalize_from_stream() and externalize_to_stream() methods on all objects that are constructed. For primary keys and copy helpers, call toString() and fromString() when an object must be flattened to bits or revived from bits. The toString() and fromString() are actually implemented by the MOFW by calling the Streamable methods. This ensures proper flattening of all CORBA types and proper handling of code pages when the bits are moved between machines. The ByteString data type used by fromString() and toString() is really a wrapper over the data format used by CORBA, and therefore Component Broker, to move data between processes.

Implementing IKey::getName

The implementation of this method returns the class name. For the PolicyKey class, the implementation would look something like this:

```
char* PolicyKey_Impl::getName()
{
   //Insert Method modifications here
   return CORBA::string_dup("PolicyKey");

   //End Method modifications here
}
```

Implementing IKey::isEqualToKey

Component Broker provides a default implementation of this method that compares the stringified values of the keys. It can be optionally implemented by the object provider. The implementation of this method should use the get method on the keys and compare the values to see if the key is for the same object. The following would be an implementation of this for the PolicyKey class.

```
::CORBA::Boolean PolicyKey::isEqualToKey(IKey inKey)
{
   PolicyKey_var pk = PolicyKey::_narrow(inKey);
   if ( pk->policyNo() == fPolicyNo )
      return 1;
   else
      return 0;
}
```

Implementing IKey::isEqualToKeyString

Component Broker provides a default implementation of this method that compares the stringified values of the keys. It can be optionally implemented by the object provider. This method is similar to the previous one, but deals with the stringified version of the key. The implementation is shown in the following example:

```
::CORBA::Boolean PolicyKey::isEqualToKeyString(ByteString inString)
{
   PolicyKey_var tempKey = PolicyKey::_create();
   tempKey->fromString(inString)
   if (tempKey->policyNo() == fPolicyNo)
      return 1;
   else
      return 0;
}
```

Summary of Key Class Construction

This primary key class enables you to use the createFromPrimaryKeyString() and findByPrimaryKeyString() methods that are part of the IManagedClient::IHome interface. To pull it all together now, the following shows the creation of a key, the creation of an object based on this key, and usage of this object.

```
// show a simple usage of createFromKeyString assuming you have a home

PolicyKey_var aPolicyKey = PolicyKey::_create();
aPolicyKey->policyNo(1234);

Policy_var aNewPolicy;
IManagedClient::IManageable_var aManageable;

aManageable = thePolicyHome->
      createFromPrimaryKeyString(aPolicyKey->toString()));
aNewPolicy = Policy::_narrow(aManageable);

aNewPolicy->amount(123.45);
...
...
```

Information about how the value of a key gets mapped to a specific business object is contained in Component Broker and in the DataObject implementation. Details on the data object implementation and this important mapping are in Chapter 11, “Assembling and Installing Business Objects on AIX and Windows NT” on page 197.

Other Ways to get the KeyClass

Some containers come with a predefined subclass of IManagedLocal::IPrimaryKey that works for all business object types that are to be stored in a particular container. If you are planning to install the business object into this kind of container, then you should consult the documentation for that container to determine the kind of key that needs to be used when dealing with business objects that live in that container.

See “Local-Only Development Process” on page 78 for information on building local-only objects such as primary keys.

Implementing the Optional Copy Helper Class

In addition to the createFromPrimaryKeyString() method that is part of the IManagedClient::IHome interface, there is also a method called createFromCopyString(), which supports another creation method for a business object. creation. This method is available to make creating objects more efficient. Rather than running createFromPrimaryKeyString() on the IManagedClient::IHome followed by a series of remote set*xxxx*() methods to load an object, createFromCopyString() enables the creation of a local transient object that is externalized and passed in its entirety to the server in one call.

To create a copy helper class, subclass from IManagedLocal::INonManageable. This is the base class for local-only objects that interact with and act like CORBA objects, support streaming, but are not accessible remotely. Following is an example of the interface that gets declared in IDL for the PolicyCopy class.

```
interface PolicyCopy : IManagedLocal::INonManagable
{
   attribute long     policyNo;
   attribute float  amount;
   attribute float  comment;
}
```

In many ways, the development process for simple copy helper classes is the same as that required for implementing primary key classes. The only difference is the fact that typically more attributes are placed into a copy helper class. A pragmatic suggestion for building your first copy helper class is to borrow pieces of the PrimaryKey subclass and modify it to become a copy helper. What is added is probably based on the pieces of the basic business object, for example Policy, that might be present in a copy.

This interface implies that the copy helper class allows the loading of all of the state of the Policy business object interface.

This interface then has an implementation with setters and getters for each of the attributes. In addition, the internalize_from_stream and externalize_to_stream methods need to be implemented. These implementations are shown in the following code segments.

```
class PolicyCopy_Impl : public virtual ::PolicyCopy_Skeleton,
      public virtual IManagedLocal_INonManageable_Impl
{
   public:
      // excerpts from the C++ interface used for implementation
      // which is probably named PolicyCopy.ih
      ...
      virtual ::CORBA::Void externalize_to_stream(
            CosStream::StreamIO targetStreamIO);

      virtual ::CORBA::Void internalize_from_stream(
            CosStream::StreamIO sourceStreamIO,
            CosLifeCycle::FactoryFinder there);

      // add getters/setters and isReady() to this list
      ...
      ...
   protected:
      ::CORBA::Long        copyPolicyNo;
      ::CORBA::Float       copyAmount;
      ::CORBA::String_var  copyComment;
};
```

Then, the implementation class for this copy object has this:

```
::CORBA::Void PolicyCopy_Impl::externalize_to_stream(
      CosStream::StreamIO targetStreamIO)
{
   targetStreamIO->write_long(copyPolicyNo);
   targetStreamIO->write_float(copyAmount);
   targetStreamIO->write_string(copyComment);
}

::CORBA::Void PolicyCopy_Impl::internalize_from_stream(
      CosStream::StreamIO sourceStreamIO,
      CosLifeCycle::FactoryFinder there)
{
   copyPolicyNo = sourceStreamIO->read_long();
```

```
      copyAmount = sourceStreamIO->read_float();
      copyComment = sourceStreamIO->read_string();
   }

   // add getters/setters and isReady() to this list
   ...
   ...
```

The copy helper is complete. It is a lightweight object that encapsulates the subset of the business object's state data that is fed in on a createFromCopyString() call. In addition to this, the implementation of the CosStream::Streamable methods enables this class to use the Externalization Service to get the contents ready for sending to the server.

A copy helper class should have a default constructor that initializes values to appropriate default values. This should help in the case where only partial information is available when the copy is built. A copy helper must contain all the information necessary to create a primary key which allows a home to create a unique business object from the copy data.

See "Local-Only Development Process" for more details on building these local-only copy helper objects.

Local-Only Development Process

Both the Copy Helper classes and the Primary Key classes are developed using a variant of the development processes used for building the business object implementations. It is actually a simpler process because these are local-only objects which are not accessible remotely. All interfaces (specified in IDL), inheriting specifically from IManagedLocal::ILocalOnly, IManagedLocal::INonManageable, or any other subclass of IManagedLocal::ILocalOnly follows this development process. A more detailed description of this process is discussed in "The Local-Only Development Process" on page 252.

For special local-only notes for copies and keys, every helper class must be written in IDL and generate a C++ binding (and implementation) because the Component Broker server is built using C++ and these classes get used on the server by the Component Broker run time. It is up to the object provider to determine if a Java version (binding and implementation) of the helper classes is also desired for use on a Java client.

Summary

This chapter explained the minimum important artifacts necessary to implement a business object. At the end of these activities, you should have the following:

- IDL Files
  - Interface for client (Policy.idl).
  - Interface for implementation (PolicyBO.idl).
  - Interface for essential state (PolicyDO.idl).
  - Interface for Primary Key class (PolicyKey.idl).
  - Interface for Copy Helper class (PolicyCopy.idl).
- Implementation Files
  - Implementation of business logic and MOFW-required methods (for example, PolicyBO.ih and PolicyBO_I.cpp that contains the PolicyBO_Impl C++ class).
  - Implementation of the Primary Key class (for example, PolicyKey.ih and PolicyKey_I.cpp).
  - Implementation of the CopyHelper class (for example, PolicyCopy.ih and PolicyCopy_I.cpp).

- Client and server bindings (for example *xxxx*_C.cpp and *xxxx*_S.cpp files) for the interface classes and implementation classes. The IDL emitter generates these bindings. Because Primary Key and Copy Helper classes are local-only, they only require _C.cpp files for bindings.
- Usage Bindings
  - Policy.hh
  - PolicyBO.hh
  - PolicyKey.hh
  - PolicyCopy.hh

These are the artifacts necessary to begin either testing the business object in the Unit Test Environment, or preparing a real managed object to be installed into an application adaptor. Table 3 on page 80 summarizes what is specified and from where in the MOFW the abstractions in the IManagedServer module are inherited.

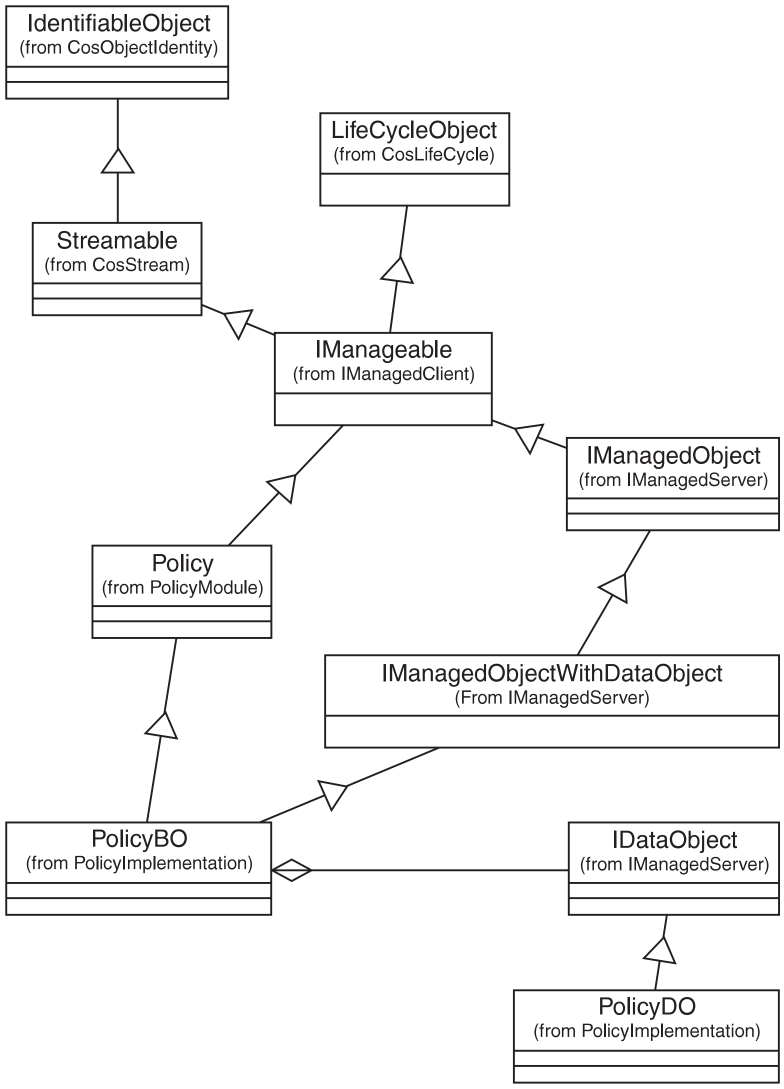


Figure 32. Basic Artifacts in Business Object Development and their MOFW Inheritances

Table 3. Artifacts Table for Business Object

Interface Name	**Implementor**	**Server DLL Name**	**Client DLL Name**	**Comments**
Policy	Object provider	PolicyBO.dll should hold Policy_S.obj	PolicyClient.dll should hold Policy_C.obj	Interface only
PolicyBO	Object provider	PolicyBO.dll should hold PolicyBO_S.obj	See note 1	Real business logic and MOFW required methods
PolicyDO		n/a	See note 1	Documents essential state
Data object		n/a	See note 1	Interface only
Manageable	Component Broker provides some	Manageable.dll	ManageableClient.dll	Interface and some default implementation
	Object provider must do some in its PolicyBO	PolicyBO.dll		
Identifiable	Object Services	Identifiable.dll	IdentifiableClient.dll	
LifeCycle		n/a		
Streamable	Component Broker provides some	Manageable.dll	StreamableClient.dll	
	Object provider must do some in its PolicyBO	PolicyBO.dll		
IManagedObject			See note 1	Interface only
IManagedObjectWith-DataObject	Object Provider (in its PolicyBO)	See PolicyBO.dll	See note 1	
PolicyCopy	Object Provider	PolicyCopy.dll	Same DLL on client and server	ILocalOnly
PolicyKey	Object Provider	PolicyKey.dll	Same DLL on client and server	ILocalOnly

Notes:

1. Not needed. You do not need a separate DLL or OBJ file for these as they are used only from inside the server and going through the _S.cpp is sufficient.

For the client DLL name you need to have the _C.cpp client side binding that corresponds to the IDL file when it is introduced regardless of where the actual implementation takes place. When it is server only, the _C.cpp included by the _S.cpp is sufficient.

Additional Information Business Object Creators Should Know

Developing the business logic that implements business object interfaces is done using as much of the C++ language as is appropriate for optimally supporting the interfaces. Use native language class libraries to assist when needed. Choices on how to best implement the business logic are yours to make.

Note: Do not create separate threads. The Component Broker server environment in which the business objects reside is complex. Component Broker must control the threading environment. Creation of additional threads from within business logic could cause unexpected results on the Component Broker server.

Where to Next?

In order to have a business object that can be tested and installed into a Component Broker server, some additional steps must be taken. First, however, you might want to unit test the business object. If you are ready to unit test, proceed to Appendix D, “Unit Test Environment” on page 333. If you want to provide more advanced features for clients to use and to leverage additional MOFW features as part of the implementation of your managed object, then see Chapter 6, “MOFW Client Programming Model – Advanced Concepts” on page 83. See Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105 for information on how to add these advanced features to a business object.

Chapter 6. MOFW Client Programming Model – Advanced Concepts

This chapter includes the following topics:

For further information on Quality of Service Interfaces, see "Expanding the Client Programming Interface" on page 246.

Transactions

Transactions are like a contract that binds a client to one or more servers. They bracket a set of system behaviors or actions that have the following properties:

Atomicity
A group of actions behave in an all-or-none fashion as an indivisible unit of work.

Consistency
After a transaction executes, the system is left in a correct state, assuming that the system was in a correct state prior to the transaction.

Isolation A given transaction's behavior is unaffected by other transactions and system activity occurring concurrently.

Durability
When a transaction commits, its effects are permanent or persistent. Because transactional models involve the concept of bracketing system behaviors, they provide commands for indicating transaction boundaries. All transactions have a start and an end that can involve committing to a change in system state or aborting the changes and rolling back to the pre-transaction state of the system.

The Component Broker Object Transaction Service provides standard primitives for transactional applications in a distributed object environment. When the Transaction Service is used in conjunction with a persistent storage medium and with some control over concurrent access to resources, you have the basics for creating robust business applications. Key elements of the Transaction Service are:

- Basic Transaction Service operations.
- Integration with the server run time.
- Client support.

Note: For additional information about the Transaction Service, see the *Component Broker Advanced Programming Guide*.

CosTransactions Module

The Component Broker CosTransactions module provides the Transactions interface defined by OMG. Some of the operations supported by CosTransactions include:

- Controlling the scope and duration of a transaction.
- Allowing multiple objects to be involved in a single, atomic transaction.
- Coordinating the completion of transactions.
- Performing recovery of transaction states following restart of failed processes.

The primary interfaces in the CosTransactions Module are:

Current Interface
The Current interface defines methods that allow a client of the Transaction Service to explicitly manage the association between threads and transactions. It also defines methods that simplify the use of the Transaction Service for most applications. These methods can be used to begin and end transactions, and to obtain information about the current transaction (see “A Simple Example” on page 85).

TransactionFactory Interface
The TransactionFactory interface defines a single create method that allows the transaction originator to create a new top-level transaction without it being associated with the current thread. It is primarily intended for creating transactions remotely.

Control Interface
The Control interface defines methods to allow a program to explicitly manage or propagate a transaction. An object supporting the Control interface is implicitly associated with one transaction only.

Terminator Interface
The Terminator interface defines methods to complete a transaction, either by requesting commitment or demanding rollback. Typically, these methods are used by the transaction originator. An object that supports the Terminator interface is implicitly associated with one transaction only.

Coordinator Interface
The Coordinator interface provides common methods for top-level transactions and subtransactions. Participants in a transaction are typically either recoverable objects or agents of recoverable objects, such as subordinate coordinators. An object supporting the Coordinator interface is implicitly associated with one transaction only.

RecoveryCoordinator Interface
The RecoveryCoordinator interface allows recoverable objects to drive the recovery process in certain situations. Each instance of this class is implicitly associated with a single resource registration, and can only be used by that resource in the particular transaction for which it is registered.

Resource Interface
The Resource interface defines the operations invoked by the Transaction Service, during transaction completion, on each resource registered using the register_resource() method of the Coordinator interface.

SubtransactionAwareResource Interface
The SubtransactionAwareResource interface defines the operations invoked by the Transaction Service, during subtransaction completion, on each resource registered using the register_subtran_aware method (or register_resource) of the Coordinator interface.

Synchronization Interface
The Synchronization interface defines the operations invoked by the Transaction Service during transaction completion on each resource registered using the register_synchronization method of the Coordinator interface.

TransactionalObject Interface
The TransactionalObject interface is used by an object to indicate that it is transactional. By inheriting from the TransactionalObject interface, an object indicates that it wants the transaction context associated with the client thread to be propagated on requests to the object.

These interfaces show the breadth of function available within the CosTransactions Module. The MOFW encapsulates the transactional interfaces and provides a programming interface to transactions. Most of what you need to know is shown in "A Simple Example."

A Simple Example

Code segments in this section show how a client could use some of the Transaction Services. The example uses Policy objects.

The following code segment performs the initialization that is required to use transactions.

```
#include <CosTransactions.hh>

CosTransactions::Current_ptr currentTransaction = NULL;
CORBA::Current_ptr orbCurrentPtr = NULL;

CBSeriesGlobal::Initialize();
CORBA::ORB_ptr orb = CBSeriesGlobal::orb();

orbCurrentPtr = orb->get_current("CosTransactions::Current");
currentTransaction = CosTransactions::Current::_narrow(orbCurrentPtr);
```

You can proceed to find, create, and use a factory as shown in earlier examples. Before attempting to create a Policy managed object, the begin() method can be called on the *currentTransaction* pointer to indicate the start of a transaction. When using transactions it is advisable to be begin the transaction early in the processing cycle. All method calls that might result in a database access must be always called within the scope of a transaction. For example, methods such as 'findByPrimaryKeyString' and 'createFromPrimaryKeyString' may result in database accesses, it is safer to begin the transaction before making these calls. Beginning transactions early in the processing cycle will hide differences in implementation differences between Application Adapters and other specific implementations in later releases.

```
PolicyKey_var keyVar = PolicyKey::_create();
keyVar->PolicyNo(55555);
theKeyString = keyVar->toString();

// Start the transaction now
currentTransaction->begin();

⋮
     /* Assume that the policy home is found  */

mVar = myPolicyHome->createFromPrimaryKeyString(*theKeyString);
myPolicy = Policy::_narrow(mVar);
myPolicy->amount(25000.00);
```

When the Managed Object has been created, methods can be called on it and its data can be manipulated as desired, all within the scope of the transaction that was started in the previous example. To indicate the termination of the transaction, the commit method is called and the Policy is released.

```
CORBA::Boolean inValue = 1; // Report heuristic exceptions

// End the transaction now
currentTransaction->commit(inValue);
```

The previous segments assume that the data manipulations resulted in desired updates to the data. The commit method completes the current transaction by making these changes to the data permanent. But what if a condition occurred in which the client wanted to end the transaction without any changes?

The following code example shows how the rollback method can be used to terminate a transaction without an update to the data. Two policies are created and have their data set. An `if` test determines whether the changes are committed or rolled back.

```
Policy_var aPolicy;
Policy_var aPolicy2;
IManagedClient::IManageable_var moVar;
IManagedClient::IManageable_var moVar2;
ByteString *theKeyString;
IExtendedLifeCycle::FactoryFinder_var myFinder;
CORBA::Object_var it;
IManagedClient::IHome_var policyHomeVar;
float inAmount;
CORBA::Boolean inValue;

CosTransactions::Current_ptr currentTransaction = NULL;
CORBA::Current_ptr orbCurrentPtr = NULL;

CBSeriesGlobal::Initialize();
CORBA::ORB_ptr orb = CBSeriesGlobal::orb();

orbCurrentPtr = orb->get_current("CosTransactions::Current");
currentTransaction = CosTransactions::Current::_narrow(orbCurrentPtr);
currentTransaction->set_timeout(600);

it = CBSeriesGlobal::nameService()->resolve_with_string(
     "host/resources/factory-finders/host-scope");
myFinder = IExtendedLifeCycle::FactoryFinder::_narrow(it);
it = myFinder->find_factory_from_string(
     "PolicyDefaultTransDB2.object interface");
policyHomeVar = IManagedClient::IHome::_narrow(it);

PolicyKey_var theKey = PolicyKey::_create();
theKey->policyNo(12345);

theKeyString = theKey->toString();

// Start the transaction now
currentTransaction->begin();

moVar = policyHomeVar->createFromPrimaryKeyString(*theKeyString);
   theKey->policyNo(99999);
   theKeyString = theKey->toString();
moVar2 = policyHomeVar->createFromPrimaryKeyString(*theKeyString);
   aPolicy = Policy::_narrow(moVar);
   aPolicy2 = Policy::_narrow(moVar2);

cout << "Enter amount for first policy" << endl;
cin >> inAmount;
aPolicy->amount(inAmount);
cout << "Enter amount for second policy" << endl;
cin >> inAmount;
aPolicy2->amount(inAmount);

if(aPolicy->amount() == aPolicy2->amount())
{
   // End transaction now with now changes to data
   try
   {
            current->rollback();
   }
   catch(CosTransactions::NoTransaction)
   {
            // No Transaction to rollback. ...
   }
   catch(...)
   {
      // Rollback failed ...
   }
}
else
{
   // End transaction now and update data
   currentTransaction->commit(inValue);
}
```

Transactions, Exceptions, and Timeouts

“A Simple Example” on page 85 provides a basic usage view of the transactional interfaces. This section provides additional guidelines on application structuring in the area of timeouts and exception handling.

Consider the following code example to explain why the code is structured in this way:

```
// get current See note 1
CBSeriesGlobal::Initialize();
CORBA::Current_var orb_current = CBSeriesGlobal::orb()->get_current("CosTransactions::Current");
CosTransactions::Current_ptr current;
current= CosTransactions::Current::_narrow(orb_current)
current->set_timeout(180); // See note 3
// you could have a loop starting here...
try
{
   current->begin(); // See note 2
   .... do work
   if (businessLogicSaysCommit)
       current->commit(1); // See note 2
   else
   try                     // See note 4
   {
            current->rollback();
   }
   catch(CosTransactions::NoTransaction)
   {
      // No Transaction to rollback. ...
   }
   catch(...)
   {
      // Rollback failed ...
   }
}
catch (const ::CORBA::SystemException &se)    // See note 5
{
   cout << "System Exception" << se.id() << endl;
   try
   {
      current->rollback();
   }
   catch(CosTransactions::NoTransaction)
   {
      // No Transaction to rollback. ...
   }
   catch(...)
   {
      // Rollback failed ...
   }
}
catch(...)  // See note 6
{
   try
   {
      current->rollback();
   }
   catch(CosTransactions::NoTransaction)
   {
      // No Transaction to rollback. ...
   }
   catch(...)
   {
      // Rollback failed ...
   }
   // do whatever you want to do next..
}
```

What does all of this mean? In the previous example, the numbers in the comments correspond to the following list items:

1. Get the current. This is the same as always. Get it once for performance reasons.
2. Begin and Commit transactions. Bracket units of work in accordance with the requirements of your business logic and domain needs. In other words, if you want to be able to rollback a set of operations, put them in a begin or commit block. The issue is how much time is normally expected between the begin and commit. You must realize that there is some level of resource locking that goes on while there is a transaction inflight. Shorter transactions will increase throughput, and so on.
3. This is an important line of code. This code signals to the server that this is how long (in seconds) the server should wait before taking matters into its own hands. This means that the server will roll back without any user code having specified rollback. If a timeout is not specified via this method call, a platform-dependent default value is used.
4. Rollback can be issued at any time based on the needs of the business logic. This undoes changes made since the last begin.

5. When a system exception occurs, you should code a rollback() in the catch block or some routine that the catch block calls.
6. The exceptions that make it to the final catch block, by definition, were thrown from within the client program. Anything that flowed over a wire would have been remapped to a CORBA exception and would have been caught in item number 5. A rollback here will also work properly and release locks. You should note that a rollback can be issued for exceptions that are not system exceptions that come back from the server. Things like IManagedClient:INoObjectWithKey can be caught and followed by a rollback().

Session Service

OS/390 Component Broker does NOT support session service.

The session service defines the notion of a session. It provides a mechanism for grouping a set of operations together as a logical unit. A session is conceptually similar to a transaction as defined and supported by the transaction service. Sessions differ from transactions in the following ways:

- A session is defined on an application scope rather than on an individual transaction scope. This provides a mechanism for checkpointing groups of persistent objects for which no externally coordinated transactional update is available.
- Sessions do not define an atomically recoverable commit scope as do transactions; sessions provide some services for checkpointing and clean conclusions of applications but do not provide the same level of application durability as transactions.
- A session can use an application profile. An application profile consists of attributes that define its properties, such as requirements for data concurrency, duration, visibility, update, execution priority, and resource dependencies.

Sessions and transactions can be used together. Transactions can be used within sessions to provide greater levels of durability within applications.

For additional information about the session service, see the *Component Broker Advanced Programming Guide*.

A Simple Example

The following example demonstrates how a single-threaded client can begin a session, perform some work, and end the session, checkpointing any non-transactional work that occurs within the session.

```
CORBA::Current_var current;
ISessions::Current_var sessionCurrent;
// Get the current for this thread of execution
// from the ORB and narrow to the session current.
current = CBSeriesGlobal::orb()->get_current("ISessions::Current");
sessionCurrent = ISessions::Current::_narrow(current);
```

Set a Time Limit for All New Sessions

The ISessions::Current interface has a setSessionTimeout() operation that enables the application to set a time limit for all sessions that are subsequently started. The default time limit is zero. That is, sessions can run indefinitely. To set a time limit for all new sessions, invoke the setSessionTimeout() operation on the ISessions::Current object, passing the new timeout value.

```
// Set the session timeout.
sessionCurrent->setSessionTimeout(100);
```

Begin a Session

You begin a session by invoking the beginSession() operation on the ISessions::Current interface. You should supply a text string representing the name of your application or the application profile under which you want the session to operate. If the specified profile cannot be found or if you specify an empty string, then a default application profile is used.

The application profile specifies certain expectations about how the session will behave. This information can be used in combination with the capabilities and policies of the running system to produce a set of execution decisions that optimize the total performance and throughput of the system. Once the session has been started you can perform any number of operations on business objects within the session. All operations invoked within the session are performed with the same session context.

```
// Begin a session context.
sessionCurrent->beginSession("LifeInsuranceApplication");

try
{
   // ... do the methods that will be executed under the
   // session ...
}
catch (ISessions::SessionResetForced)
{
   // The session was forced to reset mid-stream, probably the
   // session timeout tripped, or a session resource
   // encountered a significant error and had to force the session
   // reset.
};
```

End a Session

You complete the session by invoking the endSession() operation on the sessionCurrent. Normally, you specify the *EndModeCheckpoint* end-mode with this operation. This drives all the sessionable resources used within the session to save their state changes persistently, through embedded operations on the underlying data system.

To reset the session, end it without saving any of the changes that occurred since your last checkpoint (or since the beginning of the session if you did not perform any checkpoints). Specify the *EndModeReset* end-mode with the endSession() request.

EndModeCheckpoint and *EndModeReset* have no bearing on any transactions issued within the session other than to ensure that the session is terminated before it ends. However, if you encounter severe errors in your processing, you can end the session with *EndModeResetForce*. This forces the session to be reset immediately, including rolling back any outstanding transactions.

```
// End the session context, including checkpointing any activity that
// occurred during the session.

try
{
   sessionCurrent->endSession(EndModeCheckpoint,1);
}

// Catch a variety of exceptions. See the Advanced Programming Guide
// for details.
```

Other Information

Further information on these and other topics regarding the sessions service can be found in the *Component Broker Advanced Programming Guide*. For example, the following information is available:

Suspending and resuming sessions
There may be occasions when you want to switch the session under which you are operating. You can do this by suspending the current session and starting a new one. Later, you can resume the original session.

Explicit and implicit propagation of session context
This section describes different techniques of assigning context information for multi-threaded applications.

Checkpoint and reset a session context
Sessions can be checkpointed to save intermediate results to persistent data storage. Sessions can also be reset to revert the state of operations performed within the session.

Registering sessionable resources
An application can introduce resources and explicitly register the resources with a session.

Visibility rules
Sessions can be run concurrently. Visibility rules define how the data interactions between these concurrent sessions are defined.

Queries, Iterations and Specialized Homes

Chapter 4, “MOFW Client Programming Model” on page 33 introduced generic homes as a facility for creating business objects and for finding business objects based on their primary key. This chapter discusses how a client of a business object could use other features that a home could support.

If the object provider has built a specialized home, the usage pattern for that home is different than that of a generic home, because a specialized home has additional methods that can be used by clients. Besides the additional methods available to the client, there is also a change in how this specialized home is found.

Using Iterated Homes-Specific Functions

Many times an application needs access to multiple objects. In “Using Sets of Objects” on page 44, an example of iterating through an arbitrary collection of business objects is presented. This iteration assumes that all the objects are inserted into an IManagedReferenceCollection.

If you wanted to examine a group of homogenous objects (for example, every Claim) and perform actions on those objects that met certain criteria, the Component Broker programming model extends the concept of iteration to IHomes. The following code segment shows the usage of an iterated home.

```
CORBA::Object_var obj;
IExtendedLifeCycle::FactoryFinder_var myFinder;
IManagedAdvancedClient::IIterableHome_var policyHome;
IManagedCollections::IIterator_var theIterator;

obj = CBSeriesGlobal::nameService()->resolve_with_string(
      "host/resources/factory-finders/host-scope");
myFinder = IExtendedLifeCycle::FactoryFinder::_narrow(obj);

obj = myFinder->find_factory_from_string(
      "PolicyDefaultTransDB2.object interface");
policyHome = IManagedAdvancedClient::IIterableHome::_narrow(obj);

if ( CORBA::is_nil(policyHome) )
{
   cerr << "ERROR: Application improperly configured. " <<
        "Home for Policies is not iterable." << endl;
}
else
{
   IManagedClient::IManageable_var aBO;
   // Repeat transactional setup.
   currentTransaction->begin();

   // Create a new iterator on the Claim home
   theIterator = policyHome-> createIterator();
   try
   {
      // Loop through the objects in the home
      while ( aBO = theIterator->next() )
      {
         // Get the next element
         Policy_var aPolicy;
         aPolicy = Policy::_narrow(aBO);

         // Do something useful with it.
         cout << "Policy no = " << aPolicy->policyNo() << endl;
         cout << "Policy amount = " << aPolicy->amount() << endl;
      }
   }
   catch (...)
   {
      cerr << "ERROR: Problem occurred using iterator." << endl;
      try
      {
         current->rollback();
      }
      catch(CosTransactions::NoTransaction)
      {
         // No Transaction to rollback. ...
      }
      catch(...)
      {
         // Rollback failed ...
      }
   }
   // After iterating over the entire collection, the iterator is no
   // longer needed. Remove it.
```

```
      theIterator->remove();
      currentTransaction->commit(inValue);
   }
```

In addition to the next() method, IManagedCollections::IIterator provides the following methods with similar functionality (albeit next() is much more concise): hasMoreElements(), more(), nextElement(), nextOne().

While the previous examples are functionally identical (with identical performance), there is another interface for iterating which is functionally similar, but with different performance characteristics. The nextN() method is similar to the nextOne() method, except that instead of returning only one element, nextN() returns as many elements as you request.

This interface might be useful if, for example, an application wants to display information about a fixed number of business objects at a time (limited by the size of a window on a screen). Here is an example of how to use this interface in the same application as the previous example:

```
   char c;
   ICollectionsBase::MemberList_var theBOlist;

   // Loop through the objects in the home.
   while ( theIterator->nextN(5, theBOlist) || theBOlist->length() > 0 )
   {
      for ( int n = 0; n < theBOlist->length(); ++n)
      {
         // Narrow the next element obtained by nextN()
         aPolicy = Policy::_narrow(theBOlist[n]);

         // Do something useful with it
         cout << "Policy # " << aPolicy->policyNo();
         cout << "amount is " << aPolicy->amount() << endl;
      }
      // Having displayed as many policies as would fit in the window,
      // the application now waits for user input before getting the
      // next N policies and displaying them.
      cout << "press any key (and hit enter) to continue." << endl;
      cin >> c;
   }

   // Because the nextN() call might not have been able to return each time.
```

The previous examples illustrate several ways to iterate through the objects in a home. To understand iteration better, a few general notes about iterators follow:

- When an iterator is created, it is positioned so as to precede the first element.
- The various types of next methods (next(), nextElement(), nextOne() and nextN()) all move the position of the iterator forward, then return the element at the current position.
- There is a current() method that returns the element at the current position without moving the position of the iterator forward. Because the initial position of an iterator precedes the first element, it is an error to invoke current() prior to invoking one of the next methods.
- There is a reset() method which moves the position back to its initial position.
- A home that is iterated is not guaranteed to return its elements in the same order on successive iterations, but it is guaranteed to return each element once only on a given iteration.

Members of Component Broker Homes are always accessible through at least one key (the Primary Key), but may or may not be iteratable. If the collection is based on or wrappers a relational or object-oriented database, then both keys and iterator can be supported. The same is true for most data back-ends (such as files, IMS DL/1 or CICS File Control). If the collection wrappers a set of applications (such as the Customer Management Application in Chapter 2, “Personal Life Insurance Application Example” on page 15), iteration might not be supported by the application. Therefore, the wrappering collection cannot be iterated. Check with your System Administrator to see if your home supports iteration.

Because the LifeCycle object service factory finder interface is used to find homes and this interface has no way of telling an iteratable home from a generic home, a client must be properly configured if its function depends on access to an iteratable home. The client can protect itself against misconfiguration by verifying that the home returned by the factory finder supports the IManagedAdvancedClient::IIterableHome interface. A verification example is:

```
policyHome = IManagedAdvancedClient::IIterableHome::_narrow(obj);
if ( CORBA::is_nil(policyHome) )
{
   // Insert error message here to be displayed here.
}
```

Using Queryable Homes-Specific Functions

Iterating over the business objects in an IIterableHome is fine for some applications. Specifically, if the goal is to perform some operation on every element in the home, then iteration is a good approach. However, if the reason for iterating is to select a subset of all the business objects in the home (and then work with only that subset), then there is a more efficient way to accomplish the same thing: query.

Here the iteration example is used again; however, instead of iterating against all of the objects in a home, you select only the objects in which you are interested (by evaluating a query), and then you iterate over the subset in which you are interested. Because it is sometimes possible for the query to be performed in the database (without bringing into memory every object in the home), using query offers the possibility of significant performance increases over iterating against every object in the home.

```
CORBA::Object_var obj;
IExtendedLifeCycle::FactoryFinder_var myFinder;
IManagedAdvancedClient::IQueryableIterableHome_var policyHome;
IManagedCollections::IIterator_var theIterator;

obj = CBSeriesGlobal::nameService()->resolve_with_string(
      "host/resources/factory-finders/host-scope");
myFinder = IExtendedLifeCycle::FactoryFinder::_narrow(obj);

obj = myFinder->find_factory_from_string("PolicyDefaultTransDB2.object interface");
policyHome = IManagedAdvancedClient::IQueryableIterableHome::_narrow(obj);

if ( CORBA::is_nil(policyHome) )
{
   cerr << "ERROR: Application improperly configured. "
        << "Home for Policies is not queryable."
        << endl;
}
else
{
   // Evaluate a query on the Policy home. The results of the query
   // are returned in the form of an iterator.
   int minPolicyNo = 3;
   char theQuery[80];
   sprintf(theQuery,"policyNo>%d",minPolicyNo);   /* specifies the query itself */
```

```
    // Set up for transactions
    currentTransaction->begin();
    theIterator = policyHome->evaluate(theQuery);
    try
    {
       // Loop through the results of the query
       while ( aBO = theIterator->next() )
       {
          // Narrow the element obtained by nextOne()
          aPolicy = Policy::_narrow(aBO);

          // Do something useful with it.
          cout << "Policy no = "
               << aPolicy->policyNo()
               << endl;
          cout << "Policy amount = "
               << aPolicy->amount()
               << endl;
       }
    }
    catch (...)
    {
       cerr << "ERROR: Problem occurred using iterator."
            << endl;
       try
       {
          current->rollback();
       }
       catch(CosTransactions::NoTransaction)
       {
          // No Transaction to rollback. ...
       }
       catch(...)
       {
          // Rollback failed ...
       }
    }
    // After iterating over the entire collection, the iterator is no
    // longer needed. Remove it.
    theIterator->remove();
    currentTransaction->commit(inValue);
 }
```

The query language used to express the query is SQL with extensions and provisions for being able to query against objects (instead of data only in a database).

Using Atomic Transactions with Query Evaluator

The programming model for method duration transactions (called atomic or automatically start a new transaction) occurs when a method is invoked on the object; the method then will be wrapped within a transaction. That is, prior to invoking the method, a transaction will automatically start. When the method returns the transaction will be committed. The programming model has been extended so that creating or finding an object that is in a container (supporting method duration transactions then creation or finding) will also be wrapped in a transaction.

The programming model has not been expanded to include the query service.

Important: A transaction must have been started prior to using the query service.

For additional information about the query service, see the *Component Broker Advanced Programming Guide* .

More on Iterators

This information addresses the following:

- how iterators over homes and collections are used
- how query iterators work
- how atomic containers are configured
- side effect of end transaction

Managed Object iterators over home collections, iterators over persistent reference collections, query iterators over persistent managed objects and query data array iterators over persistent objects become invalid at the end of the current transaction in which they were created.

When designing objects that will be configured into an atomic container, the object interface should not have attributes or methods which return to the client any of the above types of iterators.

If the object interface does have such iterators, they can be retrieved by a client only if the client explicitly starts a transaction and retrieves and uses the iterator in the same transaction scope.

Note the following example:

```
interface X
{
   IMangagedCollections::IIterator methodX();
}
```

If this interface is implemented by an MO, configured into an atomic container and if my client does not start a transaction, the following statements will result in an exception because the iterator is being used outside the scope of a transaction.

```
IManagedCollections::IIterator_var i = x_ptr -> methodX();
IManagedClient::IManageable_var mo = x_ptr->next();
```

However if the client does the following, it will be valid because the iterator is being retrieved and used in the same transaction scope.

```
currentTransaction->begin();
IManagedCollections::IIterator_var i = x_ptr -> methodX();
IManagedClient::IManageable_var mo = x_ptr->next();
```

Using Keyed Reference Collections

“Using Sets of Objects” on page 44 describes how to use a Reference Collection (that is, IManagedCollections::IReferenceCollection) to maintain references to a set of Managed Objects. If the elements in a Reference Collection require keyed access (that is, Hashtable-like access), a Keyed Reference Collection can be used instead. Keyed Reference Collections are defined by the IManagedCollections::IKeyedReferenceCollection interface.

Keyed and non-keyed Reference Collections have many similarities. Both the IReferenceCollection and IKeyedReferenceCollection interfaces derive from the ManagedCollections::ICommonCollection common base interface that defines a number of methods that apply to both keyed and non-keyed collections. The interfaces include many commonly used methods such as containsElement(), isEmpty(), numberOfElements(), and createIterator(). In addition, Keyed Reference Collections are created using the same specialized home (that is, IManagedCollections::ICollectionHome) as non-keyed collections. The following code segment illustrates the creation of a Keyed Reference Collection:

```
CORBA::Object_var obj;
IManagedCollections::ICollectionHome_var kcHome;
IManagedCollections::ICommonCollection_var cc;
IManagedCollections::IKeyedReferenceCollection_var kc;

obj = myFinder->find_factory_from_string(
     "IManagedCollections::IKeyedReferenceCollection.object
        interface/TransientKeyedReferenceCollectionFactory.object home");
kcHome = IManagedCollections::ICollectionHome::_narrow(obj);
cc = kcHome->createCollection();
kc = IManagedCollections::IKeyedReferenceCollection::_narrow(cc);
```

This code is identical to that shown in “Using Sets of Objects” on page 44 to create non-keyed collections except that the IReferenceCollection interface name is replaced by IKeyedReferenceCollection.

The difference between keyed and non-keyed collections is the way objects are added and accessed. Instead of calling IReferenceCollection::addElement(), the IKeyedReferenceCollection::addElementByString() method is used to add elements to the collection. The addElementByString() method requires two arguments, the object to be added and a stringified key (that is, a ByteString) that is used for identifying the object in the collection. The key can be any appropriate subclass of IManagedLocal::IKey, either general purpose (for example, StringKey), application specific (for example, PolicyHolderIdentifierKey), or, for that matter, the primary key (PolicyHolderPrimaryKey).

For example, assume a general purpose IKey subclass has been defined:

```
interface StringKey : IManagedLocal::IKey
{
   attribute string value;
   #pragma meta StringKey localonly
}
```

Using this key, elements can be added to the Keyed Reference Collection as follows:

```
// Get some policy holders
PolicyHolder_var policyHolderJohn;
PolicyHolder_var policyHolderKatherine;

// Create a key object
StringKey_var theKey = StringKey::_create();
ByteString_var keyString;

// Add policyHolderJohn to the collection with key "John"
theKey->value("John");
keyString = theKey->toString();
kc->addElementByString(policyHolderJohn, keyString);

// Add policyHolderKatherine to the collection with key "Katherine"
theKey->value("Katherine");
keyString = theKey->toString();
kc->addElementByString(policyHolderKatherine, keyString);
```

An element can be retrieved using the getElementByString() method. For example, the following code segment retrieves the policy holder “John”:

```
theKey->value("John");
keyString = theKey->toString();
IManagedClient::IManageable_var mo = kc->getElementByString(keyString);
PolicyHolder_var thePolicyHolder = PolicyHolder::_narrow(mo);
```

To remove an element from the collection, the removeElementByString() method is used:

```
kc->removeElementByString(keyString);
```

Iterating through the elements in a Keyed Reference Collection is done in exactly the same way as for non-keyed collections. In fact, if the collection is referenced using the IManagedCollection::ICommonCollection base interface, as shown in the following code segment, the same code can be used to iterate over elements of either keyed or non-keyed collections. Note that if the variable *theCollection* were of type ICollectionsBase::IMIterable_var, the same code segment could be used to iterate over an iterable home.

```
// Get a keyed or non-keyed collection from somewhere
IManagedCollections::ICommonCollection_var theCollection;

IManagedCollections::IIterator_var theIterator =
    theCollection->createIterator();

IManagedClient::IManageable_var theBO;
while ( theBO = theIterator->next() )
{
   if ( theBO->is_a("PolicyHolder") )
       // Send him a bill
   if ( theBO->is_a("Beneficiary") )
       // Send him a check
}

try
   {
      //Loop through the collection. The "nextElement" method advances
      //the iterator to the next element (on the first invocation, this
      //will advance to the first element) and then return the element
      //pointed to by the iterator.

      while ( theIterator->more() )
       {
          theBO = theIterator->next();
          if ( theBO->is_a("PolicyHolder") )
          // Send him a bill
          if ( theBO->is_a("Beneficiary") )
          // Send him a check
       }
       catch (...)

       theIterator->remove();
```

Each element in the keyed or non-keyed collection is accessed only once and in no defined order.

A number of other methods are available on the IKeyedReferenceCollection interface:

containsKeyString() Determines if an element with a given key exists.

getElementKeyString() Determines the key of a given element.

replaceElementWithKeyString() Replaces an element (that is associated with a specified key) with another object.

With these and the other Keyed Reference Collection methods, an arbitrary set of keyed references to Component Broker managed objects can be maintained easily.

Conventions and Guidelines

This section provides additional information on what is happening on the server when client programs are invoking methods on the business objects. This section does not describe new interfaces but provides additional technical details on what actually occurs. Some of these topics are guidelines or coding techniques that can be useful when interacting with the Component Broker server.

The Component Broker server plays a unique role among application servers: it serves objects. A database server, on the other hand, serves data and a file server serves files. However, the Component Broker server also interacts with database or file servers in cases where these are the chosen options for persistence. In a database server, interaction with the data is direct and the semantics of creating, deleting, and finding are straightforward because you use the methods to the resource manager directly. In the case of an object server, from a programming perspective you encapsulate interaction with a database to get persistence. However, to provide this encapsulation, the decisions about how the database is used are unknown to the clients. Sometimes expectations are not met.

The following sections discuss the interaction patterns that the Component Broker has with various resource managers and the coding patterns that you can use to meet various client requirements.

Finding Persistent Objects

When you want to find an object using the findByPrimaryKeyString() method on the IHome, Component Broker follows a specific algorithm. This algorithm is:

1. Convert parameter into internal key format.
2. Look in the container cache of the active objects.
3. If not found, go to the database or database cache and look.
4. Return the object reference as soon as it is found.

The programming implications of this are:

- If the object exists, the object reference returned is valid and is ready to use.
- If the object was being cached by the container or the cache manager and had subsequently been deleted by a non-object program, an exception is thrown when the object reference is used.
- The probability of getting an exception when using the returned reference is directly proportional to the amount of deleting that is done by existing non-Component Broker applications that are running concurrently with Component Broker applications.

Creating Persistent Objects

When you want to create an object using the createFromPrimaryKeyString() or other create methods, Component Broker follows a specific algorithm with respect to the creation.

1. Create the managed object and its parts.
2. Put it in the container's cache of active objects.
3. Return the object reference to the client.
4. Wait for transaction commit().

5. When the transaction commits, insert the row in the table.

The implications of this algorithm for you are as follows. If the row already exists in the underlying database, an exception is thrown when the transaction completes.

In many cases the exception is desired and should be planned for accordingly. To alter this behavior for create, you can either:

- Do a findByPrimaryKeyString() before issuing the createFromPrimaryKeyString(). Remember from the previous discussion that findByPrimaryKeyString() looks in the database if necessary to find objects and in this instance would return `notFound`. If there is a slim chance that the object already exists, the performance penalty for the findByPrimaryKeyString() might be too high. Select a technique based on the needs of the application.
- Bracket createFromPrimaryKeyString() in a transaction by itself. This ensures that the exception for alreadyExists is thrown before operations on the newly-created object are started.

These techniques do not reduce the amount of written code but are alternatives for structuring client code.

The create_object() Method

The following create_object() method is platform-dependent and does NOT apply to OS/390 Component Broker.

You can use the create_object() method to replicate the function of the createFromPrimaryKeyString() and createFromCopyString() method. The reason for using the create_object() method is to be OMG compliant.

The create_object() method takes two parameters:

key Consists of:

  kind The object interface for BOIM Homes.

  id The name of the business object interface, such as Policy. It is the same value with which the Home is configured for managed object class (interface name) for the particular managed object image in question.

criteria A name-value pair. The name parameter tells the Home what kind of creation is to take place.

BOIM Homes support two names:

primary key string
A signal to use the contents of the value to perform a createFromPrimaryKeyString() using this data.

copy string
A signal to use the contents of the value to perform a createFromPrimaryCopyString() using this data.

An example follows:

```
::CosLifeCycle::Key k;
::CosLifeCycle::Criteria crit;

k.length(1);// set the length of the key to 1

char* idString=new char[7];// allocate storage for the id string
strcpy(idString, "Policy");// set it to the object class interface name
char* kindString=new char[17];// allocate storage for the kind string
strcpy(kindString,"object interface");//set it to object interface
k[0].id=idString;// assign the id of the key
k[0].kind=kindString;// assign the kind of the key

crit.length(1);// set the size of the critieria to 1
char* critname = new char[19];// allocate space for criteria name
strcpy(critname,"primary key string");

// Do a createFromPrimaryKeyString using create_object
crit[0].name=critname;//assign the name

PolicyKey_var theKey = PolicyKey::_create();// build a key
theKey->policyNo(12345);// fill in needed information
ByteString* theKeyString = theKey->toString();//create a bytestring
crit[0].value<<=(*theKeyString);

// Assume that you already found the policy home in "myHome"
try
{
   ::CORBA::Object_var policy = myHome->create_object(k,crit);
   // call create_object
   // notice that a CORBA::Object is returned
}
catch(CosLifeCycle::NoFactory)
{
   // The key passed in does not match that of the home. Check
   // that the kind is "object interface" and the id is the same
   // as that of the managed object interface in the MO Image.

   // Insert recovery code goes here.

}
catch(CosLifeCycle::InvalidCriteria &ic)
{
   // The criteria length was 0.

   // Insert recovery code goes here.

}
catch(CosLifeCycle::CannotMeetCriteria &cmc)
{
   // Recovery:
   // 1) You may have sent in two name-value pairs that could both
   //    create objects, but the home can only create one object at a time.
   // 2) The ByteString may have failed to internalize into the Key
   //    for this type of object.
   // 3) A duplicate or invalid key error may have occurred.

   // In all cases, check the error log.
}
```

```
  catch(CORBA::Exception)
  {
     // The home may not have been configured.
     // An unknown error may have occurred.

     // Insert recovery code goes here.

  }
```

To perform a createFromCopyString, the code is similar:

```
⋮
  PolicyCopy_var theCopy = PolicyCopy::_create();
                              /*  builds a key */
  // Fill in the needed information.
  theCopy->policyNo(12345);
  // Fill in the other values.
  theCopy->amount(10000.00);
  theCopy->premium(250.00);
  ByteString *theCopyString = theCopy->toString();
                              /* creates a bytestring */
  crit[0].value<<=(*theCopyString);
  CORBA::Object_var myObject = create_object(k,crit);

⋮
```

Using Handles

Component Broker supports a notion called Object Handles. A given object in the distributed system can be accessed by using an Object Handle using one of a variety of techniques. There are three main access patterns and there is handle class for each of them along with a base handle class present a consistent interface for accessing the object. It provides streaming support so handle strings can be stored persistently. Given a handle string user can locate/activate the object no matter which type of handle was used for the object.

Each of the access patterns have their merits and limitations. The Component Broker user can decide which pattern should be used for their circumstances.

SORHandle
This pattern encapsulates a stringified object reference (SOR). This pattern requires the most storage space, however it does provide the most efficient access. This pattern should be used only for objects which will never be relocated to other servers.

Note: In Component Broker for Windows NT and AIX, when SORHandle is used with objects that are not Workload Managed (WLM), the object is scoped to a specific server. When used with objects that are Workload Managed, the object can be moved to different servers in the same Server Group.

HomeKeyHandle
This pattern uses the home name and primary key of object. This pattern requires lesser amounts of storage but is slightly less efficient access due to additional path length required to find the home. However, this pattern provides a more flexible solution since homes, and therefore the objects that they manage, can be moved to other servers and hosts. This pattern should be used for all objects that may be required to be moved to other servers or hosts in the future.

Name Handle

This pattern is similar to the HomeKeyHandle pattern except it is explicitly registered in the name space and can therefore be assigned a well-known name to the user. This pattern should be used for well known objects only.

Chapter 7. MOFW Server Programming Model – Advanced Concepts

The Component Broker server programming model is based on programming by framework completion. Component Broker introduces its APIs as sets of classes and frameworks. Developers implement their business functions by defining business objects that subclass from Component Broker frameworks and use the frameworks in their implementation.

This chapter provides business object builders with additional options for providing function in business objects. Some of the material in this chapter offers alternative ways of accomplishing some of the tasks described in Chapter 5, “MOFW Server Programming Model” on page 57.

Some of the topics in this chapter assume the presence of specific application adaptors and other features within Component Broker. In other words, some of the techniques described might increase the cost of porting or targeting business objects and associated applications to other back-end databases and resource managers.

Extending a Business Object

Chapter 5, “MOFW Server Programming Model” on page 57 presents a basic model for building business objects. Most of the discussion and examples in that chapter center around implementing a new business object interface. Careful examination of the examples shows that there are several meaningful layers of inheritance in the business objects that were presented. Often multiple levels of domain inheritance exist and need to be implemented.

This chapter addresses the addition of a subclass to existing business objects using data object inheritance as the implementation technique. This is the model supported by Component Broker through Object Builder. This is not as simple as adding a single class. This chapter revisits each of the steps used to develop a business object in the context of adding another subclass of domain functionality. These steps are:

1. Developing an interface to the business object.
2. Choosing an inheritance pattern.
3. Implementing the business object methods.
4. Implementing the methods required by the MOFW interfaces.
5. Implementing the key classes.
6. Implementing the copy helper classes.

When you create a child component (that is, a component that inherits behavior or data from another component in your application), the child component objects generally inherit from their equivalent parent objects:

- The child business object file must include the parent business object file.
- The child business object interface must inherit from the parent interface.
- The child key and copy helper can inherit from their equivalents in the parent component, or they can contain selected attributes of the parent interface, without inheriting from the parent key or copy helper.
- The child business object implementation must inherit from the parent implementation.
- The child data object interface must inherit from the parent data object interface.
- The child data object implementation must inherit from the parent data object implementation.

- The child managed object must inherit from the parent managed object.

For data inheritance to work, the type of persistence provided by the parent and child data object implementations must be the same.

Many variations involve extending a business object. The next sections give examples of an extreme case where it inherits as much interface and implementation as possible. There are variations that involve less implementation inheritance that can be extrapolated from the examples given.

Extending Business Object Interfaces

v

The first step in building a business object is to construct the interface. In this case, the interface is extended. In the following example, a CarPolicy class extends the Policy.

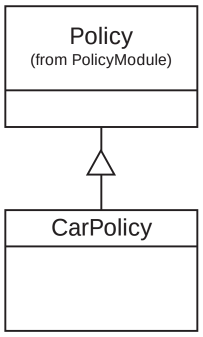


Figure 33. Inheritance of Interface for Extended Business Object

The IDL for CarPolicy should look like the following example.

```
#include <Policy.idl>
interface CarPolicy : Policy
{
  attribute long year;
  attribute string make;
  attribute string model;
  attribute long serialNumber;
  attribute float collisionDeductible;
  attribute boolean glassCoverage;
  long riskQuotient( );
};
```

The interface looks like almost any other business object interface. However, because the Policy already inherits from IManagedClient::IManageable, the CarPolicy interface does not need to.

Essential State Extensions

Extending the essential state is similar to extending the interface. The data object interface of Policy is extended as shown in the following figure.

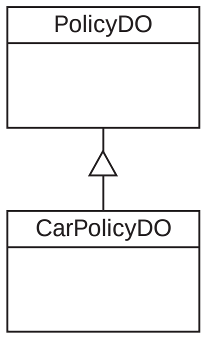


Figure 34. Extending the Interface for Essential State

The IDL for this interface looks like the following example.

```
#include <PolicyDO.idl>
interface CarPolicyDO : PolicyDO
{
   attribute long year;
   attribute string make;
   attribute string model;
   attribute long serialNumber;
   attribute float collisionDeductible;
   attribute boolean glassCoverage;
   #pragma meta CarPolicyDO localonly ,abstract
};
```

“Data Object Customization and Inheritance” on page 233 shows the decisions that must be made when customizing data objects from an implementation perspective.

Choosing an Inheritance Pattern

The choice of inheritance pattern is based on three concerns:

- Identity: whether parent and child have the same identity (that is, they share the same key)
- Performance tradeoffs: whether performance or space efficiency is more important.
- Form of persistence: whether the parent has data to be persisted, and where and how the parent's and child's data is persisted.

If the parent and child have different keys, you should probably use the *Attributes Duplication Pattern*. This means that the child's datastore provides persistence for all of its data, including inherited data. The parent's datastore only provides persistence for instances of the parent, never for instances of the child. If you do not use the overriding persistence pattern, the parent's datastore will have two primary keys: the parent's key for the parent's data, and the child's key for the child's inherited data. It then becomes problematic to determine which data belongs to which object type.

If the parent and child have the same key, you can choose between the *Key Duplication Pattern* and the *Single Datastore Pattern*. The Key Duplication Pattern will generally be more efficient in its use of space (because the persistent objects for each component contain only the data required for that component), and the Single Datastore Pattern will generally provide faster look-up time (because both local and inherited data are mapped to the same persistent object and underlying datastore).

If the parent and child are both persisted in a database, you can compromise between the Key Duplication Pattern and the Single Datastore Pattern, by using the *Single Datastore with Views Pattern*. This pattern uses unique persistent objects for retrieval (the Key Duplication Pattern), and a shared persistent object for all other uses (the shared persistence pattern). This pattern is based on views of the underlying database table, and requires that there be some unique attribute of the child that can be used to select appropriate views of the database.

This example will use the Key Duplication Pattern for illustrations purposes. This is the default pattern supported by Object Builder.

See the “Inheritance” section in the*Component Broker Application Development Tools* for further information on the inheritance patterns.

Implement the Additional Business Logic

Next, you must add the implementation of the business logic for the additional methods necessary in the subclass. Introducing another interface for the CarPolicyBO is shown in the following figure and in the following example.

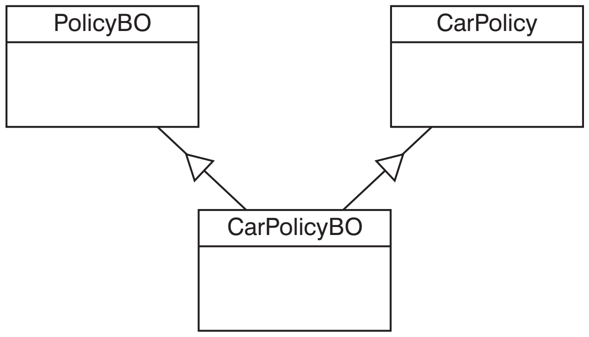


Figure 35. Extending the Business Logic Interface

```
#include <CarPolicy.idl>
#include <PolicyBO.idl>

interface CarPolicyBO : CarPolicy, PolicyBO
{
};
```

Consider using the same pattern (delegating versus caching) in the subclass as was used in the base class.

The actual implementation interface has the inheritance of the PolicyBO built right into it, as shown in the following example.

```
class CarPolicyBO_Impl : public virtual ::CarPolicyBO_Skeleton
,public virtual PolicyBO_Impl
{
   public:

   CarPolicyBO_Impl();

   ::CORBA::Long year();
   ::CORBA::Void year( ::CORBA::Long year);
   char* make();
   ::CORBA::Void make(const char* make);
   char* model();
   ::CORBA::Void model(const char* model);
   ::CORBA::Long serialNumber();
   ::CORBA::Void serialNumber( ::CORBA::Long serialNumber);
   ::CORBA::Float collisionDeductible();
   ::CORBA::Void collisionDeductible( ::CORBA::Float collisionDeductible);
   ::CORBA::Boolean glassCoverage();
   ::CORBA::Void glassCoverage( ::CORBA::Boolean glassCoverage);
   virtual ::CORBA::Long riskQuotient();

   virtual ::CORBA::Void initForCreation(::IManagedServer::IDataObject_ptr theDO);
   virtual ::CORBA::Void uninitForDestruction();
   virtual ::CORBA::Void initForReactivation(::IManagedServer::IDataObject_ptr theDO);
   virtual ::CORBA::Void uninitForPassivation();
   virtual ::CORBA::Void syncToDataObject();
   virtual ::CORBA::Void syncFromDataObject();
   virtual ::CORBA::Void externalize_to_stream(::CosStream::StreamIO_ptr targetStreamIO);
   virtual ::CORBA::Void internalize_from_stream(::CosStream::StreamIO_ptr
   sourceStreamIO,::CosLifeCycle::FactoryFinder_ptr there);
   virtual ::ByteString* getPrimaryKeyString();

   protected:

   private:

   CarPolicyDO* iDataObject;
   ::CORBA::Void initializeState();
};
```

This section is focused on business logic. The entire implementation interface is shown in the previous example. How the non-business logic or MOFW framework methods are to be handled is shown in upcoming sections.

The getters and the setters are implemented as they would be in any business object. The only difference in the other methods is that they can choose to access or use state data from the parent class. The following method implementation shows utilization of state data from both the Policy and CarPolicy class.

```
::CORBA::Long CarPolicyBO_Impl::riskQuotient ()
{
   if ( iGlassCoverage() )
   {
      if ( iYear() > 1960)
      return 1;
      else if ( policyNo() < 1000)
      return 2;
      else
      return 5;
   }
   else
   if ( amount() > 1000 )
   return 10;
   else
   return 100;
}
```

In the previous code segment, the accessor or getter methods are used to access the state data that is needed from Policy. This is the most encapsulated way of doing this. However, if the members are declared as protected instead of private in the PolicyBO_Impl class, then direct access would be possible.

The previous example shows the caching data object case. In the delegating case, the same CarPolicyDO_ptr would be used to access all of the state data regardless of whether or not it resides in the CarPolicy or the Policy.

Meet the MOFW IManageable Requirements

The following methods are required to be overridden in the simple case:

- getPrimaryKeyString
- getHandleString
- externalize_to_stream
- internalize_from_stream

Note: getHandleString() is not actually required to be overridden in the simple case, unless this object is going to be the target of a one-to-one relationship from another object.

The next section discusses considerations for the inheritance case.

getPrimaryKeyString

This method implementation in the inheritance case depends on the decision made about the key to be used for the new subclass. That is, what will the key be for the CarPolicy? If a new key class is introduced, then this method must be overridden. If the existing key class can be used, that is, use the PolicyKey for the CarPolicy class, then this method does not need to be overridden and can be inherited directly.

See “More Key Classes” on page 113 for help in determining if another key class is needed.

getHandleString

A default implementation of this method is provided by the managed object framework. Overriding this method in a subclass of a business object should be done based on the same criteria that are used to determine if an override is needed in the base class.

externalize_to_stream

This method needs to be implemented. The general direction is to call the parent class method and add those things which are necessary for the subclass.

```
::CORBA::Void CarPolicyBO_Impl::externalize_to_stream(
   ::CosStream::StreamIO_ptr targetStreamIO )
{
   // Insert Method modifications here
   PolicyBO_Impl::externalize_to_stream(targetStreamIO);

   targetStreamIO->write_long(iDataObject->year());
   targetStreamIO->write_string(iDataObject->make());
   targetStreamIO->write_string(iDataObject->model());
   targetStreamIO->write_long(iDataObject->serialNumber());
   targetStreamIO->write_float(iDataObject->collisionDeductible());
   targetStreamIO->write_boolean(iDataObject->glassCoverage());

   // End Method modifications here
}
```

While the previous example is for a caching business object, the pattern for a delegating data object is similar.

internalize_from_stream

This method needs to be implemented. The general direction is to call the parent class method and then add those things which are necessary for the subclass.

```
::CORBA::Void CarPolicyBO_Impl::internalize_from_stream(
   ::CosStream::StreamIO_ptr sourceStreamIO,
   ::CosLifeCycle::FactoryFinder_ptr there)
{
  // Insert Method modifications here
  PolicyBO_Impl::internalize_from_stream(sourceStreamIO, there);

  iDataObject->year(sourceStreamIO->read_long());
  iDataObject->make(sourceStreamIO->read_string());
  iDataObject->model(sourceStreamIO->read_string());
  iDataObject->serialNumber(sourceStreamIO->read_long());
  iDataObject->collisionDeductible(sourceStreamIO->read_float());
  iDataObject->glassCoverage(sourceStreamIO->read_boolean());

  // End Method modifications here
}
```

While the previous example is for a caching business object, the pattern for a delegating data object is similar.

MOFW Requirements – IManagedServer

The following methods must be overridden in the simple case:

- initForCreation
- uninitForDestruction
- initForReactivation
- uninitForPassivation
- syncFromDataObject
- syncToDataObject

The next section discusses considerations for the inheritance case.

initForCreation

Code this method following the same guidelines that were specified for building business objects independent of this inheritance case. If the subclass introduces additional state data, as the CarPolicy example does, then the data object must be set into a data member of the object.

If the subclass does not introduce additional state data, it does not need to save a pointer to the data object that is passed as a parameter. In the delegating case, the subclass might still want to hold a pointer to the data object rather than going through parent class get and set methods. However, regardless of whether the subclass introduces additional state or not, the parent class' initForCreation() method must be called at the beginning of the subclass' initForCreation() method.

```
::CORBA::Void CarPolicyBO_Impl::initForCreation(::IManagedServer::IDataObject_ptr theDO)
{
   // Insert Method modifications here
   PolicyBO_Impl::initForCreation(theDO);

   iDataObject = CarPolicyDO::_narrow(theDO);

   // End Method modifications here
}
```

uninitForDestruction

Implement this method following the guidelines described previously. It is also a good practice to call the parent class' uninitForDestruction() at the beginning of the subclass' uninitForDestruction() method if needed.

initForReactivation

Code this method following the same guidelines that were specified for building business objects independent of this inheritance case. If the subclass introduces additional state data, as the CarPolicy example does, then the data object must be set into a data member of the object.

If the subclass does not introduce additional state data, it does not need to save a pointer to the data object which is passed as a parameter. In the delegating case, the subclass might still want to hold a pointer to the data object rather than going through parent class get and set methods. However, regardless of whether the subclass introduces additional state or not, the parent class' initForReactivation() method must be called at the beginning of the subclass' initForReactivation() method.

```
::CORBA::Void CarPolicyBO_Impl::initForReactivation(::IManagedServer::IDataObject_ptr theDO)
{
   // Insert Method modifications here
   PolicyBO_Impl::initForReactivation(theDO);

   iDataObject = CarPolicyDO::_narrow(theDO);

   // End Method modifications here
}
```

uninitForPassivation

This method should be implemented following the normal guidelines described earlier. It is also good practice to call the parent class' uninitForPassivation() at the beginning of the subclass' uninitForPassivation() method if needed.

syncFromDataObject

The pattern for implementing this method is similar to that used for the internalize_from_stream() method. The parent method must be called first, followed by the code necessary to prime the subclassing business object cache with values from the data object.

This method is also a good place for any initialization logic that is dependent on the presence of an active and usable data object. The act of loading up the business object cache ensures that the state data of the object is ready to be used.

syncToDataObject

The pattern for implementing this method is similar to that used for the externalize_to_stream() method. The parent method must be called first, followed by the code necessary to push the subclassing business object cache values back into the data object.

It is a good practice to always implement all of these IManagedServer methods even if they are not explicitly required based on the particular subclass being introduced. This practice results in consistently generated code which is less prone to error if future changes are made to the subclass that introduces new attributes.

More Key Classes

The MOFW requires that every managed object have a primary key class associated with it. When extending a business object, there are several possibilities you can use as a primary key class. The business object subclass can:

- Use its base class' primary key class.
- Extend its base class' primary key class.
- Introduce its own primary key class.

The simplest approach is to reuse the existing key. This approach is applicable if the attributes that uniquely identify objects of the subclass are the same ones that identify objects of the base class. This is most likely the case if the subclass introduces no additional state (that is, the subclass only re-implements the base class methods, or introduces new methods). Even if the subclass introduces an additional state beyond that of the base class, this additional state might not contribute anything to object identity.

If the subclass introduces additional state, some of which, combined with the key attributes of the base class, is used to uniquely identify objects of the subclass, then the best approach is to extend the base class' primary key class. When extending the primary key class of a base class, interface and implementation inheritance can be used. Implementation inheritance allows for the reuse of the base class key functionality (specifically, its getters and setters, as well as its streaming code).

Finally, if the attributes that uniquely identify objects of the subclass are neither the same ones as the base class, nor a superset of the base class' key attributes, then the subclass must introduce its own primary key class.

However, if the existence of new state data has altered the way in which the object is uniquely identified, then a new primary key class is necessary.

Note: If the subclass does not use the base class' primary key class, then the subclass' data object needs to be able to handle this extended or new key class.

More Copy Helper Classes

If the subclassing business object introduces additional state data, then a new copy helper class might be useful. The data object needs to be able to handle this new copy helper. The copy helper can inherit interface and implementation from the base class' copy helper, or it can be written from scratch.

Extension Summary

In the managed object framework based programming model, there are a number of inheritance activities to follow. Interfaces should be inherited consistently. Implementations should be inherited when the base class' implementation can be reused to some degree. The simplest model is to inherit at all levels of the MOFW architecture and add in additional business logic as necessary. Additional requirements from the MOFW are also added in the new implementation subclass.

Managed object customization and data object customization are also different for business objects that inherit from other business objects. These topics are discussed in "Data Object Customization and Inheritance" on page 233.

Other Variations to Consider

There are other variations to consider. Some are restrictions and others are tips for leveraging inheritance in the MOFW-based programming environment:

- Do not change data object patterns. It is possible to change data object patterns from caching to delegating at various levels of the data object hierarchy but this adds undue complexity in most cases.
- There might be cases when the subclasser does not know the data object pattern being used by the base class.

Object Relationships

Component Broker applications often require persistent relationships between business objects. For instance, in the personal life insurance sample application, a Claim object has a relationship with a Policy object representing the insurance policy against which the claim has been filed. Also, a Policy object has a relationship with Claim objects for pending claims against the insurance policy.

Relationships between objects can be described in many ways. First, there is the cardinality of the relationship. If an object has a relationship to one other object at most, the relationship is considered to be cardinality-1. On the other hand, if an object has a relationship with more than one other object at a time, the relationship is considered to be cardinality-N. In a business object, relationships to other objects are implemented as object references (cardinality-1), or as collections of object references (cardinality-N).

A relationship can also be described as *optional* or *required*. If a relationship is optional, then an object is considered to be in a valid state even when it is not related to (linked to) another object. If the relationship is required, then the object must always be linked to another object. This distinction is often combined with cardinality to form the following combinations:

Class Relationship	**Cardinality**	**Required**
An instance of class X is related to 0..1 instances of class Y	-1	No
An instance of class X is related to 1 instance of class Y	-1	Yes
An instance of class X is related to 0..n instances of class Y	-N	No
An instance of class X is related to 1..n instances of class Y	-N	Yes

Finally, a relationship can also be described in terms of ownership. An object which is related to another object might or might not be considered to be the owner of the related object. If the first object is not considered to be the owner of the second object, then the relationship is often referred to as a *uses a* relationship, as in, for example, the first object *uses* the second object. This is sometimes also called an association. On the other hand, if the first object is considered to be the owner of the second object, then the relationship is often referred to as a *has a* relationship, as in, for example, the first object *has* (or contains) the second object. This is sometimes also called an aggregation.

The cardinality-1 or cardinality-N distinction results in much more fundamental differences in the business object code than the optional or required distinction and the uses or has distinction. Cardinality-1 relationships are discussed separately from cardinality-N relationships.

Cardinality-1 Relationships

The following diagram shows an example of a cardinality-1 relationship, a simple link from a Claim object to a Policy object.

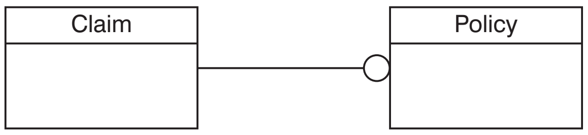


Figure 36. Cardinality-1 Relationship

The relationship depicted in the previous figure is that of an optional cardinality-1 *uses a* relationship. In a business object interface, a cardinality-1 relationship is declared as a CORBA attribute whose type is a reference to the interface of the target object. For example, an attribute called *thePolicy* on a Claim business object could be used to link to the Policy object as follows:

```
interface Claim : ...
{
   attribute Policy thePolicy;
   ...
};
```

Clients can establish and traverse the link using the attribute set and get methods respectively:

```
Policy_var aPolicy = ... // Find or create a Policy object
Claim_var aClaim = ... // Find or create a Claim object

// Set the claim's policy to "aPolicy".
aClaim->thePolicy(aPolicy);
// Get the claim's policy as "somePolicy".
Policy_var somePolicy = aClaim->thePolicy();
```

Using the Component Broker delegating pattern, the business object implementation of the access methods for thePolicy passes the object pointer to or from the data object:

```
::CORBA::Void ClaimBO_Impl::thePolicy(Policy_ptr policy)
{
   fDataObject->thePolicy(policy);
}

Policy_ptr ClaimBO_Impl::thePolicy()
{
   return fDataObject->thePolicy();
}
```

Because the relationship is optional it is possible that the data object will return either a valid pointer to a policy object or a null pointer.

To speed up the performance of link access, caching the Policy pointer might be appropriate. The Claim business object implementation class could cache a pointer to the Policy object as shown in the following example:

```
class ClaimBO_Impl ...
{
   public:
      ...
   private:
      ...
      Policy_var fCachedPolicy;
      ...
}
```

The access methods for thePolicy would now use the cached pointer:

```
::CORBA::Void ClaimBO_Impl::thePolicy(Policy_ptr policy)
{
   fCachedPolicy = Policy::_duplicate( policy );
}

Policy_ptr ClaimBO_Impl::thePolicy()
{
   return Policy::_duplicate( fCachedPolicy );
}
```

In this case it is not necessary to invoke the release() method on the previous Policy object prior to saving the new one. This is because the Policy object is being saved in a Policy_var object. This Policy_var object ensures that the previous Policy object is released when it is no longer needed, including when it is being assigned a new Policy object.

The general Component Broker business object caching pattern uses the methods syncFromDataObject and syncToDataObject respectively to load and flush the cached values. See the description of “syncFromDataObject() Method” on page 72 and “syncToDataObject()” on page 73. The implementation of the syncFromDataObject method calls data object get methods to retrieve the thePolicy pointer as well as all other data contained in the Claim object. The implementation of the syncToDataObject method calls all the data object set methods. These methods appear as follows:

```
::CORBA::Void ClaimBO_Impl::syncToDataObject()
{
   ...
   fDataObject->thePolicy( fCachedPolicy);
   ...
}

::CORBA::Void ClaimBO_Impl::syncFromDataObject()
{
   ...
   fCachedPolicy = fDataObject->thePolicy();
   ...
}
```

It is not necessary to use the _duplicate() method in the implementations of the syncToDataObject() and syncFromDataObject() methods because the Policy object is neither an input parameter nor a return value of these methods.

Because links can be expensive to compute, and sometimes are not needed, a better pattern for caching object references is to leave them out of the syncTo/FromDataObject methods and instead compute and cache them in the get method on first use. This *lazy evaluation* approach can be implemented using the following pattern:

```
::CORBA::Void ClaimBO_Impl::thePolicy(Policy_ptr policy)
{
   fCachedPolicy = Policy::_duplicate( policy );

   // Synchronize the new Policy in the BO with (to) the DO

   fDataObject->thePolicy(fCachedPolicy);
}

Policy_ptr ClaimBO_Impl::thePolicy()
{
   // If this is the first access of the Policy, get it from the DO.

   if ( CORBA::is_nil(fCachedPolicy) )
      fCachedPolicy = fDataObject->thePolicy();

   return Policy::_duplicate( fCachedPolicy );
}
```

Assuming fCachedPolicy is initialized to nil in the constructor, this pattern results in the data object get method being called the first time the Policy is accessed, or not at all if the Policy is set in the same session as the get call. Subsequent accesses return the cached pointer value.

Optional or Required Cardinality-1 Relationships

The previous section discusses how an optional relationship is implemented in a business object. An optional relationship is more flexible than a required relationship, and thus requires less code. However, if the relationship is required, the following diagram represents a required cardinality-1 *uses a* relationship.

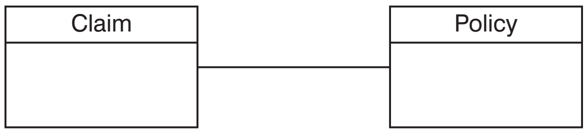


Figure 37. Required Cardinality-1 "Uses a" Relationship

If the relationship is truly required, then the link from a Claim to its Policy must exist throughout the life cycle of the Claim. Specifically, a Policy must be linked to a Claim as part of creating a Claim, and the link must be broken when a Claim is removed. The Policy that must be linked to a Claim as part of creation might exist prior to creating the Claim, or it might be created in the process of creating the Claim.

When reviewing a Claim, the link to its Policy must be broken, but the Policy is not necessarily removed. If the claim *uses a* (knows about a) Policy, then it is sufficient to release the Policy when a Claim is removed. However, if the Claim to Policy relationship is one of ownership (that is, a *has a* relationship), then when a Claim is removed, its associated Policy must also be removed.

When a Claim is removed, the link to its Policy must be broken. If the Claim is implemented using the caching pattern, then the link is automatically broken if it is being cached in a Policy_var. Otherwise, it must be explicitly broken using CORBA::release(). Changes between an optional and a required relationship depend on one of four different scenarios (discussed over the next several pages). However, there is one change which is common to all four. Under any scenario, if the relationship is required instead of optional, the code for the set method for the *Policy* attribute must make sure that the link is not broken by setting the Policy reference to nil. The following example shows the changes necessary in the case of the delegating pattern. The changes for the caching pattern are analogous.

```
::CORBA::Void ClaimBO_Impl::thePolicy(Policy_ptr policy)
{
   if ( ! CORBA::is_nil(policy) )
   {
      fDataObject->thePolicy(policy);
   }
}
```

To understand how a link from a Claim to its Policy gets established as part of creating a Claim, it is necessary to recall that there are several ways to create a business object:

- Using a generic home configured for the appropriate type of business object. See “Creating a Claim With an Existing Policy Using a Generic Home” and “Creating a Claim With a New Policy Using a Generic Home” on page 120.
- Using a specialized home developed by the business object provider. See “Creating a Claim With an Existing Policy Using a Specialized Home” on page 122 and “Creating a Claim With a New Policy Using a Specialized Home” on page 123.

Creating a Claim With an Existing Policy Using a Generic Home: Using a generic home, a business object is created using the createFromPrimaryKeyString() method. If an object provider has provided a copy helper class, then a business object can also be created using the createFromCopyString() method. However, this has no effect on this discussion because a copy helper class' attributes are defined as being a superset of the primary key class' attributes. The only input to this method is a stringified version of the business object's primary key helper object. This means that the primary key helper class for Claim must contain enough information in it such that the link to its (preexisting) Policy can be established. There are many ways in which this can be done, but the two most straightforward ways are the following:

- The primary key for Claim contains a reference to the Policy object. See “Primary Key Contains Reference to Related Object.”
- The primary key for Claim contains all of the Policy object's primary key attributes. See “Primary Key Contains Key Attributes of Related Object” on page 119.

Primary Key Contains Reference to Related Object: In this case, the interface of the primary key for Claim would look like the following example:

```
interface ClaimKey : IManagedLocal::IPrimaryKey
{
   attribute long claimNo; // key attribute for Claim

   attribute Policy thePolicy; // associated Policy
}
```

The implementation binding header file (.ih) would include the following private or protected data members:

```
::CORBA::Long        fClaimNo;
Policy_var           fPolicy;
```

The attribute for the associated Policy object would be implemented as follows:

```
::CORBA::Void ClaimKey_Impl::thePolicy(Policy *thePolicy)
{
   fPolicy = Policy::_duplicate( thePolicy );
}
```

```
Policy ClaimKey_Impl::thePolicy()
{
   return Policy::_duplicate( fPolicy );
}
```

For a primary key to flow over the wire from the client to the server, the primary key must be able to externalize and internalize all of its attributes, including those which are necessary for establishing relationships to other objects. In the case of the primary key class for the Claim object, it must externalize and internalize a stringified object reference to the Policy object as follows:

```
::CORBA::Void ClaimKey_Impl::externalize_to_stream
   (::CosStream::StreamIO_ptr targetStreamIO)
{
   // Insert Method modifications here
   targetStreamIO->write_long(fClaimNo);
   CORBA::String_var policyRefString= CBSeriesGlobal::orb()-> object_to_string(fPolicy);
   targetStreamIO->write_string(policyRefString);

   // End Method modifications here
}

::CORBA::Void ClaimKey_Impl::internalize_from_stream
   (::CosStream::StreamIO_ptr sourceStreamIO,
    ::CosLifeCycle::FactoryFinder_ptr there)
{
   // Insert Method modifications here
   fClaimNo = sourceStreamIO->read_long();
   CORBA::String_var policyRefString = sourceStreamIO->read_string();
   CORBA::Object_var policyRef = CBSeriesGlobal::orb()-> string_to_object(policyRefString);
   fPolicy = Policy::_narrow(policyRef);
   targetStreamIO->write_string(policyRefString);

   // End Method modifications here
}
```

The streaming code in the previous example allows the primary key for Claim to flow from the client application to the Claim home on the server. When the primary key reaches the Claim home on the server, it eventually gets passed to the data object implementation for Claim. The link has been established by invoking the string_to_object() method on the ORB while internalizing the Claim primary key from its stream.

Primary Key Contains Key Attributes of Related Object: In the second case, the interface of the primary key for Claim would look like the following example:

```
interface ClaimKey : IManagedLocal::IPrimaryKey
{
   attribute long claimNo; // key attribute for Claim

   attribute long policyNo; // key attribute for Policy
}
```

The implementation binding header file (.ih) would include the following private or protected data members:

```
::CORBA::Long        fClaimNo;
::CORBA::Long        fPolicyNo;
```

The attribute for the associated Policy object would be implemented as follows:

```
::CORBA::Void ClaimKey_Impl::policyNo(::CORBA::Long policyNo)
{
   fPolicyNo = policyNo;
}

::CORBA::Long ClaimKey_Impl::policyNo()
{
   return fPolicyNo;
}
```

Again, in order for a primary key to go from the client to the server, the primary key must be able to externalize and internalize all of its attributes, including those which are necessary for establishing relationships to other objects. In this case, the primary key class for the Claim object must externalize and internalize the key attributes of its related Policy object as follows:

```
::CORBA::Void ClaimKey_Impl::externalize_to_stream
   (::CosStream::StreamIO_ptr targetStreamIO)
{
   // Insert Method modifications here
   targetStreamIO->write_long(fClaimNo);
   targetStreamIO->write_long(fPolicyNo);

   // End Method modifications here
}

 ::CORBA::Void ClaimKey_Impl::internalize_from_stream
    (::CosStream::StreamIO_ptr sourceStreamIO,
     ::CosLifeCycle::FactoryFinder_ptr there)
{
    // Insert Method modifications here
    fClaimNo = sourceStreamIO->read_long();
    fPolicyNo = sourceStreamIO->read_long();

    // End Method modifications here
}
```

As with the previous scenario, the streaming code in this example allows the primary key for Claim to flow from the client application to the Claim home on the server. However, that is the end of the similarity. The previous scenario shows that the cardinality-1 relationship is established as part of internalizing the Claim primary key inside the Claim home on the server.

In this scenario, because the Claim primary key class contains the key attributes of the related Policy object, the relationship must be established in the Claim data object implementation. The Claim data object implementation would do so by extracting the Policy key attributes from the Claim primary key, finding a home of Policy objects, and invoking the findByPrimaryKeyString() method on the home.

Creating a Claim With a New Policy Using a Generic Home: In the previous scenario, the application domain specified a constraint that a Claim can only be created for an existing Policy. It is possible that the application domain, while requiring that a relationship between two objects exist, does not require one object to already be created when creating the second. While an insurance company which allows a Policy to be created at the same time as a Claim would probably not stay in business long. That scenario is used here for consistency.

Like the previous scenario, this scenario assumes that the Claim object provider has chosen not to develop a specialized home and instead expects clients of the Claim business object to use a generic home. Because the only input to the createFromPrimaryKeyString() method of a generic home is a stringified version of the business object's primary key helper object, this scenario could be implemented in a similar fashion to the previous scenario, where the Claim primary key class contains the key attributes of a Policy. However, the one difference is what happens when the Claim primary key containing information about a Policy reaches the Claim home on the server.

In the previous scenario, where the Claim is being created with an existing Policy, there is no need for the Claim data object implementation to differentiate between when the business object is being created for the first time, and when it is being reactivated after having been previously passivated. In other words, in either case, the Claim data object implementation uses the Policy key attributes from its own key to find the existing Policy object.

In this scenario, however, the Policy is being created during the creation of a Claim. As such, the Claim data object implementation must distinguish between the creation and reactivation of a Claim as follows:

- If a Claim is being created, then a new Policy object must be created.
- If a Claim is being reactivated, then the previously created Policy object must be found.

A data object does not know if the business object is being created or reactivated. However, the business object itself does know if it is being created or reactivated. If the business object is being created the home invokes the initForCreation() method on it; if the business object is being reactivated the home invokes the initForReactivation() method.

In its implementation of the initForCreation() method, the Claim business object would do the following:

- Use a factory finder to find a Policy home
- Use the Policy key attributes from the Claim data object to create a new Policy by invoking the createFromPrimaryKeyString() method on the Policy home.

In its implementation of the initForReactivation() method, the Claim business object would do the following:

- Use a factory finder to find a Policy home.
- Use the Policy key attributes from the Claim data object to find its related Policy by invoking the findByPrimaryKeyString() method on the Policy home.

There is an alternate way to implement this scenario. Assuming that the client of a Claim is not required to provide the identity of its Policy, then the primary key class for Claim does not need to contain any information about the Policy. The primary key class for Claim would look like the following example:

```
interface ClaimKey : IManagedLocal::IPrimaryKey
{
   attribute long claimNo; // key attribute for Claim
}
```

In this case, it is the responsibility of the Claim business object, not the data object, to create the Policy during its own creation. A data object has no way to distinguish between object creation and object reactivation. However, a business object knows it is being created when its home invokes the initForCreation() method on it. Thus, the implementation of ClaimBO_Impl::initForCreation() would need to somehow create a Policy object. Because the Claim business object was not provided with any information on how to identify the Policy object, it would probably do one of the following:

- Create a primary key for Policy based on some combination (or function) of its own attributes.
- Create a primary key for Policy based on some random number generator or UUID (Universal Unique Identifier).
- Use a specialized home for Policy (if one was provided).

Because the relationship of Claim to Policy is required, if the Claim business object is unable to create a Policy for some reason, then it should cause its own creation to fail. The following example shows how this would be done:

```
::CORBA::Void ClaimBO_Impl::initForCreation(
       ::IManagedServer::IDataObject_ptr theDO)
{
   // Save the data object for later use
   fDataObject = ClaimDO::_narrow(theDO);


   try // to create the Claim's Policy somehow
   {
      fCachedPolicy = ...
   }
   catch(...)
   {
      throw IManagedServer::ICreationFailed();
   }


   // Other initialization...
}
```

Creating a Claim With an Existing Policy Using a Specialized Home: Using a specialized home, it is even easier to establish a link from a Claim to its Policy as part of creating the Claim. With a specialized home, there is no need to add any information about the Policy to the primary key for Claim. Adding information about one object to the primary key of another in the previous scenarios was done for the sole purpose of establishing the required relationship. The primary key was being used not just to establish unique identity, but also as a vehicle for communicating information about the relationship from the client to the home. This is not what primary key classes were intended for, but was done out of necessity given that a specialized home was not provided.

Unfortunately, doing so introduced the following unwanted side-effect. Using the generic home configured for Claim objects at some later time to find a previously created Claim object required the client to have some information about the Policy too (at least some of the key attributes, if not a reference to the Policy object itself, depending on the scenario). This might not be an acceptable constraint in the application domain. If not, then the solution is to have the Claim object provider provide a specialized home as well.

Using a specialized home, the link between a Claim and its Policy is established in the specialized home itself, as opposed to in the Claim business object or data object. In order to support the required cardinality-1 relationship to a Policy, the specialized home for Claim would introduce new methods for creating a Claim. These new methods would have parameters which would allow the specialized home to establish the link between a Claim and its Policy. The following shows two examples of such create methods:

```
Claim_ptr ClaimHomeBO_Impl::createClaimWithPolicyRef(
          long claimNo,
          Policy_ptr policy)
{
   // Before we get too far, let's make sure that we have a valid
   // Policy reference for the required cardinality-1 relationship.
   if ( CORBA::is_nil(policy) )
      throw ClaimHome::MissingPolicy();

   ClaimKey_var claimKey = ClaimKey::_create();
   claimKey->claimNo(claimNo);
   ByteString_var claimKeyString = claimKey->toString();

   // For an inheriting specialized home, pass the claimKeyString
   // to the parent of the specialized home on creatFromPrimaryKey-
   // String(). For a delegating specialized home, pass the claim-
   // KeyString to the contained home on createFromPrimaryKeyString().
   Claim_ptr newClaim       = ...

   // Now that the Claim has been created, establish the link to its
   // Policy using the Policy which was passed in on createClaim().
   newClaim->thePolicy(policy);

   return newClaim;
}

Claim_ptr ClaimHomeBO_Impl::createClaimWithPolicyNo(
          long claimNo,
          long policyNo)
{
   // Assume that in the ClaimHomeBO_Impl::initForCreation()
   // the specialized home for Claim objects has used a factory
   // finder to find the home for Policy objects, and saved
   // its reference in fPolicyHome

   PolicyKey_var policyKey = PolicyKey::_create();
   policyKey->policyNo(policyNo);
   ByteString_var policyKeyString = policyKey->toString();

   Policy_var aPolicy; // Assume fPolicyHome is a specialized home...
   aPolicy = fPolicyHome->findByPrimaryKeyString(policyKeyString);

   ClaimKey_var claimKey = ClaimKey::_create();
   claimKey->claimNo(claimNo);
   ByteString_var claimKeyString = claimKey->toString();

   // For an inheritting specialized home, pass the claimKeyString
   // to the parent of the specialized home on creatFromPrimaryKey-
   // String(). For a delegating specialized home, pass the claim-
   // KeyString to the contained home on createFromPrimaryKeyString().
   Claim_ptr newClaim       = ...

   // Now that the Claim has been created, establish the link to its
   // Policy using the Policy which was passed in on createClaim().
   newClaim->thePolicy(policy);

   return newClaim;
}
```

Creating a Claim With a New Policy Using a Specialized Home: This scenario is similar to the previous scenario. However, because in this scenario the Policy is not created prior to creating a Claim, the createClaimWithPolicyRef() method from the previous scenario is not applicable and the implementation of the createClaimWithPolicyNo() method is different. Instead of invoking the findByPrimaryKeyString() method on the Policy home it must invoke the createFromPrimaryKeyString() method as illustrated in the following example:

```
Claim_ptr ClaimHomeBO_Impl::createClaimWithPolicyNo(
          long claimNo,
          long policyNo)
{
   // Assume that in the ClaimHomeBO_Impl::initForCreation()
   // the specialized home for Claim objects has used a factory
   // finder to find the home for Policy objects, and saved
   // its reference in  fPolicyHome'.

   PolicyKey_var policyKey = PolicyKey::_create();
   policyKey->policyNo(policyNo);
   ByteString_var policyKeyString = policyKey->toString();

   Policy_var aPolicy; // Assume fPolicyHome is a specialized home...
   aPolicy = fPolicyHome->createFromPrimaryKeyString(policyKeyString);

   ClaimKey_var claimKey = ClaimKey::_create();
   claimKey->claimNo(claimNo);
   ByteString_var claimKeyString = claimKey->toString();

   // For an inheriting specialized home, pass the claimKeyString
   // to the parent of the specialized home on creatFromPrimaryKey-
   // String(). For a delegating specialized home, pass the claim-
   // KeyString to the contained home on createFromPrimaryKeyString().
   Claim_ptr newClaim       = ...

   // Now that the Claim has been created, establish the link to its
   // Policy using the Policy which was passed in on createClaim().
   newClaim->thePolicy(policy);

   return newClaim;
}
```

"Uses a" and "Has a" Cardinality-1 Relationships

Now that the differences between optional and required cardinality-1 relationships have been described, especially as they pertain to developing a business object, the *uses a* and *has a* cardinality-1 relationships are described.

Previous sections discuss how *uses a* relationships are represented. The following figure illustrates an optional cardinality-1 *has a* relationship.

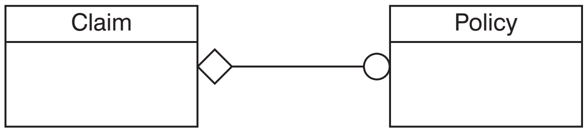


Figure 38. Optional Cardinality-1 "Has a" Relationship

The following diagram, on the other hand, illustrates a required cardinality-1 *has a* relationship.

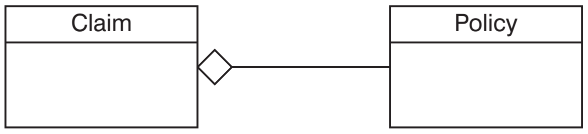


Figure 39. Required Cardinality-1 "Has a" Relationship

A *uses a* relationship means that one object has a reference to another object. In the example, a Claim object has a reference to a Policy object. In fact, there might be more than one Claim object with a reference to the same Policy object and there might also be objects of other types with references to the same Policy object. In any case, any object with a reference to this Policy object is capable of invoking any method on the Policy object's client interface. This includes the remove() method. Of course, things could get chaotic if any object with a *uses a* relationship to the same Policy object were allowed to invoke the remove() method on it. To avoid such confusion, Component Broker recommends that an object be removed only by its owner, and that an object be owned only by a single object. All other objects with references to that object should only release the reference.

A *has a* relationship is what is used to represent the concept of ownership. In other words, an object with a *has a* relationship to another object is said to own that object. In the previous two diagrams, a Claim object *has a* reference to a Policy object. Although the relationship would most likely be reversed in the insurance application domain, this scenario continues with the original example instead of introducing two new objects. Thus, a Claim is considered to be the owner of a Policy and as such is responsible for removing the Policy should this become necessary. One case in which it becomes necessary for a Claim to remove the Policy is when the Claim itself is being removed. A business object knows it is being removed when its home invokes the uninitForDestruction() method on it. The following example shows how the Claim business object would implement this:

```
::CORBA::Void ClaimBO_Impl::uninitForDestruction()
{
   // Remove the (owned) Policy object
   fCachedPolicy->remove();

   // If fCachedPolicy is declared as a Policy_ptr, then it is also
   // necessary to release the reference.
   fCachedPolicy->release();
   // If fCachedPolicy is declared as a Policy_var, then this happens
   // automatically when the BO is destructed.

   // Other un-initialization...
}
```

Another case in which it becomes necessary for a Claim to remove the Policy is if the Claim interface has a method that gives its clients the opportunity to request that the Policy be removed. This is similar to the cancelPolicy() method in the following example:

```
::CORBA::Void ClaimBO_Impl::cancelPolicy()
{
   // Remove the (owned) Policy object
   fCachedPolicy->remove();

   // If fCachedPolicy is declared as a Policy_ptr, then it is also
   // necessary to release the reference.
   fCachedPolicy->release();
   // If fCachedPolicy is declared as a Policy_var, then this happens
   // automatically when the BO is destructed.

   // Other clean-up associated with the Policy cancellation
}
```

Of course, the cancelPolicy() method makes sense only in an optional *has a* relationship.

In any case, if one business object is responsible for the removal of another business object as the result of a *has a* relationship, then the reverse must not also be the case. In other words, in Component Broker, two objects might not have a *has a* relationship with one another. If such a bi-directional relationship is required, then one object must release the other, while the other object removes the first one as described in this section.

Making Cardinality-1 Relationships Persistent

Because object references are really just memory addresses, they cannot be made persistent in that form. Therefore, object references must be converted to other forms that can be made persistent. Because persistence of all attributes, not just object relationships, is implemented in the data object, not the business object, this conversion is described further in “Data Object Customization for Cardinality Relations” on page 239.

As previously mentioned, in the Component Broker architecture, a business object's persistent state is managed by its associated data object. Links (1-1 and 1-N object relationships) are no different. The persistent representation of a link is also typically managed by a data object. However, there might be cases where it makes more sense for the business object to manage the link itself. For instance, if the link can be computed based on some application domain business logic, then it makes sense not to burden the data object with managing the link.

For example, assume that a claim maintains a cardinality-1 link to the insurance policy with which it is associated. Assume as well that the design of the application is such that the primary key of a Claim (the claim number) includes the policy number of its associated Policy, for example, claim numbers are actually prefixed by their associated policy number. In this situation, implementing the Policy reference involves no additional persistent data. The Policy get method could be implemented directly in the business object as follows:

```
Policy_ptr ClaimBO_Impl::thePolicy()
{
   ByteString_var pkString;
   // get my primary key
   ClaimPrimaryKey_var myKey = ClaimPrimaryKey::_create();
   myKey->fromString(pkString = this->getPrimaryKeyString());

   // extract the policy number from my primary key
   long myPolicyNo = myKey->policyNo();

   // get the policy home somehow (for example, using a factory finder)
   IHome_var policyHome = ...

   // create and initialize a policy primary key
   PolicyPrimaryKey_var policyKey = PolicyPrimaryKey::_create();
   policyKey->policyNo(myPolicyNo);

   // get the policy from the policy home
   IManagedClient::IManageable_var moPtr;
   return Policy::_narrow(moPtr = policyHome->findByPrimaryKey(policyKey));
}
```

As shown, this algorithm does not involve an explicit representation of the converted pointer. It is extracted from the key and is based on the application-level knowledge that claim numbers are prefixed by their associated policy number. In this situation the persistent representation of the link is actually maintained as part of another one of the object's attributes. Also, when a link can be computed based on some business logic, the object reference is often considered to be a read-only attribute of the business object.

While it is possible for a business object to manage object references itself, more commonly the business object passes the pointer to the data object and it performs the required conversion. With this approach, every data object that is used with a particular business object is free to use whatever conversion algorithm (and persistent representation) it chooses. This approach allows the business object to remain de-coupled from the persistent storage mechanism and thus possibly be re-used in different scenarios with a wide range of back-end data stores.

Cardinality-N Relationships

The previous section shows how to link an insurance Policy to a single Claim object. If, instead, the Policy object needs to reference multiple Claims, a cardinality-N relationship is required.

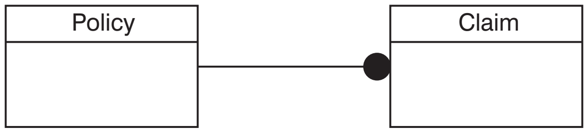


Figure 40. Cardinality-N Relationship

There are two approaches for providing a type-safe implementation for a cardinality-N relationship in Component Broker. Both approaches rely on a persistent collection object to manage the references, although one hides the collection in the implementation, while the other exposes it to clients as a first class object.

With the first approach, where the collection is hidden, the interface for adding and removing elements is provided on the Policy object itself. With this approach, the Policy business object interface would provide a set of methods for establishing, deleting, and accessing the links to Claim objects. The interface could be defined as follows:

```
interface Policy : ...
{
   void addClaim(in Claim claim);
   void removeClaim(in Claim claim);
   IManagedCollections::IIterator listClaims();
   ...
};
```

Clients would then use this interface to add, access, and remove claims, as follows:

```
Policy_var aPolicy = ...
Claim_var aClaim1 = ...
Claim_var aClaim2 = ...

//
// Add a couple of claims to the policy's set of claims
//
aPolicy->addClaim(aClaim1);
aPolicy->addClaim(aClaim2);

//
// Iterate through the policy's set of claims
//
// Get an iterator for the set of claims
IManagedCollections::IIterator_var iter = aPolicy->listClaims();
IManagedClient::IManageable_var element;
while (element = iter->next())
{
   // iterate through the set of claims
   Claim_var aClaim = Claim::_narrow(element);
   // do something with (for example, process) "aClaim"
```

```
    ...
}

//
// Remove a claim from the policy's set of claims
//
aPolicy->removeClaim(aClaim1);
```

As shown, although the underlying implementation of the relationship methods might use a collection object to manage the references, the client program is completely shielded from this fact. The Policy object encapsulates the collection behind the type-safe relationship interface methods, addClaim(), removeClaim(), and listClaims().

The second approach for implementing the cardinality-N relationship exposes the collection in the client programming model. Instead of referencing multiple Claim objects directly, the Policy object references a single collection object using a cardinality-1 link. The collection, in turn, references the multiple claims.

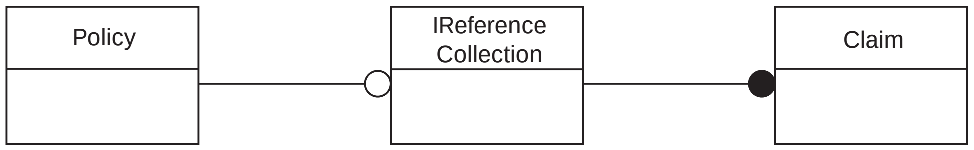


Figure 41. Cardinality-1 link to a collection object that references claims.

This picture actually applies to the previous approach as well, although there the reference collection is hidden in the implementation. One difference is that when the collection is hidden, the Policy interface is the only client interface for adding elements to the collection and therefore it provides static type checking for the elements in the collection. If, on the other hand, the collection is visible to clients, the collection itself must prevent clients from adding objects other than Claims to the relationship.

When using the explicit collection approach for implementing a cardinality-N relationship, the Policy business object interface includes an attribute whose type is a reference to the collection.

```
interface Policy : ...
{
   readonly attribute IManagedCollection::IReferenceCollection claims;
   ...
};
```

The relationship of the collection object to the Policy object can be considered to be a standard cardinality-1 link as described in “Cardinality-1 Relationships” on page 115, with one exception, the attribute is read-only. The *readonly* attribute allows clients to add and remove elements in the collection but prevents them from replacing the collection itself. In some situations this latter operation might be warranted, in which case, the *readonly* tag would be removed. In general, however, you should prevent clients from modifying the reference to the collection.

A client program uses the *claims* attribute access method to access the collection of claims. Methods on the collection can then be called to manipulate and access the actual Claim references.

```
Policy_var aPolicy = ...
Claim_var aClaim1 = ...
Claim_var aClaim2 = ...

//
// Get the set of claims;
IManagedCollections::IReferenceCollection_var claims =
    aPolicy->claims();
```

```
//
// Add a couple of claims to the policy's set of claims
//
claims->addElement(aClaim1);
claims->addElement(aClaim2);

//
// Iterate through the policy's set of claims
//
IManagedCollections::IIterator_var iter = claims->createIterator();
    // get a claims iterator
IManagedClient::IManageable_var element;
while (element = iter->next())
{
   // iterate through the set of claims
   Claim_var aClaim = Claim::_narrow(element);
   // do something with (for example, process) "aClaim"
   ...
}

//
// Remove a claim ("aClaim1") from the policy's set of claims
//
claims->removeElement(aClaim1);
```

Implementing the Relationship Interface

Depending on the interface requirements and possibly the back-end datastore being used, many different implementations of the Policy/Claim relationship are possible. Some common approaches include implementing the relationship using:

- A simple Non-keyed reference collection. See “Implementing a Relationship with a Simple Reference Collection.”
- A Keyed reference collection. See “Implementing a Relationship with a Keyed Reference Collection” on page 130.
- A Home or Collection and some additional information that is used to identify a subset of the entries in the collection. Some examples include a non-unique secondary key supported by the Home or Collection and a query evaluation string if the Home or Collection is queryable. See “Subsetting a Home or Collection” on page 131.

The following sections describe each of these implementation approaches.

Implementing a Relationship with a Simple Reference Collection: The relationship collection is exposed in the data object interface as a readonly attribute of type IReferenceCollection:

```
interface PolicyDO : ...
{
   readonly attribute IManagedCollections::IReferenceCollection claims;
   ...
};
```

The relationship interface methods, addClaim(), removeClaim(), and listClaims(), would be implemented as shown in the following example:

```
::CORBA::Void PolicyBO_Impl::addClaim(Claim_ptr claim)
{
   IManagedCollections::IReferenceCollection_var claims = claims();
   claims->addElement(claim);
}

::CORBA::Void PolicyBO_Impl::removeClaim(Claim_ptr claim)
{
   IManagedCollections::IReferenceCollection_var claims = claims();
   claims->removeElement(claim);
}

IManagedCollections::IIterator_ptr PolicyBO_Impl::listClaims()
{
   IManagedCollections::IReferenceCollection_var claims = claims();
   return claims->createIterator();
}
```

In each of the methods, the method claims is used to return a pointer to the reference collection containing the claims. Each method then delegates to its corresponding method on the reference collection.

The persistent object reference for the reference collection itself can be implemented using any of the design patterns described in “Cardinality-1 Relationships” on page 115. Using the lazy evaluation caching pattern in the business object, the claims method would be implemented as follows:

```
IManagedCollections::IReferenceCollection_ptr PolicyBO_Impl::claims()
{
    // first time?
    if (fCachedClaims == IManagedCollections::IReferenceCollection::_nil())
        fCachedClaims = fDataObject->claims();
    return fCachedClaim;
}
```

Before the data object claims method can return a collection, a reference collection must actually be created. This is typically done in the data object claims method the first time it is called. As previously discussed, the reference collection used to implement a relationship may be required to guarantee that the type of elements added to it are of a specific type (for example, Claims). A type-specific reference collection can be created by calling the createCollectionFor method on the IManagedCollections::ICollectionHome interface:

```
// get the collection home (for example, using a factory finder)
IManagedCollections::ICollectionHome_var cHome = ...

// create a reference collection that will only hold claims
IManagedCollections::IReferenceCollection_var rc =
     cHome->createCollectionFor(Claim::Claim_RID);
```

Alternatively, the simpler createCollection method can be used when no restriction on the type of collection element is required:

```
rc =    cHome->createCollection();
```

Implementing a Relationship with a Keyed Reference Collection: A Keyed reference collection can be used to implement relationships where the individual links are accessed using some kind of identifier. For example, if the claims associated with a given insurance policy are identified by, for example, a claim ID, the relationship interface might appear in the Policy business object as follows:

```
interface Policy : ...
{
   void addClaim(in Claim claim, in long id);
   void removeClaim(in long id);
   Claim getClaim(in long id);
   IManagedCollections::IIterator listClaims();
   ...
};
```

This relationship is most easily implemented using a keyed collection. For example, the data object interface might now include an attribute of type IKeyedReferenceCollection:

```
interface PolicyDO : ...
{
    readonly attribute IManagedCollections::IKeyedReferenceCollection claims;
    ...
};
```

The addClaim method could be implemented as follows:

```
::CORBA::Void PolicyBO_Impl::addClaim(Claim_ptr claim, ::CORBA::Long id)
{
   NumberKey_var key = NumberKey::_create();
   key.setValue(id);
   IManagedCollections::IKeyedReferenceCollection_var claims = claims();
   ::ByteString_var keyString = key.toString();
   claims->addElementByString(claim, keyString);
}
```

In this example, a key helper class (see Chapter 4, “MOFW Client Programming Model” on page 33), NumberKey, is used to add the element to the collection. The class NumberKey wrappers the ID in a key class that is capable of being stringified. When the key is created and initialized, the add operation is delegated to the reference collection.

The removeClaim and getClaim methods would be implemented similarly. The remaining method, listClaims, as well as the claims method that is used to access the collection object, would be implemented as shown in the non-keyed reference collection example.

Subsetting a Home or Collection: This approach uses a home augmented with information that identifies a subset of the objects in the home. This approach is particularly applicable in bottom-up development scenarios where related data already exists in legacy applications.

For example, an RDB-based insurance application might contain two related tables, one for policies and one for claims. The claims registered against a particular policy are identified by a policy# column in the claim table. This column, a foreign key in the claim table, is the primary key for the policy table.

POLICY TABLE

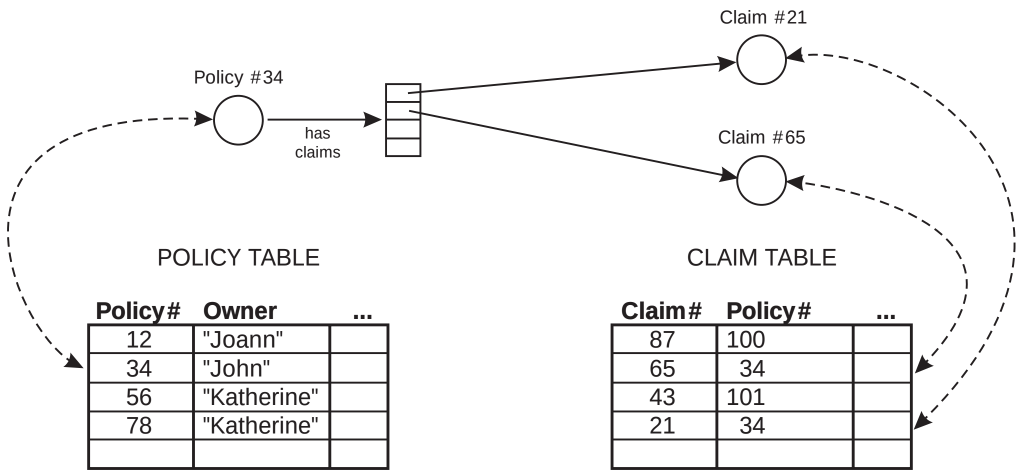
Policy#	Owner	...
12	"Joann"	
34	"John"	
56	"Katherine"	
78	"Katherine"	

CLAIM TABLE

Claim#	Policy#	...
87	100	
65	34	
43	101	
21	34	

Figure 42. RDB-based Insurance Application Tables

In Component Broker object space, these tables represent a cardinality-N relationship between a policy object and its associated claims. For example, policy number 34 would have links to claim numbers 21 and 65.



Policy#	Owner	...
12	"Joann"	
34	"John"	
56	"Katherine"	
78	"Katherine"	

Claim#	Policy#	...
87	100	
65	34	
43	101	
21	34	

Figure 43. Links Between Policy and Claims

Normally these links are stored as converted pointers maintained by a persistent reference collection of some type. In this case, the relationship information stored in the claim table itself eliminates the need for a persistent reference collection. The set of claim links for a particular policy can be derived using a query on the claim table. In Component Broker object space, this can be implemented by evaluating an OOSQL query (for example, `policyNo == 34`) on a queryable claim home.

```
  IManagedAdvancedClient::IQueryableIterableHome_var claimHome = ...

  IManagedCollections::IIterator_var iter = claimHome->evaluate("policyNo=34");

  // iterate over the claims
...
```

In this example, the home object would be retrieved using the Naming Service. The home name could be hard coded in the application, available in an environment variable, or stored persistently in the policy table.

The particular pattern used for finding the home might or might not be dictated by legacy requirements.

Because the relationship is derived from other data, adding and removing claims often involve side effects. For example, adding a claim to the reference collection would require a change in state to the claim object (for example, policy# field must be updated). If the claim is already in a relationship with another policy, it would be implicitly removed from it as a result of the add operation here. To avoid this kind of change in semantics of the addClaim method, the relationship interface could be changed to one that is more semantically consistent with the underlying implementation. For example:

```
  interface Policy : ...
  {
      Claim createClaim( in long id);
      void deleteClaim(in long id);
      Claim getClaim(in long id);
      IManagedCollections::IIterator listClaims();
      ...
  };
```

By replacing the addClaim and removeClaim operations with createClaim and deleteClaim methods, the relationship semantics map well to the home-based implementation.

Creating Specialized Homes

Component Broker provides a default IManagedClient::IHome implementation. There might be specializations of this interface specific to an underlying application adaptor. These include IManagedAdvancedClient::IQueryableIterableHome and IManagedAdvancedClient::IIteratableHome. This section explains how to extend a home with domain-specific methods for create and find.

There are a number of cases where the usage of IManagedClient::createFromPrimaryKeyString, IManagedClient::createFromCopyString and IManagedClient::findByPrimaryKeyString might not present the optimal interface for clients wishing to interact with the home to create and find business objects.

This section describes the process of extending the home and follows the pattern set out in “Extending a Business Object” on page 105.

Extending the Interface to IHome

For the example, PolicyHome is an interface that specializes the IHome interface with some specific create and find methods. The goal is to provide methods specific to the insurance policy abstraction. These methods are shown in the IDL in the following example:

```
Policy create(in float premium, in float amount);
                       // create a policy passing in the attribute values
Policy defaultCreate();
                       // default create method - policyNumber is assigned
Policy createWithNumber(in long policyNo);
                       // create a new policy with this number
Policy findPolicyByNumber(in long policyNo);
                       // find a policy by number
```

You must first decide which IHome interface should be specialized. This seems to have a simple solution: Inherit from the IHome in the managed object framework and get an interface that looks like this:

```
#include <IManagedClient.idl>
#include "Policy.idl"

interface PolicyHome : IManagedClient::IHome
{
    Policy create(in float premium, in float amount)
        raises(IManagedClient::IInvalidKey, IManagedClient::IDuplicateKey);
    Policy defaultCreate()
        raises(IManagedClient::IInvalidKey, IManagedClient::IDuplicateKey);
    Policy createWithNumber(in long policyNo)
        raises(IManagedClient::IInvalidKey, IManagedClient::IDuplicateKey);
    Policy findPolicyByNumber(in long policyNo)
        raises(IManagedClient::IInvalidKey, IManagedClient::INoObjectWKey);
};
```

Details

IManagedClient::IHome is being extended because it is the interface that is supported by all of the application adaptors provided by the server.

Exceptions have also been put on the methods that are introduced. For completeness, creation methods should raise the IManagedClient::IInvalidKey and IManagedClient::IDuplicateKey exceptions. Find related methods should raise the IManagedClient::IInvalidKey and IManagedClient::INoObjectWKey exceptions. Additional exceptions can also be introduced and raised as appropriate.

An alternative exception strategy is to have the specialized home implementations actually handle some of the exceptions. For example, catching the IManagedClient::IDuplicateKey exception on methods where the key is not passed in could be appropriate.

Alternatives to IManagedClient::IHome

Other alternatives for extending homes are IManagedAdvancedClient::IIterableHome and IManagedAdvancedClient::IQueryableIterableHome. This is true only if the additional interface supported by these extended homes is to be a proper superset of that which is available for the particular application adaptor configuration. Not all homes support these IManagedAdvancedClient interfaces.

Implement the Extended IHome Interface

The IHome itself is a managed object and the development of an extended home should be done like creating any subclass of a managed object. This is described in “Extending a Business Object” on page 105. This section highlights things that are specific to the IHome interface and extension.

Implementation Interface

The PolicyHomeBO.idl file inherits according to the pattern described in “Extending a Business Object” on page 105. It is shown in the following example:

```
#include <PolicyHome.idl>
#include <IManagedAdvancedServer.idl>
interface PolicyHomeBO : PolicyHome, IManagedAdvancedServer::ISpecializedHome
{
};
```

The PolicyHomeBO.ih file contains the PolicyHomeBO_Impl class. This class inherits the implementation from the IManagedAdvancedServer::ISpecializedHome_Impl that it plans to extend. This might vary from what was described previously, but this example continues to use the extension to the default Home implementation provided by the Managed Object Framework. The implementation interface is shown in the following example:

```
#include <IManagedAdvancedServer.ih>
#include "PolicyHomeBO.hh"
class PolicyHomeBO_Impl : public virtual ::PolicyHomeBO_Skeleton,
                          public virtual IManagedAdvancedServer::ISpecializedHome_Impl
  {
   public:
     ::Policy_ptr create (::CORBA::Float premium, ::CORBA::Float amount);
     ::Policy_ptr defaultCreate ();
     ::Policy_ptr createWithNumber (::CORBA::Long policyNo);
     ::Policy_ptr findByPolicyNumber (::CORBA::Long policyNo);

     // Methods from IManagedServer
```

```
      virtual ::CORBA::Void initForCreation (::IManagedServer::IDataObject_ptr theDO);
      virtual ::CORBA::Void initForReactivation (::IManagedServer::IDataObject_ptr theDO);
      virtual ::CORBA::Void uninitForDestruction();
      virtual ::CORBA::Void uninitForPassivation);
      virtual ::CORBA::Void syncToDataObject();
      virtual ::CORBA::Void syncFromDataObject();

    private:
      // superclass pointer of DO we are using
      IManagedAdvancedServer::ISpecializedHomeDataObject_ptr myDO;
  };
```

To create a specialized home that has query and iterable capabilities, simply replace IManagedAdvancedServer::ISpecializedHome with IManagedAdvancedServer::ISpecializedQueryableIterableHome in the PolicyHomeBO.idl and replace IManagedAdvancedServer::ISpecializedHome_Impl with IManagedAdvancedServer::ISpecializedQueryableIterableHome_Impl in the PolicyHomeBO.ih file.

The Implementation

The implementation of the new create methods involves careful usage of keys, copies, and the basic interface supported by home. All of the create methods end up using createFromCopyString or createFromPrimaryKeyString, passing a key or copy that has been loaded with the proper information to all proper creation of the object. The findByNumber() method follows a similar pattern.

Note: Care should be taken with CORBA types returned by these methods. Some types such as CORBA::String require special operations be used for return values. See the Appendix C, "C++ CORBA Programming" on page 299 for additional information.

create()

```
  ::Policy_ptr PolicyHomeBO_Impl::create(::CORBA::Float premium, ::CORBA::Float amount)
  {
     CORBA::Long policyNo = getUnique();
     cout << "The pseudo-unique policyNo will be: " << policyNo << endl;
     // create a key from that number
     PolicyKey_var theKey = PolicyKey::_create();
     theKey->policyNo(policyNo);
     // do findBy with that number to ensure that it isn't a duplicate
     Policy_var tPolicy;
     tPolicy=NULL;
     try
     {
        tPolicy=Policy::_narrow(IManagedClient::IManageable_var moPtr =
            findByPrimaryKeyString(ByteString_var findKeyStr = theKey->toString()));
        if (tPolicy)
        {
           // The Object already exists..
           throw IManagedClient::IDuplicateKey();
        }
     }
     catch(IManagedClient::INoObjectWKey &nowk)
     {
        // The Object does not exist. Good we can create it
        // just eat this exception and continue
        // If object not found on server, go ahead and create one.
        // create a copy object
        PolicyCopy_var theCopy = PolicyCopy::_create();
        // load it up with the right stuff
        theCopy->policyNo(policyNo);
        theCopy->premium(premium);
        theCopy->amount(amount);
        // call createFromCopyString
        IManageable_var aManageable;
        aManageable=createFromCopyString(ByteString_var findKeyStr = theCopy->toString());
        return Policy::_narrow(aManageable);
     }
  }
```

defaultCreate()

```
  ::Policy_ptr PolicyHomeBO_Impl::defaultCreate ()
  {
     ::CORBA::Long policyNo;

     // There is a  userData' attribute on each home. User specific
     // information can be stored in this field. We have chosen to
     // store the key generation algorithm type that should be used
     // when creating a new key.

     // Go to my DO to get the userData attribute value out of the
     string_var phUserData = myDO->getConfigInfo();
     if (strcmp(phUserData,"KeyAlgorithm1") == 0 )
     {
        // use the first key algorithm
        policyNo = getUniqueKey();
     }
     else
     {
        // use the second key algorithm
        policyNo = 100 + getUniqueKey();
     }

     // create a key from that number
     PolicyKey_var theKey = PolicyKey::_create();
     theKey->policyNo(policyNo);
     // do findBy with that number to ensure that it isn't a duplicate
     Policy_var tPolicy;
     tPolicy=NULL;
     try
     {
        tPolicy=Policy::_narrow(IManagedClient::IManageable_var moPtr =
             findByPrimaryKeyString(ByteString_var findKeyStr = theKey->toString()));
        if (tPolicy)
        {
           throw IManagedClient::IDuplicateKey();
        }

     }
     catch(IManagedClient::INoObjectWKey &nowk)
     {
        // object does not exist... we can create
        // call createFromPrimaryKeyString
        return Policy::_narrow(IManagedClient::IManageable_var moPtr =
             createFromPrimaryKeyString(ByteString_var findKeyStr = theKey->toString()));
     }
```

```
}
```

createWithNumber()

```
::Policy_ptr PolicyHomeBO_Impl::createWithNumber (::CORBA::Long policyNo)
{
   // create a key
   PolicyKey_var theKey = PolicyKey::_create();
   theKey->policyNo(policyNo);
   // call createFromPrimaryKeyString
   return Policy::_narrow(IManagedClient::IManageable_var moPtr =
        createFromPrimaryKeyString(ByteString_var findKeyStr = theKey->toString()));
}
```

Note that this method does not perform exception checking. If something other than IManagedClient::IDuplicate should be thrown when the number provided as input is already in use, then a try/catch block and associated logic would be required.

findByNumber()

```
::Policy_ptr PolicyHomeBO_Impl::findByPolicyNumber (::CORBA::Long policyNo)
{
   // create a key
   PolicyKey_var theKey = PolicyKey::_create();
   theKey->policyNo(policyNo);
   // call findByPrimaryKeyString
   return Policy::_narrow(IManagedClient::IManageable_var moPtr =
        findByPrimaryKeyString(ByteString_var findKeyStr = theKey->toString()));
}
```

Meet MOFW IManageable Requirements

Because this extension to the home introduces no new additional methods and no new key for the home itself, getPrimaryKeyString, getHandleString, externalize_to_stream, and internalize_from_stream do not need to be implemented.

MOFW Requirements – IManagedObject Interfaces

This section includes the following:

- "initForCreation()"
- "initForReactivation" on page 138
- "uninitForDestruction" on page 138
- "uninitForPassivation" on page 138
- "syncFromDataObject" on page 138
- "syncToDataObject" on page 138

initForCreation(): The initForCreation method is only required to call the parent, passing the dataObject that comes in as a parameter. Currently, homes are not actually created using this method. Homes are configured onto systems and brought into existence as part of server initialization. This code is not executed; the example is included for completeness.

```
::CORBA::Void PolicyHomeBO_Impl::initForCreation (::IManagedServer::IDataObject_ptr theDO)
{
   // call my parent
   IManagedAdvancedServer::ISpecializedHome_Impl::initForCreation(theDO);
}
```

initForReactivation: Set the data object using the same pattern as initForCreation. Any other specialized home specific code that needs to execute when a home is activated goes in this method implementation.

```
::CORBA::Void PolicyHomeBO_Impl::initForReactivation (::IManagedServer::IDataObject_ptr theDO)
{
   //Call my parent
   ::IManagedAdvancedServer::ISpecializedHome_Impl::initForReactivation(theDO);
   // set my DO
   myDO = IManagedAdvancedServer::ISpecializedHomeDataObject::_narrow(theDO);
}
```

uninitForDestruction: uninitForDestruction also only calls the parent method.

```
::CORBA::Void PolicyHomeBO_Impl::uninitForDestruction()
{
   // call my parent
   ::IManagedSystemObject::IHome_Impl::uninitForDestruction(theDO);
}
```

uninitForPassivation: Call the parent following the same pattern as uninitForDestruction.

syncFromDataObject: Call the parent following the same pattern as uninitForDestruction.

syncToDataObject: Call the parent following the same pattern as uninitForDestruction.

Keys

The same class used for the MOFW home works here. Homes are found generally using factory finders and are not created. Keys are not used in the programming model. Keys are used in the internal server run time.

Copy Helper

The same class used for the MOFW home works here. Homes are not created, and therefore no copy helper is needed or usable based on the life cycle of homes.

Leveraging Server Provided Essential State Extensions

A specially defined attribute is passed through from the object builder or DDL that represents this specialized home. Rather than having a separate data object for the specialized home, it is much more efficient to access this data when it is needed. This is done as follows:

```
char* x = myDO->getConfigInfo();
if ( /* some evaluation of x */ )
{
   // make up a number using srand or some program that helps with this
   policyNo = /*whatever;
}
else
   policyNo=/* whatever */
```

This data is read-only and cannot be changed.

Overriding Specific Methods on Specialized Homes

In addition to supplying specific methods that allow clients to create and find objects without using keys and copy helper, specialized homes also provide the mechanism for overriding other interfaces supported by the IManagedClient::IHome interface. The following specific methods that can be overridden are described below:

- IManagedClient::IHome::createFromPrimaryKeyString
- IManagedClient::IHome::createFromCopyString
- IManagedClient::IHome::findByPrimaryKeyString

The IManagedClient::IHome::createFromPrimaryKeyString and createFromCopyString can be overridden to prevent creation of objects of a particular type. There are cases when using implementation inheritance and polymorphism where homes are configured onto a system even when the class is abstract. This is usually to facilitate polymorphic findByPrimaryKeyString operations. While findByPrimaryKeyString is desired as a polymorphic operation, the create capabilities may in fact be prohibited. Other reasons include special checking that needs to be done before the actual create is issued. While this can be done with specialized create methods, using the base programming model methods of createFromPrimaryKeyString and createFromCopyString may be desirable in some situations.

Overriding IManagedClient::findByPrimaryKeyString is done to facilitate special find logic. This is most common in cases where polymorphic relationships exist between business objects. For example, if an abstract class of a has subclasses of “b” and “c,” it would be reasonable to try to find an “a” with a given key. While there are no “a” instance because it is abstract, it is reasonable to have an overridden findByPrimaryKeyString that would look at all concrete subclasses of “a” to properly execute the find operation. By overriding findByPrimayKeyString instead of using a specialized find method, data objects that deal with polymorphic relationships can work unchanged. They depend on findByPrimaryKeyString as part of the attribute getter implementation.

The contacts of these methods must be maintained even when they are overridden. The exceptions thrown must still be honored by the specialized home. For example, IManagedClient::IDuplicateKey must still be thrown by create and IMangedClient::INoObjectWKey should be returned. These are integral to much of the mainline programming practice in Component Brokerclient programs. This contract must be maintained even in the specialized home overridden versions of these methods. New exceptions cannot be introduced on these methods as there is not overriding or overloading of interface specification allowed in IDL.

Summary of Home Extension

Extending a home is much like extending any business object but simpler. Refer to “Create the Managed Object Class and Implementation” on page 197 for more details.

Copy Helpers – Sharing Opportunities

Examination of the examples presented in this book might raise some questions about redundancy and duplication of interface and implementation. When the Policy example was originally described, there were separate classes for the Copy Helper Class (PolicyCopy) and for the actual interface that clients generally code to for business logic purposes (Policy). These classes look alike, yet have no inheritance or other meaningful relationship between them. They are essentially in separate class hierarchies partially because the CopyHelper classes are ILocalOnly types of classes while the real business objects are true CORBA objects.

While sharing implementations might not make sense, it is possible to share interfaces between the actual business object interface and the CopyHelper class. In the example, a new abstraction called PolicyCommon was introduced. This is an abstract class that introduces that which is common between the Policy Copy Helper and the Policy interface used for building the business object. Notice that Policy remains an IManageable, PolicyCopy remains an INonManageable, and PolicyCommon has no parent.

Although this example might not appear to simplify and reduce complexity, more realistic cases that include more methods and attributes might make this reasonable.

It is mostly attributes that go into the *xxxx*Base class, although methods could also be included. All attributes that are part of the real business object that are meaningful as part of a transient copy are candidates. If there are methods that apply to both the real business object and a transient copy, then those methods can go into this *xxxx*Base class. Methods that are used to validate and edit the attributes are the best candidates for inclusion into the *xxxx*Base class.

Figure 44 shows an example of how this might be done.

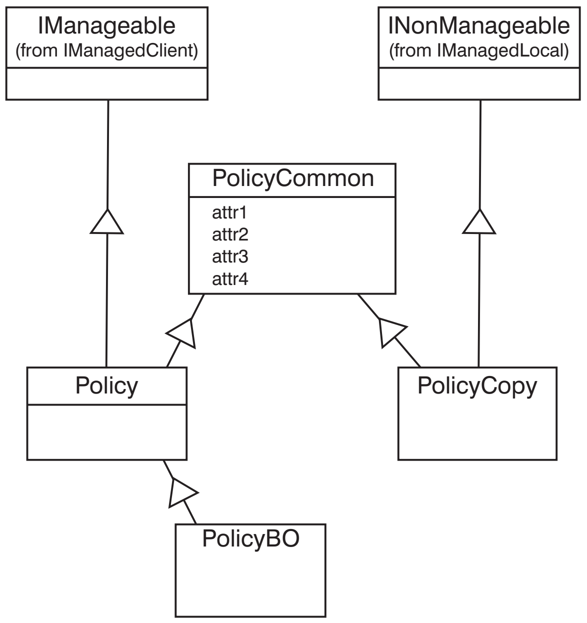


Figure 44. Example of Interface Sharing Opportunities

Moving Object Data in Bulk

One of the important design points in any distributed system is to “reduce trips over the wire.” This basic guideline will cause a number of low level design and implementations decisions to be made in object systems. There is always the risk of exposing too much data and violating encapsulation. However, performance often times takes precedence over design purity.

During the life-cycle of an object there are a number of points where the “reduce trips over the wire” guidelines should be considered. These will be explained in the following paragraphs.

Object creation is the first opportunity to reduce trips over the wire. The createFromCopyString method on all Component Broker homes is the simple answer to creating objects with multiple attributes from remote clients in an efficient way. If a business object has five attributes, of which only one is the key, then it is obviously more efficient to use createFromCopyString as opposed to a createFromPrimaryKeyString followed by four setter methods on the newly-created business object. Specialized homes can also offer create methods that gather up as much from the client as is reasonable and take it over the wire on one method call to the specialized home.

Once created, the ability to move bulk data for objects becomes more complicated. Some of the possible patterns are described.

For situations where a subset of the state is needed to prime user interfaces a data array query (evaluate_to_data_array method on a query evaluator) should be considered. This returns tuples, not objects that represent the data for the objects. If usage of the actual objects is also expected to occur, then the object reference should be requested as a return value from the evaluate_to_data_array call. This reduces trips over the wire. However, there is another opportunity. Even when using a data array query, there is no way to update multiple object attributes in one call. The evaluate_to_data_array got lots of data, but there is no obvious way to use it to update the objects they represent.

In our example of a business object with five attributes, an update(a1,a2,a3,a4,a5) method on the business object could perform this 'update all attributes' function. The implementation of this method in the business object would probably validate that the key is the same and update all non-key attributes. Remember, keys cannot change on managed objects. A variation on update is to use an update of the form update(copyHelperString). This does not change the trips over the wire, but offers a different style for client programmers to use.

As with any copies or caches, be very precise about when the copy is update to date and when it might be considered 'dirty'. Responsibility for keeping the business object synchronized with the copy falls to the business object provider. Additional logic may be appropriate in the update() method to ensure data integrity and to prevent unnecessary updates to the third-tier resource managers.

Treating object data as structures or buffers should be done as a performance optimization. It should not be a general practice when doing object-oriented programming due to the reduction in encapsulation that is introduced. This is a performance tuning and optimization technique.

Multiple Interfaces to Business Objects

There are situations where a business object has different sets of clients. Good object-oriented analysis and design generally leads developers to interfaces that start with a base or minimal set of capabilities and then introduces additional interfaces that have an increasing set of methods. Be careful defining these interfaces and when handing them out to clients to avoid compromising the encapsulation that is necessary for a robust object design.

Figure 45 on page 142 is an example of factoring the interface to be used by clients.

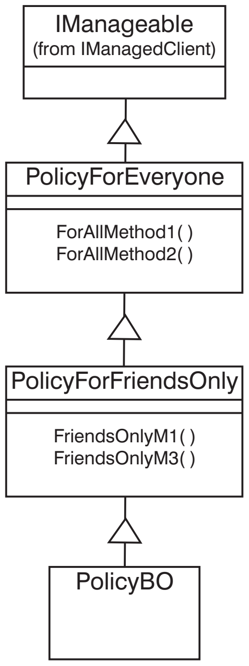


Figure 45. Example of Interface Factory

The previous example factors the Policy interface into PolicyForEveryone and PolicyForFriendsOnly. For clients that need access only to the PolicyForEveryone interface, the client side DLL (in the C++ case) should include only bindings for the PolicyForEveryone interface. Clients that want to access all of the methods in both the PolicyForEveryone interface and PolicyForFriendsOnly would need a client side DLL (in the C++ case) that contained C++ bindings for both of these interfaces.

Other factorings are possible.

Circular References

Business object interfaces that have circular references should all be contained within a module. This means that if interface A includes interface B, and interface B refers to something from interface A, both of these interfaces are required to be in the same module. While not recommended, it is also possible to have circular references like this at the global scope.

Chapter 8. MOFW – ActiveX Client Programming Model

This chapter discusses the specifics of:

- Using an ActiveX client to access managed objects on a Component Broker server.
- Developing code using managed object proxies in an ActiveX environment

This chapter follows the format of Chapter 4, “MOFW Client Programming Model” on page 33, excluding the discussions of abstract concepts.

ActiveX Client View of Component Broker Applications

The intent is to keep the interface to Component Broker objects as familiar to the ActiveX developer as possible.

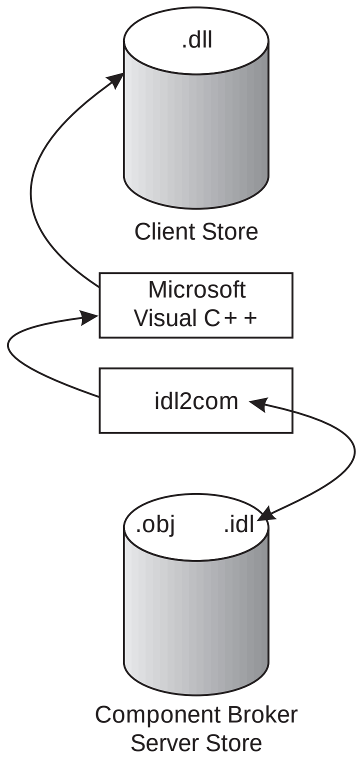


Figure 46. ActiveX Client View

The development view of the managed object (MO) proxies that are used on the client platform starts on the server. When developers are programming managed objects for Component Broker, they are obliged to create IDL files which represent those managed objects. Typically, the Component Broker developers use the Object Builder to help them coordinate these files, but that is not required.

The key, for an ActiveX client developer, is that the IDL for the MOs must be run through the idl2com compiler. The idl2com compiler takes the interface definitions in the IDL, and produces the managed object proxies that are used from the client. The proxies are produced in C++. Supporting files that need to be compiled are placed on the client for client applications to use. See Developing the Component Broker ActiveX Client for further information.

As in Figure 46, the route to producing the C++ files for the managed object proxies is by calling idl2com.

Once the C++ files for the managed object proxy are available, the proxy class can be treated as any other C++ class, with the exception of the specific requirements added to the class based on its involvement with the CORBA based Component Broker server. These are detailed throughout this chapter. Look at the run-time elements of the solution.

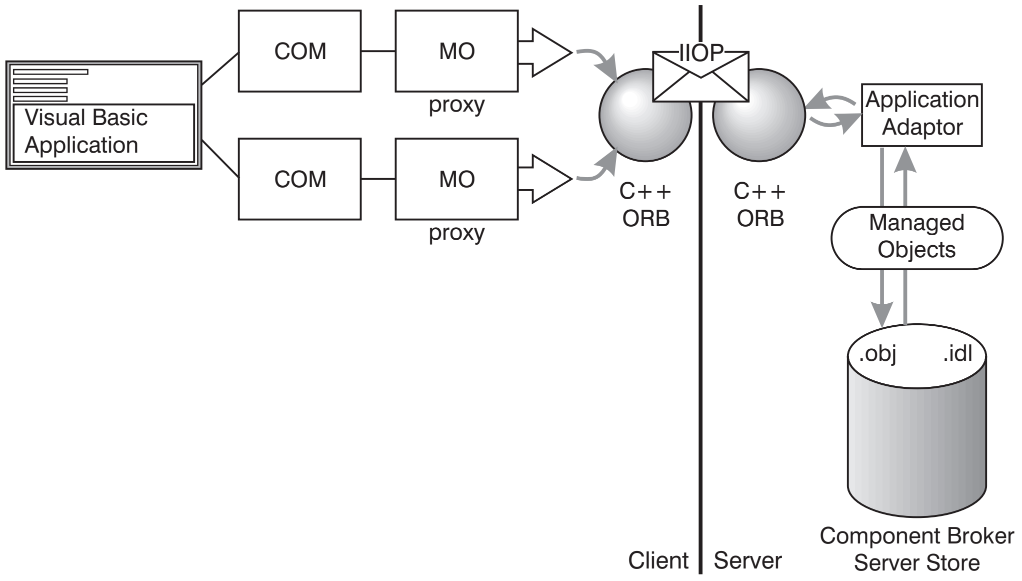


Figure 47. ActiveX Client Run-time Scenario

A run-time scenario for an ActiveX client begins when the managed object proxy is accessed. All managed objects accessed from the ActiveX client are *wrapped* at run time by a COM object. The first order of business is to connect to the ORB and establish a reference to the root of the system name space. From there, the issues of finding and using the managed objects through the proxy are issues related to how the Component Broker programing model is set up. Only the syntax changes for the ActiveX environment.

Developing the Component Broker ActiveX Client

To produce a DLL containing an ActiveX accessible interface to a CORBA object, perform the following basic steps:

1. Create your IDL.
2. Process IDL using idlc to produce the client side bindings.
3. Process IDL to generate the .ih and _I.cpp files if needed.
4. Generate a GUID using guidgen.exe.
5. Process IDL using idl2com to produce the ActiveX accessible interfaces.
6. Compile and link the bindings.
7. Register the interfaces using regsvr32.

At this point the interfaces are now available for application usage.

Generating and Registering DLLs

More detailed steps on generating and registering DLLs follow:

1.

   Create your IDL either manually or by using Object Builder. See “Object Builder” in *Component Broker Application Development Tools* and *Component Broker for Windows NT and AIX Online Documentation* for details. If you choose to use Object Builder to create your IDL, it is important to know that you cannot use the client-side usage bindings generated by Object Builder. You can only use the *.idl, *_I.cpp and *.ih files. The *_I.cpp and *.ih files are only needed when you have defined localonly objects.

2. Emit the client-side usage bindings from the IDL. Use the idlc command to emit the client-side usage bindings from the IDL, specifying the `-mcpponly` and `-suc:hh` options on the command. For example:

   ```
   idlc -mcpponly -suc:hh Policy.idl
   ```

   Important: If you did not emit the client-side usage bindings into your ActiveX build directory, you must copy or move them there now.

   See the "idlc Command" in the *Component Broker Programming Reference*for details on idlc.

3. If you have defined localonly objects (that is your IDL contains #pragma localonly), you need to create the implementation headers and templates from the IDL. This step is **only** necessary if you have defined localonly objects. You have two options:

   - If you used Object Builder to create the IDL, use the *.ih and *_I.cpp files that Object Builder automatically generated.
   - Otherwise, manually generate the *.ih and *_I.cpp files using the idlc command.

   If using the idlc command to produce the *.ih and *_I.cpp files from the IDL, specify the `-mcpponly` and `-sih:ic` options on the command. For example:

   ```
   idlc -mcpponly -sih:ic Policy.idl
   ```

   You must now copy or move the *_I.cpp and *.ih files into your ActiveX build directory.

4. Generate a GUID using the guidgen.exe utility included with Microsoft Visual C++.

5. Use the idl2com command to produce the ActiveX accessible interfaces from the IDL. For example:

   ```
   idl2com -g AE3E2131-C6DE-11d0-92AF-08005ACE818D Policy.idl
   ```

   See the idl2com Command in the *Component Broker Programming Reference* for details on idl2com.

6. Use nmake -f [filename] to compile and link the ActiveX accessible interface DLL for the CORBA object. Use the IDL name, appended with the .mak extension for [filename]. For example:

   ```
   nmake -f Policy.mak
   ```

   When producing the DLL for the CORBA object, ensure that all required IDL for the object (including IDL referenced by the object's IDL) has been processed by the idl2com command into the same directory. This ensures that the correct header and library files are available when the .mak file is processed.

7. Use regsvr32 [filename] to register the DLL in the Windows system registry. Use the IDL name, appended with the .dll extension for [filename]. For example:

   ```
   regsvr32 Policy.dll
   ```

Unregistering and Moving DLLs

If you find in the future that you wish to move or no longer need the DLL(s) you have registered, perform the following steps:

1. Use regsvr32 /u [filename] to unregister the DLL. Use the IDL name, appended with the .dll extension for [filename]. If you no longer need the DLL, you are finished.

2. If the DLL produced in step 6 above is to be moved to a different directory, do the following steps:

   a. Move the DLL file and its corresponding TLB file to the new directory. The TLB file was generated during the makefile processing (step 4 above).

b. Reregister the DLL.

Component Broker ActiveX Client Application Development Information

Using COM or OLE objects Conformance to the OMG's COM-CORBA Interworking Part A specification is not complete. CORBA objects can be accessed through COM and OLE automation-produced interfaces, but CORBA objects can not access COM or OLE objects.

Using OLE automation interfaces While the produced OLE Automation interfaces are intended to be generic OLE automation interfaces available to any OLE controller, only Visual Basic 5.0 has been used to test these at this time.

Using remote CORBA objects During installation, several OMG COM-CORBA Interworking Specification interfaces are installed and registered. Of key importance are the CORBA Factory interfaces: GetObject and CreateObject. Use these interfaces to get started using remote CORBA objects. The samples provide guidance on using these interfaces.

Using IBM-supplied COM wrappers During installation, several pre-built COM/OLE Automation interface DLLs for some of the Object Services are provided and registered for you. These DLLs are contained within the installed bin directory, and supporting TLB, LIB, and header files are installed as well. The specific list of DLLS which are shipped is contained in the RegActX.bat file found in the installed bin directory. This file registers the DLLs.

Using VBScript and Internet Explorer A sample is provided that shows the use of a CORBA object from VBScript and Internet Explorer. This sample is available in the samples subdirectory (InstallVerification/ProgrammingModel/Applications/ActiveX) under the directory where Component Broker was installed.

Only the following data types may be passed to a CORBA object from a VBScript script:

- primitives (ie. long, short, char, boolean, etc)
- strings
- objects
- structures containing only primitives, strings or objects
- unions containing only primitives, strings or objects

Do not use arrays or sequences in a VBScript for this release. Exception Handling In Visual Basic, there are 2 means of working with exceptions: You can pass an “exception object” (a VARIANT type) as the last parameter to a method. The last parameter is always optional. If an exception occurs, this object will be filled in with the necessary exception information (a CCORBAException object.). You can pass a null for the last parameter which tells the bindings that there is no exception object to be filled in. The method instead needs to pass back an HRESULT which is then mapped by Visual Basic. The Visual Basic application has to put in place the ON ERROR code (similar to a C++ or Java try/catch block) to catch the exception.

Client Programming Model: Basic Tasks

In Chapter 4, “MOFW Client Programming Model” on page 33, the following tasks that a client is likely to want to do are discussed:

- Find an object.
- Use an object.
- Create an object.
- Use a set of object.

- Remember an object.
- Release or delete an object.

This chapter explores the same topics (with the same examples) but presents them from the perspective of ActiveX development. The examples are primarily in Visual Basic. An ActiveX client application developer can use the COM class output of the idl2com compiler directly from C++, and following COM rules, as well.

Initializing The Component Broker Client Environment

Initializing the Component Broker client environment is discussed in detail in Chapter 4, “MOFW Client Programming Model” on page 33. Initialization provides access to the ORB and Naming Service.

The following Visual Basic code accomplishes this task:

```
Dim corbaFactory as Object
Dim orb as Object
Dim NameService as Object
Dim genericObject as Object

Set corbaFactory = CreateObject("CORBA.Factory")
Set orb = CreateObject("CORBA.ORB")
Set NameService = CreateObject("IDL:IExtendedNaming.NamingContext")

Set genericObject = orb.ResolveInitialReference("NameService")
NameService.narrow genericObject
```

After initialization, the client has access to the *orb* and *nameService* variables. You can see how the client might use these in subsequent sections.

Remember that the IExtendedNaming is used to make things go a little easier.

The previous example shows how you can initialize the environment, and specifically, how access to the NameService can be achieved. However, see “Bound in the Naming Service” for information on how access to objects can be obtained without these steps, if the object is well-known in the name space. The ActiveX client run time performs the previous sequence of operations automatically on CreateObject/GetObject methods calls to the Corba.Factory COM object.

Finding a Managed Object

You can find an object in one of two ways. In the first technique, the object may have a Name and the client can use the Component Broker Naming Service to look up the object by its Name. In general, only a small subset of the object instances in a distributed system are in the Naming Service. These are typically large, well-known objects such as collections of business objects or important object instances in the Object Model.

The second technique for finding an object is to use the Naming Service to find a well-known object, for example, a collection, and then navigate to the desired object from the well-known object. Navigation occurs by looking in collections or following references to other objects.

Bound in the Naming Service

Assume the insurance company in the example placed several important Claim objects in the Naming Service. The following Visual Basic code example shows how to find such a Claim, belonging to a customer named Lou.

```
' nameService initialized as above

Dim tempObj as Object
Dim louClaim as Object
Set louClaim = CreateObject("IDL:Claim")
Set tempObj = nameService.resolve_with_string(".:/Applications/LifeInsurance/Claim/LouClaim")
louClaim.narrow tempObj
Set tempObj = Nothing
   ...
' No longer need louClaim
Set louClaim = Nothing
```

The Component Broker server run time initializes the global object instance nameService to refer to the root of the installation's Naming Service.

For a refresher on determining the naming context, and details on how the name space is specified, see Chapter 4, “MOFW Client Programming Model” on page 33.

Recall that in addition to the resolve_with_string() method, Naming Contexts also support the bind_with_string() method which associates a name with an object instance. The following Visual Basic code example could have been used to name the Lou Claim object.

```
' Declare and create louClaim prior to the following code segment
    ...
' Add Lou to the Name Space
nameService.bind_with_string("/.:/Applications/LifeInsurance/Claim/LouClaim", louClaim);
```

For additional information on the Component Broker Naming Service, see References in the Component Broker Online Documentation.

By Methods on Held Objects

Once you have an object, you can use its methods to find related objects. Continuing the previous example, after finding Lou's Claim, you can find other objects that the Claim references. The following Visual Basic example gets you a reference to Lou's Policy:

```
' Find Lou's Policy
Dim louPolicy as Object
Set louPolicy = louClaim.policy
```

Using the PrimaryKey Helper Class

The preceding example was simplified in that LouClaim was in the Naming Service. In the event that you do not have a well-known name for the claim, you can use Homes to find a specific claim.

Homes are instances of the IHome class. The managed object provider might decide to implement and provide a tailored subclass of IHome, or might use an instance of the base class. The relationship between managed objects and collections is explained in “Using Sets of Objects” on page 153.

For the overview, you can find the Home for Claim objects by the following Visual Basic code segment

```
Dim claimHome as Object
Dim myFinder as Object
Dim tempObj as Object

Set claimHome = CreateObject("IDL:Claim")
Set myFinder = CreateObject("IDL:IExtendedLifeCycle.FactoryFinder")
Set tempObj= nameService.resolve_with_string("/.:/Applications/LifeInsurance/Homes/Claim")
```

```
myFinder.narrow tempObj
Set tempObj = Nothing
Set tempObj = myFinder.find_factory_from_string("Claim.object interface")
claimHome.narrow tempObj
Set tempObj = Nothing
     ...
' claimHome is now usable
     ...
' No longer need myFinder
Set myFinder = Nothing
```

Now you need to find Lou's Claim. If you know the Claim number, all you need is the Primary Key Helper class for the Claim. Every managed object class has a set of local helper classes that let you use its keys. An instance of a Key Helper Class is always local to the client's process.

The provider of the managed object that you are working with has given you access to source for, or actual .DLL files containing the code for the Key Helper Class. Regardless, it is up to you to ensure that you have access to them and can use them in your applications.

Key Helpers, like all helper classes are created with a static method on the class named _create(). This static method gets generated by the bindings that accompany all subclasses of ILocalOnly. As an ActiveX client programmer, you can create a helper object by instantiating the ActiveX wrapper for the helper class.

Having created an instance of a Primary Key, the key must be set by one or more attributes on the Primary Key object. When all of the *key* attributes have been set, the Primary Key object is now usable. The Claim Home uses this Primary Key to find the previously created Claim object. Remember, the PrimaryKey is on the client system, but the Claim object and the Claim Home are on the server system. If the client passes a Primary Key object as a parameter to the Home, and the Home is on a remote system, the remote system might get a proxy back to the client's PrimaryKey instance. This would turn the client into a server and unpredictable results could occur. Therefore, the Component Broker programming model uses strings as the method for passing keys to potentially remote objects.

Continuing the example, the following Visual Basic code segment would find Lou's Claim in the Home (assuming his number is 1234).

```
' Create an instance of the Key Helper Class
Dim ClaimPrimaryKey as Object
Set ClaimPrimaryKey = CreateObject("IDL:ClaimPrimaryKey")

' Set the claimNo attribute in the key
claimPrimaryKey.ClaimNo = 1234

' Get the data out of the Stream to go onto the wire to the server
Dim claimStringvar as Variant.
claimStringvar = claimPrimaryKey.toString()

' Turn the variant returned by toString() method into a safearray of shorts
Dim claimString() as Integer
ReDim claimString(UBound(claimStringVar))
For counter = 0 to UBound(claimStringVar)
   claimString(counter) = claimStringvar(counter)
next counter

' Call find by key on the Home to find Lou's Claim
Dim tempObj as Object
Dim louClaim as Object = CreateObject("IDL:Claim")
```

```
Set tempObj = claimHome.findByPrimaryKeyString(claimString)
louClaim.narrow tempObj
Set tempObj = Nothing
  ...
' No longer need louClaim
Set louClaim = Nothing
```

The object provider of a public managed object always provides you with a set of helper classes for using the Homes that contain his managed objects. There is always exactly one Primary Key helper class. The object provider gives a client developer the:

- The interface definitions for the Key Classes. In addition to the Primary Key, there might be a secondary Key Helper Classes. A Secondary Key might also uniquely identify an object, or multiple instances may have the same value.
- An implementation of the Key Classes.
- Documentation for their use.

Using a Managed Object

When you find a reference to a managed object, you can invoke methods on it. For example,

```
person.set_Name("Lou Smith");
```

Calls the set_Name method on the person object identified by the person. The Component Broker internal implementation handles the use of remote objects.

Creating a Managed Object

Component Broker managed objects can be created in a number of ways. The following sections describe the default way in which you can easily create managed objects.

Creating a New Object – Create From Key

Every Component Broker managed object class has an instance of a Factory associated with it. The Factory provides a set of interfaces for creating instances of a managed object. The Factory gets some of its interface from the base class CosLifeCycle::GenericFactory. The createFromPrimaryKeyString method is introduced in the IManagedClient::IHome interface supplied by Component Broker. This interface specializes the CosLifeCycle::GenericFactory interface and plays the role of factory for Component Broker managed objects. Object providers might implement and provide a tailored subclass of this interface, or might use the implementation of IHome provided. You need to know how to find the right IHome for creation. Homes are at well-known locations in the Naming Service. The input required for the factory finder is the name of the interface of the class that you want this factory to make instances of. The following Visual Basic code fragment gets a reference to the Claim Factory for the Life Insurance Application.

```
Dim myFinder as Object = CreateObject("IDL:IExtendedLifeCycle.FactoryFinder")
Dim tempObj as Object
Set tempObj = nameService.resolve_with_string("/.:/Applications/LifeInsurance/FactoryFinders")
myFinder.narrow tempObj
Set tempObj = Nothing

Dim claimHome as Object = CreateObject("IDL:ClaimHome")
Set tempObj = myFinder.find_factory_from_string("Claim.object interface")
claimHome.narrow tempObj
Set tempObj = Nothing
      ...
```

```
' No longer need claimHome
Set claimHome = Nothing
```

You need to provide the IHome with information necessary to manufacture a new object instance. At a minimum, the Primary Key must be provided. A Visual Basic example of creating a new Claim with a *claimNo* of `1234` is:

```
' Create an instance of the Primary Key Helper Class
Dim ClaimPrimaryKey as Object
Set ClaimPrimaryKey = CreateObject("IDL:ClaimPrimaryKey")

' Set the claimNo attribute in the key
claimPrimaryKey.ClaimNo = 1234

' Get the data out of the Stream to go onto the wire to the server
Dim claimStringvar as Variant.
claimStringvar = claimPrimaryKey.toString()

' Turn the variant returned by toString() method into
' safearray of shorts
Dim claimString() as Integer
ReDim claimString(UBound(claimStringVar))
For counter = 0 to UBound(claimStringVar)
   claimString(counter) = claimStringvar(counter)
next counter

' Call createFromPrimaryKeyString on the Factory to create Lou's Claim
Dim tempObj as Object
Dim theClaim as Object = CreateObject("IDL:Claim")
Set tempObj = claimHome.createFromPrimaryKeyString(claimPrimaryKey)
theClaim.narrow tempObj
Set tempObj = Nothing
  ...
' No longer need theClaim
Set theClaim = Nothing
```

The previous two examples are almost identical. Create a Primary Key Object to define the identity of the object that is made. Then, the createFromPrimaryKeyString call is made on the Home and the Key is passed.

The createFromPrimaryKeyString method is defined by the IHome class, and all business objects can be created by this method.

An object provider might provide you with a subclass that introduces other, easier to use creation methods. Additional create methods are described, with examples, in Chapter 6, “MOFW Client Programming Model – Advanced Concepts” on page 83.

Creating a New Object - Create from Copy

Setting and getting the attributes of a managed object can be expensive. There are two main reasons for this. First, if the managed object is implemented in another language, each get or set method is actually a cross-language call. Cross language calls are more expensive than simple, same language calls. The get and set overhead is even more expensive if the managed object is remote because each call is actually a remote procedure call and involves significant overhead. Consider the following code segment:

```
' Assume that 'theClaim' is declared and created as in the
' previous code segment.

' Creating 'theClaim' above required one RPC. Setting the rest of the
' objects attributes involves one RPC per attribute. The following
' lines of code show four such RPC's. This could, of course, be any
' number of RPC's, depending on the complexity of the object.

' Now initialize the Claim's attributes
theClaim.date = "10/14/96"
theClaim.state= entered
theClaim.reason = accident
theClaim.description = "Side-swiped by teenager in a red convertible."
```

This segment could involve the following remote method calls:

- The client to Home of Claims to create the Claim.
- The client to Claim managed object to set its *date* attribute.
- The client to Claim managed object to set its *state* attribute.
- The client to Claim managed object to set its *reason* attribute.
- The client to Claim managed object to set its *description* attribute.

The previous calls could be reduced to a single remote method call by using the createFromCopyString() method on an IHome instead of the createFromPrimaryKeyString() method. To use the createFromCopyString() method, the object provider must provide you with a Copy Helper Class. The following Visual Basic code segment presents the code from the previous example rewritten using this design pattern.

```
' Create a new "local" Claim in my process and language.
' Use a Copy Helper Class that the Claim MO provider gave me.
Dim ClaimCopy as Object
Set ClaimCopy = CreateObject("IDL:ClaimCopy")

' Now initialize the Claim's attributes. Note that these methods
' execute locally, within the same language.
claimCopy.date = "10/14/96"
claimCopy.state = entered
claimCopy.reason= accident
claimCopy.description= "Side-swiped by teen-ager in a red convertible."

' Pass this local copy to the Home and have him return a new Claim
' MO whose attributes are initialized from the local copy's values.
' Since not all ORBs support Pass-By-Value, we first convert the
' local copy helper object to a string.
Dim tempObj as Object
Dim theClaim as Object = CreateObject("IDL:Claim")
Dim claimStringvar as Variant.

claimStringvar = claimCopy.toString()

' Turn the variant returned by toString() method into
' safearray of shorts
Dim claimString() as Integer
ReDim claimString(UBound(claimStringVar))
For counter = 0 to UBound(claimStringVar)
   claimString(counter) = claimStringvar(counter)
next counter
```

```
Set tempObj = claimHome.createFromCopyString(claimString)
theClaim.narrow tempObj

Set tempObj = Nothing
Set classClaimCopy = Nothing
Set claimCopy = Nothing
   ...
' No longer need theClaim
Set theClaim = Nothing
```

Like a Key Helper Class, a Copy Helper Class instance is always local to your process and implemented in the language you are using. The object provider gives you the interface and implementation of the helper class.

Copy Helper Classes are especially useful if the client application needs to interact with an object during initialization, and then create a managed object from the attributes. A common scenario for this is entering data for the object from a GUI. The GUI updates the local Copy Helper Object, and then the createFromCopyString() method is called when the **Do** button is clicked on the end user interface (EUI).

Releasing and Deleting Objects

Eventually, you no longer need to use an object that was created or found. Component Broker supports two interpretations on "no longer needs".

- The remove() method deletes the object and its persistent instance data.
- The release() method informs the object that the client application no longer plans to reference the object. The object still exists in the server, and other applications may be using it, but the client calling the release() method is done with the object.

Using Sets of Objects

An IHome represents a set of managed objects, all of the same type, whose relationship to one another is defined by the object provider, and maintained by the fact that they were all created in the same home. Sometimes an application needs to define (and manage) the relationships between managed objects, based on the particular business task at hand. This might even include relationships between managed objects of different types (for example, PolicyHolder and Beneficiary). This can be done using an IManagedReference Collection, as shown in the following Visual Basic code segment:

```
Dim mixedCollection as Object
Set mixedCollection = CreateObject("IDL:IManagedCollections.IReferenceCollection")

' Find a collection in the name space which contains PolicyHolders and Beneficiaries

' Create an iterator on the reference collection that was found above. When an iterator is
' created, it is automatically positioned preceeding the first element.
Dim anIterator as Object
Set anIterator = mixedCollection.createIterator()

Dim element as Object
Set element = CreateObject("IDL:IManagedClient.IManageable")

' Loop through the collection. The "next" method advances the iterator
' to the next element (on the first invocation, this will advance to the
' first element). If the iterator is now past the end of the collection,
' the "next" method returns FALSE; otherwise, returns TRUE and sets the output
' parameter to the element at which it is now positioned.
```

```
Do While anIterator.next(element)
   If element.is_A(PolicyHolder) Then
      ' Send him a bill
   End If
   If element.is_A(Beneficiary) Then
      ' Send him a check
   End If
Loop
```

The combination of the IManagedCollections::IReferenceCollection and the IManagedCollections::IIterator were used in the previous code segment. An IManagedCollections::IReferenceCollection is a generalized collection of object references that is iteratable. IManagedCollections::Iterator supports advancement of the iterator and retrieval of elements by the next() method. IManagedCollections::IReferenceCollection supports adding and removing elements using the addElement() and the removeElement() methods. IManagedCollections::IManagedReferenceCollection is the most basic kind of collection supported in Component Broker. Combining this with the capabilities of IHome provides the basis for writing simple applications and the foundations for the more advanced query and collections capabilities provided by Component Broker. For more information on collections and query, see Chapter 6, “MOFW Client Programming Model – Advanced Concepts” on page 83.

Remembering your Favorite Objects

Component Broker allows you to remember a managed object instance, by introducing the concept of an object reference. An object reference is opaque and you cannot set its internal structure. However, a reference always and uniquely refers to a managed object regardless of where it resides in the network.

Continuing the example, assume that you perform the following Visual Basic code segment.

```
' This example just creates the string containing the object reference
' This string could be written to a file using normal VB conventions if
' desired

' Get a "string" version of my reference to Robert
' robert points to Robert or a proxy to Robert

Dim corbaFactory as Object
Set corbaFactory = CreateObject("CORBA.Factory")

Dim orb as Object
Set orb = corbaFactory.GetObject("CORBA.ORB.2") ' Get access to the orb

Dim robertStringifiedReference as String
robertStringifiedReference = orb.ObjectToString(robert)

' Save the string to a file using normal VB file IO

' I do not need Robert anymore
set robert = nothing
```

If you save a reference to Robert as a Stringified Object Reference, then you can use this string to re-access Robert at a later time. The following Visual Basic code segment presents an example of re-accessing the Robert object.

```
' Get back the string from the file I saved earlier into
' robertStringifiedReference

' Make an object reference for Robert
Dim tempObj as Object
Dim robert as Object = CreateObject("IDL:Robert")
Set tempObj    = orb.StringToObject(robertStringifiedReference)
robert.narrow tempObj

' I can now work with Robert.
robert.Name()
```

When References Explode

Another application may call the remove() method on objects referenced by a client application. Component Broker supports a Concurrency Control Service to regulate sharing of objects. If a client does not lock an object, it can be deleted. Language environments like ActiveX prevent this from happening in simple, single process applications, but achieving the same level of function is impossible in a cross-system, multi-language environment. If you are operationally reusing data and applications, some legacy code not under the object-oriented application's control can delete the instance data for objects. An object to which you have a reference can vanish unless you use Concurrency Control.

Therefore, if you do not use Concurrency Control, you need to be prepared for exceptions. If an invalid object reference is used, an exception is thrown. While not perfect, this is a substantial improvement over the semantics of languages like C++ that can result in unpredictable behavior if invalid references are used.

Chapter 9. MOFW - Java Client Programming Model

This chapter deals with the specifics of using a Java Client to access managed objects on a Component Broker server. It follows the format laid out in omitting the lengthy discussion of abstract concepts, and instead dealing with the specifics of developing code using managed object proxies in Java. Read Chapter 4, "MOFW Client Programming Model" on page 33 before you use this chapter.

Throughout this chapter, there are references to one of the components of the solution as the idl.toJava (com.ibm.idl.toJava.Compile is the full name) compiler. IBM's compiler is based on Java.

Note:

> To allow Component Broker to use multiple character sets, you must use only the Portable Character Set supported by the seven-bit ASCII code set when developing client and server code.
>
> The Portable Character Set supported by Component Broker follows:

```
0 1 2 3 4 5 6 7 8 9
: ; < = > ? @ [ \ ] ^ _  '  { | } ! " # $ % & ( ) * + , - . / <space>
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 Y Z
```

Java Client View of Component Broker Applications

The development view of the managed object (MO) proxies that are used on the client platform starts on the server. When developers are programming managed objects for Component Broker, they are obliged to create IDL files that represent those managed objects. Typically, the Component Broker developers use the Object Builder to help them coordinate these files, but that is not required.

The key, for a Java client developer, is to run the IDL for the managed objects through the idl.toJava compiler. The idl.toJava compiler takes the interface definitions in the idl, and produces the managed object proxies that are used from the client. The proxies are produced in the form of .java files, that need to be compiled and placed either in a location where the Web server has access for applets, or on the client for applications.

As in Figure 48 on page 158, there are two routes to producing the .java files for the managed object proxies. Whether you process the IDL files generated by the Object Builder tool, or process IDL files that you wrote, the results should be the same.

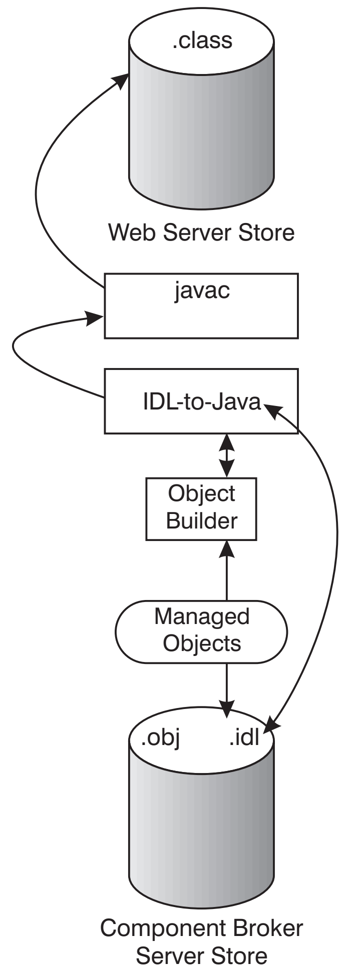


Figure 48. Java Client View

When the .class files for the managed object proxy are available, the proxy class can be treated as any other Java .class, with the exception of the specific requirements added to the class, based on its involvement with the CORBA-based Component Broker server. These are explained in this chapter. Look at the run-time elements of the solution.

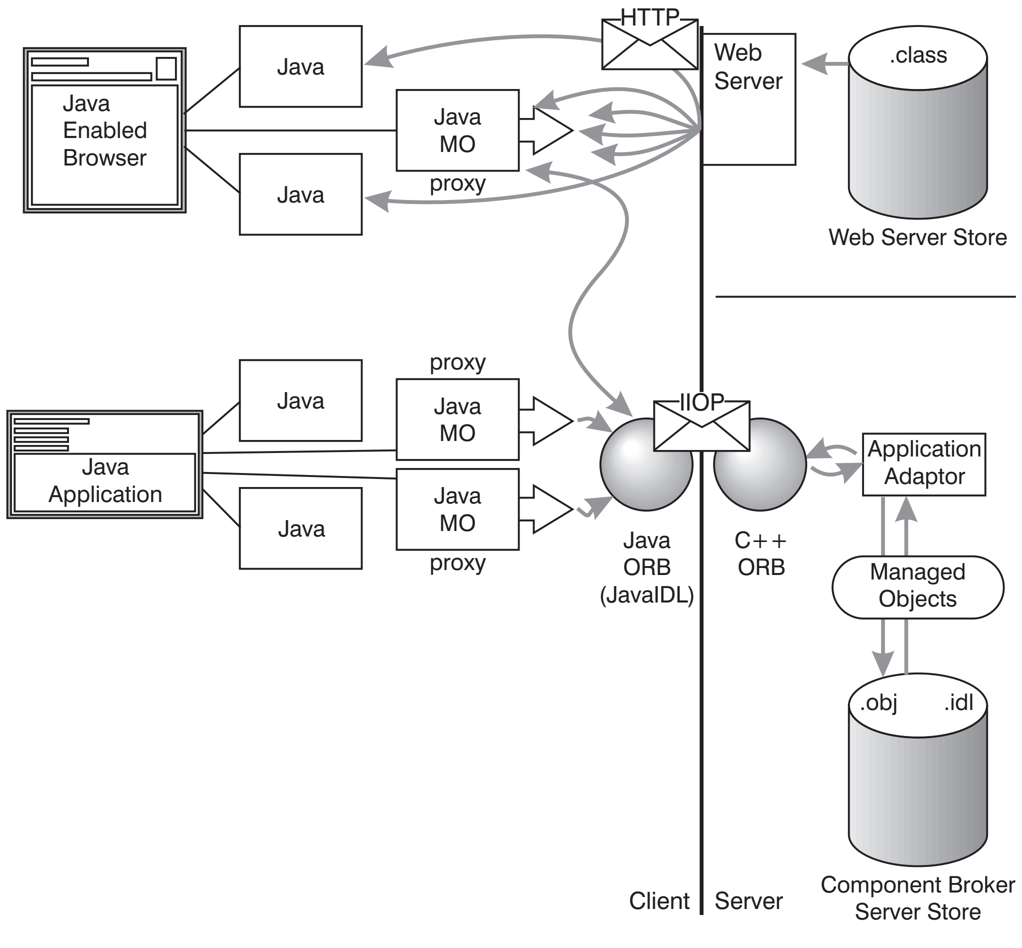


Figure 49. Java Client Run-time Scenario

A run-time scenario for a Java client begins when either the applet, or the application containing the managed object proxy is accessed. In either case, the first order of business is to connect to the ORB, and establish a naming context. From there, the issues of finding and using the managed objects through the proxy are really issues related to how the Component Broker programing model is set up. Only the syntax changes for the Java environment.

Since only one Object Request Broker (ORB) is started per Java Virtual Machine (JVM), the ORB will be shared among all applets running in the JVM. Applets intended to run with other applets in the same Java Virtual Machine should be designed to efficiently and safely use the shared ORB.

Client Programming Model: Basic Tasks

Chapter 4, “MOFW Client Programming Model” on page 33 describes the following tasks that a client is likely to want to do:

- Find an object.
- Use an object.
- Create or delete an object.
- Use a set of objects.
- Remember an object.

This chapter explores the same kinds of tasks as in Chapter 4, “MOFW Client Programming Model” on page 33, with the same examples, but presents them from the unique perspective of a Java client developer. There is, however, one additional step that is discussed prior to getting into the programming model for the client.

Preparing to Use VisualAge for Java for Development of Component Broker Java Clients

Before using VisualAge for Java for development of Component Broker Java clients, it is necessary to import the IBM Java ORB into a VisualAge for Java project. Do this using the following steps:

1. Start VisualAge for Java.
2. Insert your Component Broker Client CD into your CD drive.
3. From the VisualAge for Java Workbench, select File → Import.
4. Select “Jar file” as the input source.
5. Browse to your CD drive and navigate to the \jclient directory.
6. Select the file somojor.zip.
7. Select .class and resource as the file types to import.
8. Enter the name of a new project or browse to an existing one.
9. Click the **Finish** button.

Since somojor.zip is a large file, it may take a few minutes for the importation process to complete.

Preparing Managed Objects for Remote Access

The client programming model is based on the idea of having remote proxies available for managed objects on the server which are represented in IDL files. Depending on how managed objects are created in your organization, you may not be the one who performs this step. Even if that is the case, it is helpful to know what is going on.

The first thing that you need to do is to make sure that the proxy is available to the client. Given that there is an IDL definition for the managed object, there are two ways to generate the Java proxy for the client. The first way to generate a proxy is by instructing Object Builder that you would like a Java client proxy for the managed object you are dealing with. The second way to generate the Java client proxies is to use the idl.toJava compiler directly. The syntax for the idl.toJava compiler is:

```
java com.ibm.idl.toJava.Compile [options] idl file
```

where *idl file* is the name of a file containing IDL definitions, and *[options]* is any combination of the following options. The options may appear in any order; *idl file* is required and must appear last.

Options:

-bean	Generate classes that can be used as Java beans. By default, the client stubs are not beans.
-d *symbol*	This is equivalent to the following line in an IDL file: `#define` *symbol*.
-emitAll	Emit all types, including those found in #included files. By default, only those types found in *idl file* are emitted.
-f*side*	Define what bindings to emit. *side* is one of the following: client, server, all, serverTIE, or allTIE. serverTIE and allTIE cause delegate model skeletons to be emitted. If this flag is not used, -fclient is assumed.

-i *include path*
: By default, the current directory is scanned for included files. This option adds another directory.

-keep If a file to be generated already exists, do not overwrite it; by default it is overwritten.

-m Generate information to be included in a make description file; the output goes to a .u file. By default, this information is not generated.

-pkgPrefix *t pkg*
: Wherever the type or module *t* is encountered, ensure it resides within *pkg* in all generated files. *t* is a fully-qualified Java-style name.

-sep *string*
: This option is only valid with the -m option. Replace the file separator character with *string* in the file names listed in the .u file.

-stateful **Warning**: Non-standard IDL! Parse stateful interface objects (used for objects-by-value).

-td *target_directory*
: Emit bindings to *target directory* rather than to the current directory.

-v Verbose mode. By default, unless there are errors, no messages are output.

In most cases, the invocation of this compiler is straightforward. Regardless of which method you use, the tools should generate several Java proxy files. The rest of this chapter describes how to use these files.

Depending on whether you intend to use the proxies in an application, or in an applet served up by a Web server, you have to decide where to put, or where to tell the tools to put, the proxies. When you have made these decisions, and generated the proxies, you can start developing the code that uses the proxies for the managed objects on the server.

Initializing the Component Broker Client Environment

The CBSeriesGlobal interface is provided as a convenience. A Component Broker client could make the same set of calls as is encapsulated in the Initialize() method independently, and in some cases, this allows more flexibility. To use the CBSeriesGlobal interface, you must include the following line:

```
import com.ibm.CBUtil.CBSeriesGlobal
```

The CBSeriesGlobal interface provides a number of different Initialize methods depending on your needs as a Java *applet* or application. All of these Initialize methods encapsulate the calls to initialize the ORB and get an initial reference to the Name Service. Therefore, initializing a Component Broker Java *application* client requires a single line of code, one of the following:

```
CBSeriesGlobal.Initialize(host);
CBSeriesGlobal.Initialize(host, port);
CBSeriesGlobal.Initialize(arguments);
CBSeriesGlobal.Initialize(arguments, properties);
```

To initialize a Component Broker Java *applet* client also requires a single line of code:

```
CBSeriesGlobal.Initialize(applet);
CBSeriesGlobal.Initialize(applet, properties);
```

Only after the client has called the Initialize method, can they use the static methods *orb()* and *nameService()*. Subsequent sections discuss how the client might use these methods.

The CBSeriesGlobal interface encapsulates the call to initialize the ORB, which appears in CBSeriesGlobal as:

```
java.util.Properties props = new java.util.Properties();
props.put("org.omg.CORBA.ORBClass", "COM.ibm.CORBA.iiop.ORB");
orb = (COM.ibm.CORBA.iiop.ORB) ORB.init (naught, props);
```

Where *naught* is a null array of strings, which could be the command line arguments passed in to initialize the ORB. *props* is a properties object filled in with the host name and port of the Bootstrap server to which the client should connect. The Bootstrap server is installed as part of the Component Broker server. If you are unsure of the host and port for this Bootstrap server, contact your Component Broker system administrator. The property names for the Bootstrap server's host and port are com.ibm.CORBA.BootstrapHost and com.ibm.CORBA.BootstrapPort. These properties would be added to the properties object in the following manner:

```
props.put("com.ibm.CORBA.BootstrapHost", host);
props.put("com.ibm.CORBA.BootstrapPort", port);
```

Where host and port are Java String objects containing the values for the Bootstrap server's host and port. In addition to host and port, to avoid conflicts with other Java ORBs, a property is added to the property list to make a direct reference to the Component Broker Java ORB. This prevents problems with inadvertently using other ORBs. This problem can occur most often when using Java *applets* in Web browsers. To avoid the problem, modify the code for your Java *applet* to include an explicit cast to the Component Broker Java ORB, com.ibm.CORBA.iiop.ORB.

Within CBSeriesGlobal, the Naming Service is resolved in the following manner:

```
obj = fOrb.resolve_initial_references("NameService");
fNameService = NamingContextHelper.narrow(obj);
```

As you can tell, there are many ways to use these interfaces. The Initialize() methods allow the user to use CBSeriesGlobal to eliminate concern over such intricacies.

CBSeriesGlobal is a convenience interface that is not required for all client programs. However, if the client program uses either a CopyHelper or PrimaryKey that contains an object as one of its attributes then initializing CBSeriesGlobal is a requirement. This is because the implementation of the CopyHelper and PrimaryKey depend on CBSeriesGlobal:orb() when using the ORB *object_to_string()* operation.

IExtendedNaming is used to make things go a little easier. If you need a refresher on it, refer to References in the Component Broker Online Documentation.

Finding a Managed Object

You can find an object in one of two ways. The object might have a Name and the client can use the Component Broker Naming Service to look up the object by its Name. In general, only a small subset of the object instances in a distributed system are in the Naming Service. These are typically large, well-known objects such as collections of business objects or important object instances in the Object Model.

The second technique for finding an object is to use the Naming Service to find a well-known object, for example a collection, and then navigate to the desired object from the well-known object. Navigation occurs by looking in collections or following references to other objects.

Finding a Well-known Object Using the Naming Service

Assume that the insurance company in the example placed several Claim objects in the Naming Service. The following code segment shows how to find such a Claim, belonging to a customer named Lou.

```
Claim louClaim = Claim.narrow(
   CBSeriesGlobal.nameService().resolve_with_string(
      ".:/Applications/LifeInsurance/Claim/LouClaim") );
```

The Component Broker client and server run times initialize the global object instance nameService to refer to the root of the installation's Naming Service.

For a refresher on determining the naming context, and details on how the name space is specified, refer to Chapter 4, “MOFW Client Programming Model” on page 33.

Recall that in addition to the resolve_with_string() method, Naming Contexts also support the bind_with_string() method which associates a name with an object instance. The following code segment could have been used to name the Lou Claim object.

```
// Assume all of the appropriate imports have been done.
// Do a bunch of stuff with a reference to Lou's Claim

Claim louClaim; // declared somewhere prior to the next set of code
louClaim.
  ...
// Get a reference to the Life Insurance Application's
// Claim Naming Context using IBM's Extended Naming
// (com.ibm.IExtendedNaming is syntactically optional)

com.ibm.IExtendedNaming.NamingContext nc;

// getting the naming context for Claim Objects

nc = com.ibm.IExtendedNaming.NamingContextHelper.narrow(
     CBSeriesGlobal.nameService().resolve_with_string(
        "/.:/Applications/LifeInsurance/Claim") );

// Add Lou to the Name Space

nc.bind_with_string("LouClaim",louClaim);
```

Finding an Object Using Collections and Navigation

When you have an object, you can use its methods to find related objects. Continuing the previous example, after finding Lou's Claim, you can find other objects that the Claim references. The following example gets you a reference to Lou's Policy:

```
// Find Lou's Policy

Policy louPolicy; // declare a local variable
louPolicy = louClaim.policy();
```

The example is simplified in that LouClaim is in the Naming Service. In the event that you do not have a well-known name for the claim, how do you find a specific claim? Homes are instances of the IHome class. The managed object provider may decide to implement and provide a tailored subclass of IHome, or might use an instance of the base class. The relationship between managed objects and collections is explained in “Using Sets of Objects” on page 167.

You can find the Home for Claim objects by using the following code segment:

```
import com.ibm.IManagedClient.* ;

IHome claimHome;
claimHome = IHomeHelper.narrow( CBSeriesGlobal.nameService().resolve_with_string(
        "/.:/Applications/LifeInsurance/Homes/Claim") );
```

Now you need to find Lou's Claim. If you know the Claim number, all you need is the Primary Key Helper class for the Claim. Every managed object class has a set of local helper classes that let you use its keys. An instance of a Key Helper Class is always local to the client's process and language.

The provider of the managed object that you are working with has given you access to source for, or actual, .class files for the helper class. Regardless, it is up to you to ensure that you have access to them and can use them in your applications.

Key Helpers, like all helper classes are created with a static method on the class named _create(). This static method gets generated by the bindings that accompany all subclasses of ILocalOnly. The same rule is in place for copy helpers and objects of other classes.

Having created an instance of a Primary Key, the key must be set by one or more attributes on the Primary Key object. When all of the key attributes have been set, the Primary Key object is now usable. The Claim Home uses this Primary Key to find the previously created Claim object. Remember, the PrimaryKey is on the client system, but the Claim object and the Claim Home or on the server system. If the client passes a Primary Key object as a parameter to the Home, and the Home is on a remote system, the remote system might get a proxy back to the client's PrimaryKey instance. This would turn the client into a server and unpredictable results could occur. Therefore, the Component Broker programming model uses strings as the method for passing keys to potentially remote objects.

Continuing the example, the following code segment finds Lou's Claim in the Home (assuming his number is 1234).

```
import <package>.ClaimPrimaryKeyHelper;
// Package containing ClaimPrimaryKey and Helper;
// Create an instance of the Key helper Class
ClaimPrimaryKey claimPrimaryKey = ClaimPrimaryKeyHelper._create();

// Set the claimNo attribute in the key
claimPrimaryKey.setClaimNo(1234);

// Must get data out of the Stream to go onto the wire to the server
String claimString = claimPrimaryKey._toString();

// Call find by key on the Home to find Lou's Claim
Claim louClaim = ClaimHelper.narrow(claimHome.findByPrimaryKeyString(claimString) ) ;
```

The object provider of a public managed object always provides you with a set of helper classes for using the Homes that contain his managed objects. There is always exactly one Primary Key helper class. The object provider gives a client developer:

- Interface definitions for the Key Classes. In addition to the Primary Key, there may be Secondary Key Helper Classes. A Secondary Key may also uniquely identify an object, or multiple instances may have the same value.
- Implementation of the Key Classes.
- Documentation for their use.

The Component Broker programming model mandates Key Helper Classes for managed objects. The helper classes make things easier for you. The keys are passed as strings. You only need to use the available set*xxxx*() methods on the Helper Class to prepare the key information. String manipulation to concatenate pieces of multi-valued keys is not necessary. Using a Key Helper Class enables type errors to be detected at compilation time, and alleviates the need for you to know the field ordering and algorithm for defining a key.

A Note on Security

Setting the client up to work with a secure server is primarily a configuration and administration issue. For additional information, see Security under Concepts in the Component Broker Online Documentation.

The client programming implications of security are fairly limited. Although there is much that you can do with security, there is little that you must do. The Concepts and Procedures sections under Advanced Topics in the Component Broker Online Documentation describe many of the features available to someone needed to handle complex security issues. For the most part, however, you need to know that any request to a Component Broker object on the server can be denied if the client does not have the necessary security tokens.

Using a Managed Object

When you have found a reference to a managed object, you can use it by invoking methods on it. For example:

```
person.setName("Lou Smith");
```

This example calls the setName() method on the person object identified by *person*. The Component Broker internal implementation handles the use of remote objects.

Creating a New Object

Component Broker managed objects can be created in a number of ways. The following sections describe the default ways to easily create managed objects.

Create From Key

Every Component Broker managed object class has an instance of a Factory associated with it. The Factory provides a set of interfaces for creating instances of a managed object. The Factory gets some of its interface from the base class CosLifeCycle.GenericFactory. The createFromPrimaryKeyString() method is introduced in the IManagedClient.IHome interface supplied by Component Broker. This interface specializes the COSLifeCycle.GenericFactory interface and plays the role of factory for Component Broker managed objects. Object providers may implement and provide a tailored subclass of this interface, or they might use the implementation of IHome provided.

You need to know how to find the right IHome for creation. Homes are at well-known locations in the Naming Service. The input required for the factory finder is the name of the implementation class that this factory makes instances of. The following code fragment gets a reference to the Claim Factory for the Life Insurance Application.

```
import com.ibm.IExtendedLifeCycle.*;
import org.omg.CosLifeCycle.;

// (com.ibm.IExtendedLifeCycle is syntactically optional)
com.ibm.IExtendedLifeCycle.FactoryFinder myFinder;

myFinder = (com.ibm.IExtendedLifeCycle.FactoryFinderHelper.narrow(
       CBSeriesGlobal.nameService().resolve_with_string(
         "/.:/Applications/LifeInsurance/FactoryFinders")));
IHome claimHome = IHomeHelper.narrow(myFinder.find_factory_from_string(
         "Claim.object interface"));
```

You need to provide the IHome with information necessary to manufacture a new object instance. At a minimum, the Primary Key must be provided. An example of creating a new Claim with a claimNo of 1234 is:

```
// Create an instance of the Primary Key Class
ClaimPrimaryKey claimKey = ClaimPrimaryKeyHelper._create();

// Set the claimNo attribute in the key
claimKey.setClaimNo(1234);

// Call createFromKey on the Factory to create Lou's Claim
Claim theClaim;
theClaim = ClaimHelper.narrow( claimHome.createFromPrimaryKeyString(claimKey._toString()) );
```

This example is almost identical to the previous example. First, create a Primary Key Object to define the identity of the object to be made. Then, call createFromPrimaryKeyString on the Home to pass the Key.

The createFromPrimaryKeyString method is defined by the IHome class, and all business objects can be created by this method. An object provider might provide you with a subclass that introduces other, easier to use creation methods. Additional create methods are described, with examples in Chapter 6, “MOFW Client Programming Model – Advanced Concepts” on page 83.

Create from Copy

Setting and getting the attributes of a managed object can be expensive. There are two main reasons for this. First, if the managed object is implemented in another language, each get or set method is actually a cross-language call. Cross-language calls are more expensive than simple, same language calls. The get and set overhead is even worse if the managed object is remote because each call is actually a remote procedure call and involves significant overhead. Consider the following code segment:

```
// Assume that 'theClaim' and 'theClaimHome' are declared in the previous code
// segment, and that 'theClaimHome' is located as in the previous code segment.
// Create a new Claim with a Home generated claimNo
theClaim = theClaimHome.create();

// Now initialize the Claim's attributes
theClaim.setDate("10/14/96");
theClaim.setState(entered);
```

This fragment could involve the following remote method calls:

- The client to Claim Home (specialized IHome) to create the Claim.
- The client to Claim managed object to set its *date* attribute.
- The client to Claim managed object to set its *state* attribute.

An object provider must provide you with a Copy Helper Class for use with the createFromCopy() method on an IHome. The following code segment rewrites the previous example using this design pattern.

```
import <packagename>.ClaimCopyHelper;
// Package with ClaimCopy and Helper;
// Create a new "local" Claim in my process and language.
// Use a Copy Helper Class that the Claim MO provider gave me.
ClaimCopy claimCopy = ClaimCopyHelper._create();

// Now initialize the Claim's attributes.
claimCopy.setDate("10/14/96");
claimCopy.setState(entered);

// Pass this local copy to the Home and have him return a new Claim
// MO whose attributes are initialized from the local copy's values.
// Since not all ORBs support Pass-By-Value, we first convert to a string.
Claim theClaim = ClaimHelper.narrow(claimHome.createFromCopyString(claimCopy._toString()) );
```

Like a Key Helper Class, a Copy Helper Class instance is always local to your process and implemented in the language you are using. The object provider gives you the helper class interface and implementation.

Copy Helper Classes are especially useful if the client application needs to interact with an object during initialization, and then create a managed object from the attributes. A common scenario for this is entering data for the object from a GUI. The GUI updates the local Copy Helper Object, and then createFromCopy() is called when the **Do** button is pushed on the end user interface (EUI).

Releasing and Deleting Objects

Eventually, you no longer need to use an object that you created or found. Component Broker supports two interpretations on “no longer needs.”

- The remove() method deletes the object and its instance data.
- The release() method informs the object that the client application no longer plans to reference the object. The object still exists in the server, and other applications may be using it, but the client calling the release() method is done with the object

Using Sets of Objects

An IHome represents a set of managed objects, all of the same type, whose relationship to one another is defined by the object provider, and maintained by the fact that they were all created in the same home. Sometimes an application needs to define and manage the relationships between managed objects, based on the particular business task at hand. This might even include relationships between managed objects of different types (that is, PolicyHolder and Beneficiary). This can be done using an IManagedReference Collection, as shown in the following code segment:

```
import com.ibm.IManagedCollections.*
com.ibm.IManagedCollections.IReferenceCollection mixedCollection;

// Find a collection in the name space which contains PolicyHolders and Beneficiaries
   ...
// Create an iterator on the reference collection that was found above. When
// an iterator is created, it is automatically positioned preceding the first element.

IManagedCollections.IIterator anIterator = mixedCollection.createIterator();
```

```
IManagedClient.IManageable element;
// Loop through the collection. The "next" method advances the iterator to the next
// element (on the first invocation, this advances to the first element). If the
// iterator is now past the end of the collection, the "next" method returns NULL.;
// otherwise, the "next" method returns the element the iterator
// at which it is now positioned.

while ( anIterator.next(element) )
{
   if ( element.is_A(PolicyHolder) ) // Send a bill
   if ( element.is_A(Beneficiary) )  // Send a check
}
```

The combination of the IManagedCollections.IReferenceCollection and the IManagedCollections.IIterator are used in the previous code segment. An IManagedCollections.IReferenceCollection is a generalized collection of object references that is iteratable. IManagedCollections.Iterator supports advancement of the iterator and retrieval of elements by the next() method. IManagedCollections.IReferenceCollection supports adding and removing elements by addElement() and removeElement(). IManagedCollections.IManagedReferenceCollection is the most basic kind of collection supported in Component Broker. Combining this with the capabilities of IHome provides the basis for writing simple applications and the foundations for the more advanced query and collections capabilities provided by Component Broker. For more information on collections and query, see Chapter 6, “MOFW Client Programming Model – Advanced Concepts” on page 83.

Remembering your Favorite Objects

The Component Broker allows you to remember an interesting or important managed object instance by introducing the concept of an object reference. An object reference is opaque and you cannot set its internal structure. However, a reference always and uniquely refers to a managed object regardless of where it resides in the network.

Object references are available to the Java programmer without regard to what kind of Java programming you are doing. Applet restrictions on the use of the file system, however, render this feature unusable for the applet programmer. The following example assumes that the work is being done from an application.

Continuing the example, assume that you run the following code segment.

```
File output_file = new File("SOMEFILE.DAT");
FileOutputStream output = new FileOutputStream(output_file);

// Get a "string" version of my reference to Robert
// robert points to Robert or a proxy to Robert
String robertStringifiedReference = CBSeriesGlobal.orb.object_to_string(robert)

// Save the string to a file using a "pseudo" file routing
output.write(robertStringifiedReference);
output.close();

// I do not need Robert anymore
robert.release();
```

You saved a reference to Robert as a stringified object reference, and can use this string to re-access Robert at a later time. The following code segment presents an example of re-accessing the Robert object.

```
String infile = "SOMEFILE.DAT";
File input_file = new File(infile);
FileInputStream input = new FileInputStream(input_file);

// Get back the string I saved

String robertStringifiedReference;
input.read(robertStringifiedReference); // returns the bytes read, or -1,
                   // if you'd like to check that.

// Make an object reference for Robert
robert = CBSeriesGlobal.orb.string_to_object(robertStringifiedReference);

// I can now work with Robert.
robert.setName();
```

When References Explode

Concurrency control is not presently extended to the client proxies. You need to be prepared for exceptions. If an invalid object reference is used, an exception is thrown.

Java Exception Handling

The Java exception handling model has some slight differences from the C++ exception handling model. Within C++ a method can throw two types of exceptions, either exceptions that are explicitly declared on the “raises” clause of the method or any of the CORBA standard exceptions.

In the Java Programming Language, Java exceptions are primarily checked exceptions, meaning the compiler checks your method only throws exceptions that have been declared as throwable in the “raises” clause. These classes extend the java.lang.Exception class. Other run time exceptions and errors extend the classes RuntimeException and Error, which are unchecked exceptions, meaning (much like the CORBA Standard exceptions) they can be thrown from any method.

Exceptions such as IDataKeyAlreadyExists and IDataKeyNotFound are checked exceptions while other exceptions, such as IDataObjectFailed extends java.lang.RuntimeException and are unchecked exceptions and can be thrown from any method.

When doing exception handling while using Java, be aware that exceptions other than those listed on the “raises” clause of methods can be thrown by any method at any time. These unchecked exceptions typically occur as a result of some drastic run time error.

Chapter 10. Java Server Programming Model

This chapter describes how to use Java to implement managed objects that run in a Component Broker server.

Some aspects of server Java programming were introduced in Chapter 9, “MOFW - Java Client Programming Model” on page 157. Managed objects access the CBSeriesGlobal object, Object Services, and other managed objects just like client programs. This could be called the client aspect of server programming, because the managed object acts as a client to the service or to another object. In Component Broker, the client aspect of server Java programming is almost identical to programming in a Java client. However, when generating Java classes from IDL, the IDLC compiler is used for server Java but the idl.toJava compiler is used for the client.

Server Java also introduces some new constructs and procedures that have no counterparts in client programming. Implementing a business object, adapting it to the server execution environment, and managing its persistent data are all tasks that are specific to the server environment. This chapter deals with those server-specific issues.

Overview of Java Managed Object Development

Component Broker lets you use Java to implement managed objects by performing the following steps. The steps involved are the same as those listed in Chapter 5, “MOFW Server Programming Model” on page 57; differences are discussed in this chapter:

1. Develop an interface to the business object.
2. Choose a pattern for handling essential state.
3. Implement the business object methods.
4. Implement the methods required by the MOFW interfaces.
5. Implement the necessary primary key class.
6. Implement the optional copy helper class.

After performing these steps, you will be ready to unit test the Java business object. Unit testing of Java is described in “Unit Test for Java Business Objects” on page 347.

After unit testing, you will be ready to package the Java business object for running on a server (for information about packaging the Java business object, see “Assembling and Installing Java Business Objects” on page 261). This procedure changes only slightly from the non-Java procedures described in Chapter 11, “Assembling and Installing Business Objects on AIX and Windows NT” on page 197, and mainly involves using the Component Broker tools to generate and compile C++ code to connect the business object into the server run-time environment.

Figure 50 on page 172 shows how the tools are used to construct the pieces of a server-installable Java business object.

The top part of the figure represents the activities described in this chapter. The Object Builder tool and the IDLC compiler are used to create the Java source code for the business object (_PolicyBOBase.java in the figure), plus a collection of other related Java source files. Included here are the Java Key and Copy helper classes required by Java clients of the business object.

Next the Java business logic is tested using the Java unit test environment. This permits exercising the Java code in a mock-up run-time environment outside a server. Because all the code involved at this point

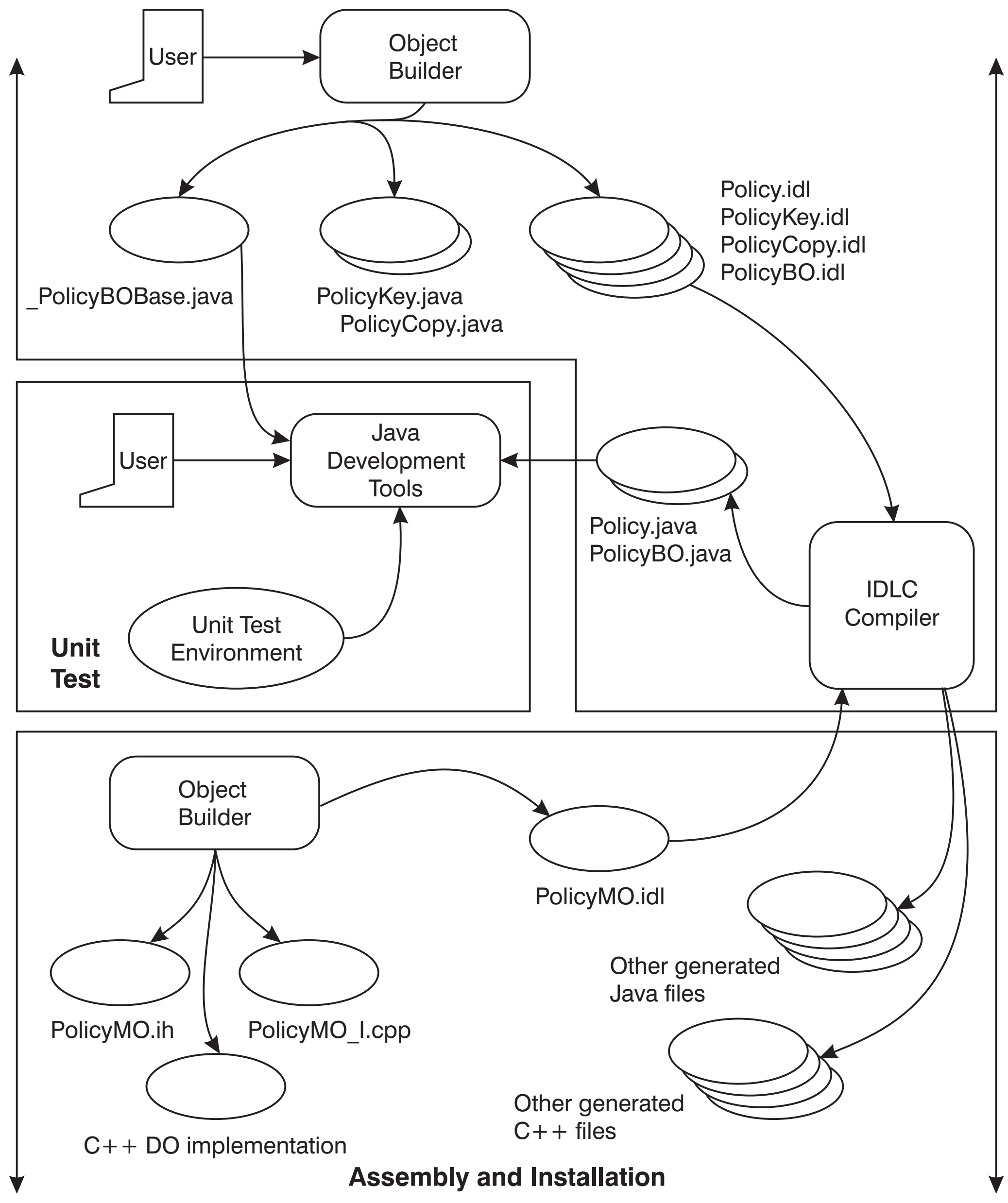


Figure 50. Using Tools to Construct a Server-installable Java Business Object

is Java, you can use any Java development tools to complete, test, and debug your code. For more information, see “Unit Test for Java Business Objects” on page 347.

Finally, you return to the Object Builder tool to generate other C++ and Java code that adapts the business object to the server environment and connect it to your chosen persistent stores, if any. For more information, see “Assembling and Installing Java Business Objects” on page 261.

Developing an Interface to the Business Object

The procedures and tips in Chapter 5, "MOFW Server Programming Model" on page 57 apply equally well to the development of Java business objects. The interface for a business object is defined in a way that is independent of the language used to implement it and yields an IDL interface definition. As before, the interface should inherit from the IManagedClient::IManageable interface. If you follow the module scoping advice from Chapter 5, "MOFW Server Programming Model" on page 57, the Policy interface might look similar to the following segment:

```
#include <IManagedClient.idl>
#include "Beneficiary.idl"

module XYZCompanyInsurance
{
   exception InvalidAmount {};

   interface Policy : IManagedClient::IManageable
   {
      void addBeneficiary(in Beneficiary benRef);
      void delBeneficiary(in Beneficiary benRef);

      readonly attribute long policyNo;       // Primary key

      void changeAmount(in float newAmount) raises (InvalidAmount);
      float getAmount();

      attribute string comment;
      readonly attribute float premium;
   };
};
```

This IDL is mapped into Java according to the CORBA IDL-Java mapping specification when you run the IDLC compiler with the `-suj` option, as follows:

```
idlc -suj Policy.idl
```

The `uj` in the example stands for user Java, because it tells IDLC to generate the Java classes and interfaces needed for a client to use the declared IDL interfaces and types. Even though they are called the user (or client) bindings, these Java artifacts are also needed on the server. For the sample Policy.idl, the generated files are:

- Policy.java
- PolicyHolder.java
- PolicyHelper.java
- _PolicyStub.java
- InvalidAmount.java
- XYZCompany/InvalidAmountHelper.java

See the IOM (Interlanguage Object Model) online documentation for more information about these files. See "Assembling and Installing Java Business Objects" on page 261 for information about the `-sbj` option that generates additional business-object Java used only on the server.

The IDL exception becomes a Java final class extending a standard CORBA class for user-defined exceptions:

```
package XYZCompanyInsurance;

public final class InvalidAmount extends org.omg.CORBA.UserException
{
   public InvalidAmount() {};
}
```

The UserException class, in turn, extends java.lang.Exception, and so its subclasses can all be used in a Java throw statement, similar to:

```
throw new XYZCompanyInsurance.InvalidAmount();
```

This example also illustrates the following rules for Java package names:

- CORBA-defined classes and interfaces, like UserException above, are in the org.omg.CORBA package.
- IBM-defined extensions all appear in packages whose names begin com.ibm.
- User-defined types and interfaces appear in a Java package that is the name of the enclosing module.

These rules are illustrated again in the Java interface created from the IDL Policy interface:

```
package XYZCompanyInsurance;

public interface Policy extends com.ibm.IManagedClient.IManageable
{

   void addBeneficiary(Beneficiary benRef);

   void delBeneficiary(Beneficiary benRef);

   /**
    * Getter method for attribute "policyNo"
    */
   int policyNo();

   void changeAmount(float newAmount) throws XYZCompany.InvalidAmount;

   float getAmount();

   /**
    * Getter method for attribute "comment"
    */
   java.lang.String comment();

   /**
    * Setter method for attribute "comment"
    */
   void comment(java.lang.String comment);

   /**
    * Getter method for attribute "premium"
    */
   float premium();

}
```

The mapping of names and types is according to the CORBA specification of which the following table is a brief summary:

1. Names that conflict with Java keywords have a single underscore prepended to them. The affected names are:

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

2. IDL names ending with the suffixes `Helper`, `Holder`, or `Package`, when used to define an IDL type or interface, also have an underscore prepended in Java. Therefore, the following IDL segment:

   ```
   module XYZCompanyInsurance {
       interface PolicyHelper {
   ```

   Yields the following Java:

   ```
   package XYZCompanyInsurance;
       public interface _PolicyHelper extends com.ibm.IManagedClient.IManageable
   ```

3. IDL operation names that are the same as methods of the class java.lang.Object also have underscores prepended. These names are:

clone	finalize	hashcode	notifyAll	wait
equals	getClass	notify	toString	

4. Types are mapped according to the following table:

IDL Type	**Java Type**	**IDL Type**	**Java Type**
boolean	boolean	long unsigned long	int int
char	char	float	float
wchar	char	double	double
octet	byte	array	array
string	java.lang.String	sequence	array
wstring	java.lang.String	void	void
short	short	any	org.omg.CORBA.Any
unsigned short	short	Object	org.omg.CORBA.Object

Note: org.omg.CORBA.Any is an abstract class and cannot be instantiated. Each Java-CORBA ORB provides an implementation of Any. To create an instance in a Component Broker environment, use:

```
org.omg.CORBA.Any any =
        com.ibm.CBCUtil.CBSeriesGlobal.orb().create_any();
```

Loading C++ DLLs from Java BO

To access a managed object that may be in an application installed on another host in the network, the Java BO programmer must explicitly load the C++ DLL to access the managed object's C++ components. To do this in Object Builder, define a private static method on the BO, and call it something like libLoad, and have it return a boolean. Then, define a private static attribute, say libLoaded, and invoke the libLoad method as it's initializer.

The content of the libLoad method needs to invoke the Java System.load function to explicitly specify the path and DLL name to be loaded. Unfortunately, this forces a tight coupling of the Java BO implementation and the directory in which the application is installed. For example, the code generated by Object Builder will look something like this:

```
static private boolean iLibLoaded = libload();
static private boolean libload()
{
   //Version identifier DCE:12457A40-339B-11d2-B197-0004ACEA9E5A:1
   // Insert Method modifications here
   boolean retval = false;
   try
   {
      System.load("d:\\ntapps\\PolicyApp\\bin\\PolicyC");
      retval = true;
   }
   catch(UnsatisfiedLinkError u)
   {
      System.out.println("ERROR:Caught a load error: " + u);
   }
   catch(Exception e)
   {
      System.out.println("ERROR:Caught an exception loading the C++ DLL: " + e);
   }
   return retval;
   // End Method modifications here
}
```

An alternative solution is to copy the needed DLL into a directory that exists in the path, then the System.loadLibrary function can be used, where only the DLL name is specified. This call would look like this:

```
System.loadLibrary("PolicyC");
```

Selecting a Pattern for Handling Essential State

This step proceeds as described in Chapter 5, “MOFW Server Programming Model” on page 57, with one significant difference: When the Java business object is installed in the server, there is a companion C++ object created to help tie it in to the server's C++ infrastructure. In order to correctly interconnect the Java object and its C++ companion, the Java object needs to implement the IManagedServer::IWrappable interface. This is reflected in the IDL created during this step, as shown in the following segment:

```
module XYZCompanyInsuranceBO
{
   interface PolicyBO : Policy,
            IManagedServer::IManagedObjectWithDataObject,
            IManagedServer::IWrappable  /* only for Java */
   {
    };
};
```

Implement Business Object Methods

In this step an implementation of the business object interface defined in “Selecting a Pattern for Handling Essential State” on page 176 is created. In Chapter 5, “MOFW Server Programming Model” on page 57 this was done using C++, but for this implementation it must be done in Java.

Because the business object client interface was called XYZCompanyInsurance::Policy (in IDL), the derived interface would be named XYZCompanyInsuranceBO::PolicyBO. Component Broker server Java requires that the implementation of this interface be provided by a class named XYZCompanyInsuranceBO._PolicyBOBase. The Base suffix is a Component Broker convention, and the underscore ensures that this class does not collide with any IDL types whose names end with Base.

The implementation class is also required to implement the PolicyBO interface and to extend the base class IManagedClient._IManageableBase. If you use the Object Builder to create your Java code it is defined with the correct inheritance.

The implementation class also needs to contain implementations for the business methods of the object, as well as some of the framework methods. Other framework methods are implemented in _IManageableBase, and do not need to be overridden. The framework methods that do require implementation are listed, but if you use the Object Builder to create your implementation class it automatically generates the correct subset of framework methods.

The implementation of the PolicyBO interface could look like the following segment:

```
package XYZCompanyInsuranceBO;

public class _PolicyBOBase
               implements PolicyBO               // See note 1
               extends com.ibm.IManagedClient._IManageableBase
{
   public void addBeneficiary(Beneficiary ben)
   {
      try
      {
         beneficiaries.addElement(ben);          // See note 2
      }
      catch(com.ibm.ICollectionsBase.IInvalidElement e)
      {
      }
   }

   public void delBeneficiary(Beneficiary ben)
   {
      try {
         beneficiaries.removeElement(ben);
      }
      catch(com.ibm.ICollectionsBase.IElementNotFound e)
      {
      }
   }

   public int policyNo() {return iPolicyNo;}     // See note 3
```

```
    public void changeAmount(float newAmount)
                            throws InvalidAmount
    {
         if(newAmount < 0.0)
               throw new InvalidAmount();
         iAmount = newAmount;                    // See note 4
    }

    public float getAmount()
    {
       return iAmount;
    }

    public String comment()
    {
       return iComment;
    }

    public void comment(String comment)
    {
       iComment = comment;
    }

    protected com.ibm.IManagedCollections.IReferenceCollection beneficiaries;
    protected int iPolicyNo;                     // See note 5
    protected float iAmount;
    protected string iComment;
    protected float iPremium;

    /* ... framework methods go here, we will see them later */

 };
```

This segment shows one possible implementation of the business methods of the Policy interface. For clarity, none of the framework methods are included in the example.

Notes about the implementation of the PolicyBO interface:

1. This class implements the PolicyBO interface and extends the standard implementation class mentioned earlier. `implements XYZCompanyInsurance.Policy` could have been written out in full, but because this class is part of the same Java package as the interface, the qualifying package name can be omitted.
2. The PolicyBO interface was specified to use the IManagedObjectWithDataObject pattern, so the business object maintains copies of all state data including the IReferenceCollection used to implement the beneficiaries relationship. The use of collections to represent relationships is described in Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105 and is not repeated here.

   The use of a try block in the addBeneficiary routine should be explained. In the IDL for the Policy interface, there was no “raises” clause specified for this routine, so that when the IDLC compiler generated the Java interface for Policy it did not include a “throws” clause on the corresponding Java method. According to the rules of the Java language, implementations of addBeneficiary are then not permitted to throw any user exceptions.

   However, the addElement() method of the IReferenceCollection.interface can throw the IInvalidElement exception. Either those exceptions must be caught and dealt with inside the addBeneficiary() method as is done in this example, or the Policy IDL must be modified to say that the addBeneficiary() method can raise the IInvalidElement exception.

3. Because the Policy IDL declared *policyNo* as a read-only attribute, only a get-accessor method is provided. Non read-only attributes also get a set-accessor. Because IDL attributes cannot take “raises” specifications, accessors can throw only subclasses of java.lang.RuntimeException. CORBA system exceptions do map to subclasses of RuntimeException, but user-defined IDL exceptions do not.

4. The correct pattern to use when setting or getting attributes depends on the mapped Java datatype. If the attribute is not an IDL interface type but maps to a modifiable Java object, copies should be created in both the getter and setter methods. This rule is also appropriate for methods like changeAmount() that do not correspond to IDL attributes but do set or return business object instance data.

   In the case of the changeAmount() method, the data type involved is “float” which is not an object type, and so simple assignment is appropriate. The same is true for IDL types like char, short, and long. It is also true for IDL string and enum types because, although they map to Java objects, the values of the objects cannot be modified.

   Contrast this with an IDL struct, which maps into a Java class with public fields. If a setter method saved a reference to the passed-in object instead of copying it, the caller could later modify the shared object. To avoid this the business object can save a copy of the passed-in value object and make another copy when returning a value. This is usually appropriate for objects corresponding to the IDL struct, union, array, sequence, exception, and any complex data types. Also, the copy should be what is usually called a deep copy so that, for example, copying an array of struct objects yields an array of copies of all the original objects.

   Any convenient deep copying technique is acceptable. Here is a standard technique that relies on the support for marshalling these objects, using the write and read methods of the corresponding Helper class:

   ```
   org.omg.CORBA.portable.OutputStream os =
      com.ibm.CBCUtil.CBSeriesGlobal.orb().create_output_stream();
   TypeHelper.write(os, object);
   Typecopy = TypeHelper,read(os);
   ```

5. These are the cached attributes as described in Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105. They can be specified as private or as protected if it is likely that a derived business object needs direct access to them. They should not be public or package access (that is, not keyword access).

Managing Memory

If you have already read Chapter 5, “MOFW Server Programming Model” on page 57 and Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105, you have seen several mentions of methods called _duplicate() and release() and the distinction between _var and _ptr references and may be wondering where are the corresponding complications in Java.

The answer is that, except for the hint in the previous section about copying values, there are none. Those complications are related to management of memory in C++, which is a manual or at best a semi-automated process.

In Java, those worries are handled automatically by the Java garbage collector. You do not need to do anything special.

Tip: Sometimes the garbage collector can be fooled into believing that an object is not garbage, even though you do not need it anymore, because your program still has a variable somewhere that refers to it.

This could cause a performance degradation if a large amount of storage is involved. You can minimize the effect by explicitly assigning null to references to large objects when you do not need them.

```
int[] iarray = new int[1000000];
/* work with iarray */
iarray = null;       /* nullify the pointer */
```

Using 'this' References in Business Objects

Care must be taken when programming all methods in business objects that use references to themselves when communicating with other objects. Methods must use the programming model as described in this section when using these self references. The technique of using “this” is no longer supported in the programming model in these circumstances.

A local proxy class is created for each interface defining the managed object implementation, the managed object interface, the business object interface, and every other interface that they may support. Only instances of the local proxy of the managed object implementation are instantiated and these proxies must be used for self references. The Helper._self(this) method can be used to access this proxy in the business object.

The home provides a copy reference to a local proxy for the create() and findBy() methods that return object references. The following example shows the set of rules to follow when an object passes itself as an argument or returns itself as a return argument:

```
public class I
{
   I foo()
   {
      // Use IHelper._self(this)
      sequence[i] = IHelper._self(this);
      struct.i = IHelper._self(this);
      return IHelper._self(this);
   };
}
```

Reference Scoping

Local references to Java Business objects are not valid outside the remote method call which created the reference. For example, it is not legal to set the value of a Java Business Object into a static variable and then retrieve that object from the static on a subsequent method call and use it. The passivation scheme for Java BO's and the mixin technology of the server require that interposing being done by the C++ MO before running outside the context of the initiating remote method is not allowed.

Implement the Managed Object Framework Methods

The framework methods that need to be implemented for a Java business object are the same as those described in Chapter 5, “MOFW Server Programming Model” on page 57, but as you would expect the Java code looks slightly different. This section includes the Java versions of those methods for the Policy business object.

IManageable::getPrimaryKeyString

The IDL return type of this method is ByteString, which is an IDL typedef. Because there is no typedef capability in Java, types are mapped according to the underlying resolved type. ByteString's underlying type is *sequence of octet*, which maps to byte[] in Java.

```
public byte[] getPrimaryKeyString()
{
   PolicyKey key = PolicyKeyHelper._create();
   key.policyNo(iPolicyNo);
   return key._toString();
}
```

The PolicyKey is a locally-implemented interface that is discussed in more detail later in this chapter. The PolicyKey interface is defined in IDL and can be implemented in either C++ or Java. You can use the _create() method of the Helper class to create an instance because that method is available whatever the implementation language.

The underscore on the _toString() method illustrates a rule from list item 3 on page 175, where the IDL method name toString() would conflict with a method introduced by java.lang.Object.

IManagedClient::IManageable::getHandleString

The Java version of this method is quite straightforward. In the following example, the stringified object reference (SOR) standard version of the Handle is used:

```
public byte[] getHandleString()
{
   com.ibm.IHandlesImpl.ISORHandle iSORHandle =
         com.ibm.IHandlesImpl.ISORHandleHelper._create(this);
   return iSORHandle._toString();
}
```

Note: This is the default implementation provided by the Framework. If this is sufficient (and it should be) there is no need to override this in the business object unless this business object will participate in relationships that are to be managed using other handle patterns.

CosStream::Streamable::externalize_to_stream

Here is one way to externalize the Policy implementation. The following segment stores a stringified reference to the beneficiaries object:

```
public void externalize_to_stream(org.omg.CosStream.StreamIO s)
{
      s.write_long(iPolicyNo);
      s.write_float(iAmount);
      s.write_float(iPremium);
      s.write_string(iComment);
      String benString = com.ibm.CBCUtil
                         .CBSeriesGlobal.orb()
                         .object_to_string(beneficiaries);
      s.write_string(benString);

}
```

CosStream::Streamable::internalize_from_stream

This method corresponds to the externalize_to_stream() method:

```
public void internalize_from_stream(org.omg.CosStream.StreamIO s,
                                    org.omg.CosLifeCycle.FactoryFinder ff)
   throws org.omg.CosLifeCycle.NoFactory,
          org.omg.CosStream.ObjectCreationError,
          org.omg.CosStream.StreamDataFormatError
   {
   if(s.read_long() == iPolicyNo)
      {
      iAmount = s.read_float();
      iPremium = s.read_float();
      iComment = s.read_string();
      String benString = s.read_string();
      beneficiaries = BeneficiariesHelper.narrow(com.ibm
                     .CBCUtil.CBSeriesGlobal.orb()
                     .string_to_object(benString));
      }
   else
     {
     throw new org.omg.CosStream.StreamDataFormatError();
     }
   }
```

As noted in Chapter 5, “MOFW Server Programming Model” on page 57, this method should not modify the key attribute *iPolicyNo* of the business object. Instead, it throws an exception if the key does not match. Note that an arbitrary user exception cannot be thrown here; it must be one that was declared in the “raises” clause in the IDL that introduced this method, which was the OMG standard interface CosStream::Streamable.

Note also the use of BeneficiariesHelper.narrow() instead of a Java cast to convert from org.omg.CORBA.Object (the return type of the string_to_object() method) to the Beneficiaries interface. The Java cast is not reliable if the object is located in a remote server, or is implemented in a language other than Java. In those situations the cast may fail when the narrow function would succeed. It is good practice to use narrow when dealing with any IDL-defined interfaces.

The streamable method implementations shown here are examples. There is no required implementation at this time because there is no dependency on these methods from within Component Broker.

IManagedServer::IManagedObject::initForCreation

If the pattern chosen for handling essential state requires a data object, this method needs to save a reference to the supplied data object. Also, if the pattern is WithCachedataObject, as it is in the PolicyBO example, the cached values in the business object need to be initialized. Because that function is needed elsewhere, you could split it into a separate private function as illustrated in the following segment:

```
private PolicyDO theDO;

private void initializeState()
{
   iPolicyNo = theDO.policyNo();
   iAmount = theDO.amount();
   iPremium = theDO.premium();
   iComment = theDO.comment();
   beneficiaries = theDO.beneficiaries();
}

public void initForCreation(com.ibm.IManagedServer.IDataObject do)
      throws com.ibm.IManagedServer.ICreationFailed
{
   theDO = PolicyDOHelper.narrow(do);

   initializeState();
}
```

IManagedServer::IManagedObject::uninitForDestruction

Nothing needs to be done for this method:

```
public void uninitForDestruction()
      throws com.ibm.IManagedServer.IDestructionFailed
{
}
```

IManagedServer::IManagedObjectWithDataObject::initForReactivation()

In the example this method saves the data object reference:

```
public void initForReactivation(com.ibm.IManagedServer.IDataObject do)
        throws com.ibm.IManagedServer.IReactivationFailed
{
   theDO = PolicyDOHelper.narrow(do);
}
```

IManagedServer::IManagedObjectWithDataObject::uninitForPassivation()

No resources need to be released for this method:

```
public void uninitForPassivation()
        throws com.ibm.IManagedServer.IPassivationFailed
{
}
```

IManagedServer::IManagedObjectWithDataObject::syncFromDataObject() and IManagedServer::IManagedObjectWithDataObject::syncToDataObject()

The business object uses the caching pattern so this method and the companion syncToDataObject() method must be implemented. This is where the initializeState() method introduced earlier is reused:

```
public void syncFromDataObject()
        throws com.ibm.IManagedServer.ISyncronizationFailed
{
   initializeState();
}

public void syncToDataObject()
        throws com.ibm.IManagedServer.ISyncronizationFailed
{
   theDO.policyNo(iPolicyNo);
   theDO.amount(iAmount);
   theDO.premium(iPremium);
```

```
    theDO.comment(iComment);
    theDO.beneficiaries(beneficiaries);
  }
```

Tip: Preserving the robustness of the Component Broker server environment imposes requirements on executing threads that are not met by threads spawned by the new java.lang.Thread() method in Java. Therefore, you should avoid creating threads in a server.

Note: The same restrictions and recomendations pertaining to initForCreation, uninitForDestruction, initForReactivation, uninitForPassivation,syncToDataObject, and syncFromDataObject apply to the Java BO versions of these methods. For additional information, see “Implementing IManagedObject Required Methods” on page 70.

Implementing the Primary Key Class

A Primary Key class is used by clients of your business object, and also by the Component Broker server infrastructure. Currently the server infrastructure needs its Primary Key to be implemented in C++, while pure Java clients need a Primary Key class implemented in Java.

Java business objects running in the server can use either the pure Java implementation, or can access the C++ class from Java through the interlanguage capabilities of IOM. The second alternative has slightly poorer performance because of the expense of interlanguage calls.

As a result, whenever you implement any Component Broker business object you must supply a C++ Primary Key class. If this class is built with IOM interlanguage bindings, it is sufficient for server Java use. However, both the pure Java client and the Java unit test environment require a second implementation of the Primary Key in pure Java; if you build one of these, you can also use it with Java in the server. In that case, the server environment contains both the C++ implementation, for use by the server infrastructure, and the Java implementation for use by Java business objects.

The Component Broker Object Builder generates both C++ and Java implementations of Primary Key classes when you use Object Builder to create the business object.

A C++ Primary Key class example was discussed in Chapter 5, “MOFW Server Programming Model” on page 57. The equivalent Java class should not be too surprising now that you have seen what the Java business object looked like. The following segment is an example of an IDL for the PolicyKey interface:

```
#include
module XYZCompanyInsuranceKeys
{

   interface PolicyKey : IManagedLocal::IPrimaryKey
      {
         attribute long policyNo;

      };
   #pragma meta PolicyKey localonly
};
```

The following segment illustrates the Java classes implementing the XYZCompany::PolicyKey interface:

```
package XYZCompanyInsuranceKeys;

public class _PolicyKeyImpl                           // See note 1
       extends com.ibm.IManagedLocal._IPrimaryKeyImpl   // See note 2
       implements PolicyKey                           // See note 3
```

```
{

   int fPolicyNo = 0;                                  // See note 4

   public int policyNo ()
   {
      return fPolicyNo;
   }

   public void policyNo (int policyNo)
   {
      fPolicyNo = policyNo;

   }

   // Methods from Streamable
   public void externalize_to_stream (                 // See note 5
          org.omg.CosStream.StreamIO targetStreamIO)
   {
      {
         targetStreamIO.write_long(fPolicyNo);
      }
   }

   public void internalize_from_stream (
           org.omg.CosStream.StreamIO sourceStreamIO,
           org.omg.CosLifeCycle.FactoryFinder there)
                   throws org.omg.CosStream.StreamDataFormatError
   {
      fPolicyNo = sourceStreamIO.read_long();
   }

   public java.lang.String getName()
   {
      return "XYZCompanyInsuranceKeys::PolicyKey";
   }

}  // _PolicyKeyImpl

package XYZCompanyInsuranceKeys;

public class PublicKeyHelper                           // See note 6
{

   public static PublicKey _create()
   {
      return new _PublicKeyImpl();
   }
}
```

Notes about the example:

1. Java Primary Key classes are intended for use both in the server and in a pure Java client. Because the Java client environment does not support IOM, this class is built as a pure Java, single-language class. One consequence is that the class name is not required to end with Base like a business object implementation. Another consequence is that references to this object can be used only from Java and cannot be passed successfully as a parameter of an interlanguage call. An attempt to do so generates a run time CORBA::MARSHAL exception.

2. This Java base class provides implementations of several methods. As described in Chapter 5, "MOFW Server Programming Model" on page 57, the IKey::isEqualToKey() and IKey::isEqualToKeyString() methods optionally can be overridden, but the supplied implementations should be sufficient for most needs.
3. The PolicyKey interface is defined in the file PolicyKey.java generated by the `IDLC -suj PolicyKey.idl` command. No other IDLC-generated source files are needed.
4. Following normal Java practice, instance data can be initialized inline, as shown here, or in a separate constructor. In this particular case, the initializer could have been omitted entirely and the Java default zero initialization of instance data accepted.
5. It is essential that the internalize and externalize methods on the Java and C++ versions of the Primary Key class must match, so that instance data can be streamed out of a Java Primary Key and then streamed into a C++ one. The Component Broker server infrastructure does this because it operates only with the C++ version.
6. Replace the Helper class that the `idlc` command generated with a simple one like this. Providing a Helper class with a static _create method allows users of your Primary Key class to create instances in a way consistent with other IDL-defined objects.

Implementing the Optional Copy Helper Class

You can implement a pure Java copy helper class for use with the IManagedClient::IHome::createFromCopyString() method. Just like the pure Java Primary Key class, the copy helper can be used in a pure Java client or on the server.

If you do create a Java copy helper, you must also produce a C++ one and install it in the server. The Component Broker infrastructure manipulates copy helpers and requires that they be implemented in C++.

```
module XYZCompanyInsuranceCopy
{
   interface PolicyCopy : IManagedLocal::INonManageable
   {
      attribute long policyNo;
      void changeAmount(in float newAmount)
                    raises (InvalidAmount);
      float getAmount();
      attribute string comment;
   };
}
```

The previous example of an IDL appears to be a subset of the Policy business object interface. This is not an accident. The purpose of the copy helper is to hold the data that is used to initialize a new Policy object, so the copy helper needs to have similar attributes. It is also a good idea to duplicate any simple validity checks that the Policy business object's logic would perform. The following Java implementation corresponds to this IDL:

```
package XYZCompanyInsuranceCopy;

public class _PolicyCopyImpl
        extends com.ibm.IManagedLocal._INonManageableImpl
        implements PolicyCopy
{
   public int policyNo()
   {
      return iPolicyNo;
   }
```

```
    public void policyNo(int no)
    {
       iPolicyNo = no;
    }

    public void changeAmount(float newAmount)
                             throws InvalidAmount
    {
       if(newAmount < 0.0)
             throw new InvalidAmount();
       iAmount = newAmount;
    }

    public float getAmount()
    {
       return iAmount;
    }

    public String comment()
    {
       return iComment;
    }

    public void comment(String comment)
    {
       iComment = comment;
    }

    private int iPolicyNo;
    private float iAmount;
    private string iComment;

    public void externalize_to_stream(org.omg.CosStream.StreamIO s)
    {
       s.write_long(iPolicyNo);
       s.write_float(iAmount);
       s.write_string(iComment);
    }

    public void internalize_from_stream(org.omg.CosStream.StreamIO s,
                            org.omg.CosLifeCycle.FactoryFinder ff)
    throws      org.omg.CosLifeCycle.NoFactory,
                org.omg.CosStream.ObjectCreationError,
                org.omg.CosStream.StreamDataFormatError
    {
       iPolicyNo = s.read_long();
       iAmount = s.read_float();
       iComment = s.read_string();
    }
 }
```

The streaming methods do not need to be stream-compatible with those in the Policy business object implementation. In fact, they cannot, because the copy helper does not contain all of the instance data that the business object does. But, as was true for the Primary Key class, it is essential that the streaming methods of the Java implementation be stream compatible with those of the C++ implementation.

Advanced Concepts

This section includes the following advanced concepts:

- “Extending a Business Object” on page 188
- “Relationships” on page 191
- “Specialized Homes” on page 191

Extending a Business Object

A Java business object can be implemented as a subclass of an existing Java business object, adding extra business methods and state data and overriding or extending the framework method implementations of the base business object class. A Java business object cannot be defined as an extension of a C++ business object.

Defining a CarPolicy interface as a refinement of Policy, this IDL is reproduced from Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105:

```
module XYZCompanyCarInsurance
{

   interface CarPolicy : Policy
   {
      attribute long year;
      attribute string make;
      attribute string model;
      attribute long serialNumber;
      attribute float collisionDeductible;
      attribute boolean glassCoverage;
      long riskQuotient();
   };
};
```

As before, a business object interface is also defined. It is not strictly necessary to explicitly inherit from the IManagedServer:IWrappable interface, because PolicyBO has already done so:

```
module XYZCompanyCarInsurance
{
   interface CarPolicyBO : CarPolicy, PolicyBO;
};
```

Development of the data object interface and implementation are unchanged from Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105, and the discussion that follows assumes an appropriate CarPolicyDO is available.

In the implementation, bodies for all the new methods must be supplied including set and get methods for the new attributes. Any framework methods whose implementations in the base business object are no longer appropriate must also be replaced:

```
package XYZCompanyCarInsurance;

public class _CarPolicyBOBase
                extends _PolicyBOBase
                implements CarPolicyBO
{
   protected int year;
   protected String make;
   protected String model;
   protected int serialNumber;
   protected float collisionDeductible;
   protected boolean glassCoverage;

   public int year()                      { return year; } // See note 1
   public void year(int y)                { year = y; }

   public String make()                   { return make; }
   public void make(String s)             { make = s; }

   public String model()                  { return model; }
   public void model(String s)            { model = s; }

   public int serialNumber()              { return serialNumber; }
   public void serialNumber(int i)        { serialNumber = i; }

   public float collisionDeductible()     { return collisionDeductible; }
   public void collisionDeductible(float f) { collisionDeductible = f; }


   public boolean glassCoverage()         { return glassCoverage; }
   public void glassCoverage(boolean b)   { glassCoverage = b; }

   public int riskQuotient()              { /* body omitted */ }

   public void externalize_to_stream(org.omg.CosStream.StreamIO s)
   {
      super.externalize_to_stream(s);                        // See note 2
      s.write_long(year);
      s.write_string(make);
      s.write_string(model);
      s.write_long(serialNumber);
      s.write_float(collisionDeductible);
      s.write_boolean(glassCoverage);
   }

   public void internalize_from_stream(org.omg.CosStream.StreamIO s,
                           org.omg.CosLifeCycle.FactoryFinder ff)
         throws      org.omg.CosLifeCycle.NoFactory,
                     org.omg.CosStream.ObjectCreationError,
                     org.omg.CosStream.StreamDataFormatError
   {
      super.internalize_from_stream(s, ff);
      year = s.read_long();
      make = s.read_string();
      model = s.read_string();
      serialNumber = s.read_long();
      collisionDeductible = s.read_float();
      glassCoverage = s.read_boolean();
   }

   private CarPolicyDO cpDO;

   private void initializeState()                            // See note 3
   {
      year = cpDo.year();
      make = cpDo.make();
```

```
      model = cpDo.model();
      serialNumber = cpDo.serialNumber();
      collisionDeductible = cpDo.collisionDeductible();
      glassCoverage = cpDo.glassCoverage();
   }

   public void initForCreation(com.ibm.IManagedServer.IDataObject inputDO)
         throws com.ibm.IManagedServer.ICreationFailed
   {
      super.initForCreation(inputDO);                              // See note 4
      cpDO = CarPolicyDOHelper.narrow(inputDO);
      initializeState();
   }

   public void initForReactivation(com.ibm.IManagedServer.IDataObject inputDO)
         throws com.ibm.IManagedServer.IReactivationFailed
   {
      super.initForReactivation(inputDO)
      cpDO = CarPolicyDOHelper.narrow(inputDO);
   }

   /* void uninitForPassivation()                                See note 5 */


   public void syncFromDataObject()
         throws com.ibm.IManagedServer.ISyncronizationFailed
   {
      super.syncFromDataObject();
      initializeState();
   }

   public void syncToDataObject()
         throws com.ibm.IManagedServer.ISyncronizationFailed
   {
      super.syncToDataObject();
      cpDo.year(year);
      cpDo.make(make);
      cpDo.model(model);
      cpDo.serialNumber(serialNumber);
      cpDo.collisionDeductible(collisionDeductible);
      cpDo.glassCoverage(glassCoverage);
   }
}
```

Notes about the example:

1. Java permits both a field named year and one or more methods named year, so you do not need to invent artificial prefixes for the names of the instance data as was done for the Policy object.
2. Here is an example where a method implementation that extends rather than replaces the PolicyBO method is supplied. To get this effect, the base class (also known as the superclass) method is invoked using Java's super keyword, which exists for this purpose. In most cases it is appropriate to call the super method first, but sometimes it makes more sense to do it last.
3. Because the PolicyBO method of the same name was private, another can exist here without introducing any ambiguity and without interfering with each another. Private methods exist independently and do not override one another, so calling the initializeState() method from within

_PolicyBOBase invokes its method, while calling the initializeState() method from within _CarPolicyBOBase gets this one.

4. Another instance where implementation of the parent is extended rather than replaced.
5. There is nothing to do in the uninitForPassivation() method, so there is no need to supply an override of the implementation in _PolicyBOBase.

Relationships

The discussion of this topic in Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105 applies almost unmodified to Java business objects:

- The discussion about the release() and _duplicate() methods does not apply because these methods are not necessary in Java.
- The discussion about the remove() method is important because the method permanently destroys a persistent object and is not done automatically as a result of garbage collection. Persistent data is erased only under explicit program control.

Specialized Homes

The basic implementation of IManagedClient::IHome provided by Component Broker provides the function of the findByPrimaryKeyString(), createFromPrimaryKeyString(), and createFromCopyString() methods. Although these methods are sufficient in many situations, there are cases where the user-friendliness of a Home would be enhanced by adding methods specific to the business object that the Home is to service. This section describes how to create these specialized Homes in Java.

Java Specialized Homes construction differs from the procedure detailed in Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105. The construction also differs from the way conventional Java business objects are built. These differences exist because the base Home implementation provided by Component Broker is written in C++ and there is no corresponding Java implementation for a Java derived class to extend. Since IOM does not support implementation inheritance between languages, the approach used earlier to create a Java CarPolicy cannot be used here.

Instead, for this special case, we simulate a restricted form of inheritance where the Java derived class is not permitted to override arbitrary base class methods, but can only add new methods. If you try to violate the override restriction by coding an override of a method in IManagedClient::IHome into the Java derived class, you will not get any compile time indication that anything is wrong but the resulting class will not behave as you expect: Whenever that method is called on the resulting Home only the C++ method will be called and the Java override will be ignored.

To use this restricted form of inheritance, which Object Builder supports only for specialized Homes, begin by specifying the Home's interface as specified in Chapter 7, “MOFW Server Programming Model – Advanced Concepts” on page 105. It is necessary to define an IDL interface containing the extension methods and have this specialized Home interface PolicyHome derive from IManagedClient::IHome, as shown here:

```
interface PolicyHome : IManagedClient::IHome
{
   XYZCompany::Policy create(in float premium, in float amount)
                raises ( IManagedClient::IInvalidKey,
                             IManagedClient::IDuplicateKey,
                             IManagedClient::IInvalidCopy );
             // create a policy passing in the attribute values

   XYZCompany::Policy defaultCreate()
```

```
                    raises ( IManagedClient::IInvalidKey,
                             IManagedClient::IDuplicateKey );

    XYZCompany::Policy createWithNumber(in long policyNo)
                    raises ( IManagedClient::IInvalidKey,
                             IManagedClient::IDuplicateKey );

    XYZCompany::Policy findByPolicyNumber(in long policyNo)
             raises ( IManagedClient::IInvalidKey,
                             IManagedClient::INoObjectWKey );

  };
```

Now create the implementation of this client interface as though extending a business object implementation called IManagedAdvancedServer::ISpecializedHome:

```
  interface PolicyHomeBO : PolicyHome,
                        IManagedAdvancedServer::ISpecializedHome,
                        IManagedServer::IWrappable
  {
  };
```

The Java implementation is pretty straightforward. Implement the extension methods, which will typically use the find and create methods of the underlying standard Home, and narrow the reference they return from IManagedClient::IManageable to the specific interface of the managed object, in this case Policy:

```
  public class _PolicyHomeBOBase
         extends com.ibm.IManagedAdvancedServer._IHomeBase      // See note 1
         implements PolicyHomeBO
  {
    public Policy create(float premium, float amount)
          throws          com.ibm.IManagedClient.IInvalidKey,
                          com.ibm.IManagedClient.IDuplicateKey,
                          com.ibm.IManagedClient.IInvalidCopy
    {
      PolicyCopy theCopy = PolicyCopyHelper._create();
      theCopy.policyNo(getUnique());                          // See note 2
      theCopy.premium(premium);
      theCopy.amount(amount);
      return PolicyHelper.narrow(
                createFromCopyString(theCopy._toString()));
    }

    public Policy defaultCreate()
          throws          com.ibm.IManagedClient.IInvalidKey,
                          com.ibm.IManagedClient.IDuplicateKey
    {
      PolicyKey pkey = PolicyKeyHelper._create();
      pkey.policyNo(getUnique());
      return PolicyHelper.narrow(
                createFromPrimaryKeyString(pkey._toString()));
    }

    public Policy createWithNumber(int policyNo)
          throws          com.ibm.IManagedClient.IInvalidKey,
                          com.ibm.IManagedClient.IDuplicateKey
    {
      PolicyKey pkey = PolicyKeyHelper._create();
      pkey.policyNo(policyNo);
      return PolicyHelper.narrow(
                  createFromPrimaryKeyString(pkey._toString()));
    }

    public Policy findByPolicyNumber(int policyNo)
          throws          com.ibm.IManagedClient.IInvalidKey,
                          com.ibm.IManagedClient.INoObjectWKey
    {
      PolicyKey pkey = PolicyKeyHelper._create();
      pkey.policyNo(policyNo);
      return PolicyHelper.narrow(
                  findByPrimaryKeyString(pkey._toString()));
    }

    private int counter = System.currentTimeMillis();         // See note 3

    private synchronized int getUnique()                      // See note 4
    {
      return counter++;
    }

    private com.ibm.IManagedAdvancedServer.ISpecializedHomeDataObject
      iDataObject;                                            // See note 5
    private String userData;

    public void initForCreation( com.ibm.IManagedServer.IDataObject theDO)
      throws com.ibm.IManagedServer.ICreationFailed           // See note 6
    {
      super.initForCreation(theDO);                           // See note 7
      iDataObject = IManagedAdvancedServer.ISpecializedHomeDataObjectHelper.narrow(theDO);
      userData = iDataObject->getConfigInfo();
  }

    public void uninitForDestruction()
     throws com.ibm.IManagedServer.IDestructionFailed
    {
      super.uninitForDestruction();
    }

    public void initForReactivation( com.ibm.IManagedServer.IDataObject
  theDO)
     throws com.ibm.IManagedServer.IReactivationFailed
    {
      super.initForReactivation(theDO);
      iDataObject = IManagedAdvancedServer.ISpecializedHomeDataObjectHelper.narrow(theDO);
      userData = iDataObject->getConfigInfo();
    }

    public void uninitForPassivation()
     throws com.ibm.IManagedServer.IPassivationFailed
    {
      super.uninitForPassivation();
    }

    public void syncToDataObject()
     throws com.ibm.IManagedServer.ISyncronizationFailed
    {
      super.syncToDataObject();
    }

    public void syncFromDataObject()
     throws com.ibm.IManagedServer.ISyncronizationFailed
    {
      super.syncFromDataObject();
    }

  };
```

Notes about the example:

1. It was previousy stated that there is no Java implementation to extend, but that appears to be happening in this example. Actually, this Java base class contains only dummy versions of the IManagedAdvancedServer::ISpecializedHome methods, which need to be present in order to successfully compile the derived class. Once the Home is installed in the server, the dummy methods are never called because the C++ implementation is used instead.
2. This pattern of creating a Copy Helper object, calling createFromCopyHelper(), and narrowing the result, is just what a client would write to implement this functionality if Policy didn't have a specialized Home. This is typically what the extension methods do. That is, they encapsulate client code sequences so the client does not need to contain them anymore.
3. A simple way to get an initial value for a unique ID generator. If something like this is not sufficient and you find you need a more robust mechanism that involves persistent storage, you should implement a normal Java business object to manage that data. This second object can then be located through the Naming Service using a fixed name known to this Home.
4. This method is marked synchronized to ensure that multiple requests running on different threads in the server do not get the same ID value.
5. This example does not make use of the Home's Data Object. For more information, see “Server Provided Essential State Extensions” on page 195.
6. The following framework methods are given special treatment so that both the methods in the C++ Home base class (IBOIMExtSystemObject::IHome_Impl) and the Java methods will be executed:
   - initForCreation()
   - uninitForDestruction()
   - initForReactivation()
   - uninitForPassivation()
   - syncToDataObject()
   - syncFromDataObject()

   This special treatment applies only to these methods. The general rule remains: For methods introduced in the specialized Home interface PolicyHome, only the Java versions will be called; for inherited methods, only the C++ versions will be called. “The Managed Object for a Java Specialized Home” on page 265explains this more fully and discusses how the Home is installed into the server.
7. You may be confused by this super call because it calls up to the Java base class whose methods, as was previously explained, are dummy methods. However, these calls are here for future compatibility.

After the Java Home extension is configured into the server environment, calls are routed to the C++ code for methods in IBOIMExtSystemObject::IHome and to the Java side for the extension methods introduced in PolicyHome, and to both implementations for those six special case framework methods. This occurs whether the call originates in C++ or Java, and whether the client is another business object or an application on a remote system.

Server Provided Essential State Extensions

This facility, described in “Leveraging Server Provided Essential State Extensions” on page 138, is also available in Java. It allows a single string of read-only data to be associated with each Home; the value is supplied under the name of userData in the Object Builder dialog and as an attribute of the same name in the generated DDL file that defines a Home.

This data is accessed by calling the ISpecializedHomeDataObject::getConfigInfo() method on the Home's Data Object. Because this is a fairly expensive operation, you should call this function only once and keep a copy of the result in your Home, as demonstrated in the previous example.

Chapter 11. Assembling and Installing Business Objects on AIX and Windows NT

The following chapter is platform-dependent and does NOT apply to OS/390 Component Broker. See *OS/390 Component Broker Programming: Assembling Applications* for further information.

Previous chapters in this book have outlined how business objects are created and unit tested. Examples of various techniques for structuring business objects have been shown. However, details about how data objects are implemented, how business objects are customized for installation and how business objects become manageable by the server have been vague or omitted.

Much of the material in this chapter is reference only. Component Broker tools generate most of the code that this chapter proclaims as necessary to have a running business object. The intention of this chapter is to provide a detailed perspective of what is happening under the covers so that additional customization can be done and a more thorough understanding of the total Programming Model can be gained. The general approach for this chapter is to use the Policy example and explain for each of these sections what additional customization is necessary to have a running business object.

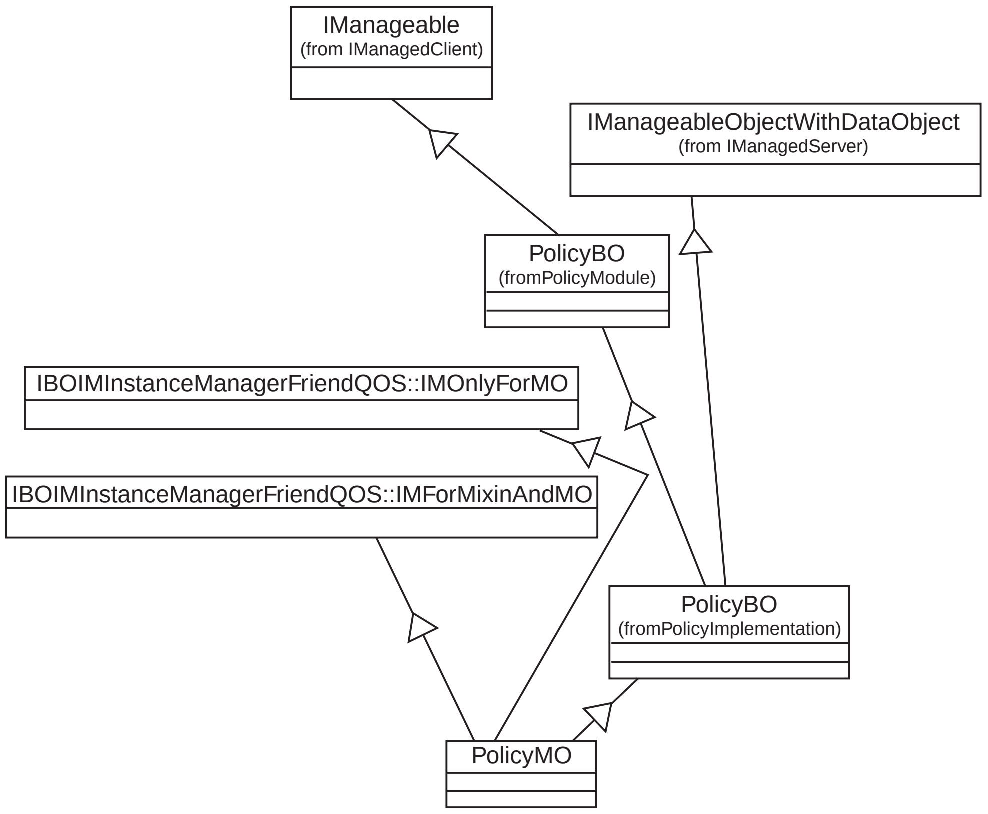
Create the Managed Object Class and Implementation

This section describes how the Managed Object Class is constructed. To complete the rest of the "Not ready to use at this point" things from Table 3 on page 80, a new class is introduced. This is called the managed object class. It adds management capabilities to the business object. It is called the PolicyMO in the example.

However, it does much more than filling out the rest of the things labeled "Not ready to use at this point" in Table 3 on page 80. There are additional methods that are implemented in the PolicyMO. These deal specifically with the application adaptor into which a business object is installed. In some cases, behavior is added or the business logic is surrounded. In others, there are additional methods that assist in implementing the quality of service provided by the application adaptor.

The example managed object is for use in BOIM Based application adaptors.

From an interface perspective, the managed object mixes together the business object interface and some additional things from the application adaptor in which the business object resides.



IManageable
(from IManagedClient)
IManageableObjectWithDataObject
(from IManagedServer)
PolicyBO
(fromPolicyModule)
IBOIMInstanceManagerFriendQOS::IMOnlyForMO
IBOIMInstanceManagerFriendQOS::IMForMixinAndMO
PolicyBO
(fromPolicyImplementation)
PolicyMO

Figure 51. IDL Hierarchy View of Managed Object

The new abstraction being introduced is the PolicyMO. The IDL for the Policy managed object is as follows:

```
#ifndef _PolicyTransMO_idl
#define _PolicyTransMO_idl
#include <PolicyBO.idl>
#include <IBOIMAInstanceManagerFriendQOS.idl>
#include <IBOIMServerFriendQOS.idl>
interface PolicyMO : PolicyBO,
             IBOIMInstanceManagerFriendQOS::IMOnlyForMO,
             IBOIMInstanceManagerFriendQOS::IMForMixinAndMO
             IBOIMServiceFriendQOS::IMDynamicDispatching
{
};
#endif
```

This is a pretty simple interface that mixes together the IDL for the business object and some additional interfaces that need to be implemented. These additional interfaces are represented by IBOIMInstanceManagerFriendQOS::IMOnlyForMO, IBOIMInstanceManagerFriendQOS::IMForMixinAndMO and IBOIMServiceFriendQOS::IMDynamicDispatching. They need to be implemented by the managed object. The implementation of these methods in the managed object is most often just a delegation to another object called the mixin.

Implementing the new PolicyMO interface requires inheritance of the PolicyBO_Impl and implementing the other interfaces of the PolicyMO generated by the emitters. The implementation interface is as follows:

```
#ifndef _PolicyMO_ih_included
#define _PolicyMO_ih_included
#ifdef SOMCBNOLOCALINCLUDES
#include <PolicyBO.ih>
#include <PolicyTransMO.hh>
#include <IBOIMInstanceManagerFriendQOS_IMixinImpl.ih>
#else
#include "PolicyBO.ih"
#include "PolicyTransMO.hh"
#include "IBOIMInstanceManagerFriendQOS_IMixinImpl.ih"
#endif

class PolicyTransMO_Impl : public virtual ::PolicyTransMO::Skeleton,
                      public virtual PolicyBO_Impl

{
   public:
   PolicyMO_Impl();
   virtual ::CORBA::Float amount ();
   virtual ::CORBA::Void amount (::CORBA::Float amount);
   virtual ::CORBA::Long policyNo ();
   virtual ::CORBA::Float premium ();
   virtual ::CORBA::Void premium (::CORBA::Float premium);
   virtual ::CORBA::Void addBeneficiary ();
   virtual ::CORBA::Void delBeneficiary ();

   #ifdef CBS_TRACE_DEBUG
   virtual ::CORBA::Void initForCreation(::IManagedServer::IDataObject_ptr theDO);
   virtual ::CORBA::Void initForDestruction();
   virtual ::CORBA::Void initForReactivation(::IManagedServer::IDataObject_ptr theDO);
   virtual ::CORBA::Void initForPassivation();
   virtual ::CORBA::Void externalize_to_stream(::CosStream::StreamIO_ptr targetStreamIO);
   virtual ::CORBA::Void internalize_from_stream(::CosStream::StreamIO_ptr sourceStreamIO,
                                               ::CosLifeCycle::FactoryFinder_ptr there);
   #endif
   virtual ::ByteString* getPrimaryKeyString();

   //Methods from Framework

   virtual ::IManagedClient::IHome_ptr getHome();
   virtual ::CosLifeCycle::Key* external_form_id();
   virtual ::CosObjectIdentity::ObjectIdentifier constant_random_id();
   virtual ::CORBA::Boolean is_identical(::CosObjectIdentity::IdentifiableObject_ptr other_object);
   virtual ::CosLifeCycle::LifeCycleObject_ptr copy(::CosLifeCycle::FactoryFinder_ptr there,
                              const ::CosLifeCycle::Criteria & the_critera);
   virtual ::CORBA::Void move(::CosLifeCycle::FactoryFinder_ptr there,
                              const::CosLifeCycle::Criteria & the_critera);
   virtual ::CORBA::Void remove();
   virtual ::IExtendedObjectIdentity::AbsoluteIdentity* get_absolute_identity();

   //Application Adaptor Methods

   virtual ::CORBA::Void checkpointToDatastore();
   virtual ::CORBA::Void refreshFromDatastore();
   virtual ::CORBA::Void externalizeKey(::IIMFLocalToServer::IKeyStream_ptr keyStream,
                                   ::IIMF::IContainer_ptr container);
   virtual ::IIMF::IContainer_ptr getContainer();
   virtual ::CORBA::Void before_completion();
```

```
    virtual ::CORBA::Void after_completion(::CosTransactions::Status status);

    //Special Methods

    virtual ::CORBA::Void setMixin(::IIMFLocalToServer::IMMixinForDelegatingMO_ptr mixinPtr);
    virtual ::IBOIM::InstanceManagerFriendQOS::IMixin_ptr getMixin();
    virtual ::CORBA::Void _incref();
    virtual ::CORBA::Void _decref();
    virtual ::CORBA::Void callMethodByName(const char * method_name, ::IBOIM ServiceFriendQOS::ArgList
                          & method_arguments, ::CORBA::Any*& method_return_value);
    virtual CORBA::Object_ptr _localReference();
    virtual void _localReference(CORBA::Object_ptr objectPointer);

    protected:
    IBOIMInstanceManagerFriendQOS_IMixinImpl mixinPointer;

    private:
    PolicyBO_var  _localProxy;
};
#endif      /* _PolicyTransMO_ih_included */
```

The following are a few kinds of methods in this interface.

- Methods defined in the business object interface (for example, Policy). These are the domain specific methods and the get and set methods associated with public attributes.
- Methods defined by the OMG Object services that are supported by the application adaptor. This also includes extensions to the OMG Object services interfaces which have been made.
- Methods defined by the BOIM application adaptor interfaces. These are augmentations made to the interface so that application adaptors can manage the business objects.
- Special Methods. These are necessary to make the infrastructure work correctly. This set of methods is actually implemented directly in the managed object.

Each of these kinds of methods are described in sections, with an example of the implementation pattern that is used.

Business Object Methods in the Managed Object Implementation

All business domain specific methods which are introduced need to have additional implementation in the managed object. This implementation makes a call to the mixin object before and after calling the real business logic. An example follows:

```
::CORBA::Float PolicyTransMO_Impl::amount()
{
    ::CORBA::Float retval;
    IBOIMExtLocalToServer_IMixinPointerImpl mixinPointer(getMixin());
    CALL_MIXIN_BEFORE2(mixinPointer,"getamount");

    #ifdef CBS_TRACE_DEBUG
       void * trc_handle = BOSS_TRACE_SERVER_START_2(this, "getamount");
    #endif

    //call the real business logic
    retval = PolicyBO_Impl::amount();

    #ifdef CBS_TRACE_DEBUG
       BOSS_TRACE_SERVER_STOP(trc_handle,"getamount");
```

```
   #endif

   CALL_MIXIN_AFTER2(mixinPointer, "getamount");
   return retval;
}
```

There are some #ifdef statements in here for enabling remote debugging. These are only included in the compiled code when there is a desire for a version of the managed object that is enabled for debug mode.

The method on the mixinPointer to constMethod ("getamount") is a special method call. This method will be generated on all attribute getters and on all methods which are marked as "constant method" using the objectBuilder business object interface wizard. This method is processed by the mixin. If every method called in a unit of work is a constMethod, then the datastore will not be unnecessarily updated. The applies only to business objects that cache in the business object.

This basic pattern for the implementation of the xxxxxMO_Impl methods applies to all business logic methods.

OMG Services Methods in the Managed Object Implementation

There is a set of methods defined by the OMG Object Services, for which the mixin object has an implementation. The pattern of implementation for these methods follows:

```
::CORBA::Boolean PolicyTransMO_Impl::is_identical(
                 ::CosObjectIdentity::IdentifiableObject_ptr other_object)
{
   IBOIMExtLocalToServer_IMixinPointerImpl mixinPointer(getMixin());
   return mixinPtr->is_identical(other_object);
}
```

The mixin object reference holder (returned by mixin()) is used to construct a mixin pointer object. The mixin pointer object adjusts a counter in the mixin reference to prevent it from being removed while it is in use. The mixin pointer "->" operator selects the mixin or alternate mixin if the mixin no longer exists.

Application Adaptor Methods in the Managed Object Implementation

Another set of methods is introduced by the application adaptor framework. The implementation pattern for these is the same as the pattern used for the object services. An example implementation follows:

```
void PolicyTransMO_Impl::checkpointToDatastore()
{
   IBOIMExtLocalToServer_IMixinPointerImpl mixinPointer(getMixin());
   mixinPointer->checkpointToDatastore();
}
```

Special Methods in the Managed Object Implementation

The setMixin(), _incref(), and _decref() methods in the managed object have special forms. The setMixin method is implemented as follows:

```
void PolicyMO_Impl::setMixin(IIMFLocalToServer::IMMixinForDelegatingMO_ptr theMixin)
{
    mixin_.setMixin(theMixin);
}
```

The setMixin method stores the pointer to the mixin in the mixin holder (mixin_). The holder is used when delegating to the mixin. If this pointer is not set correctly, attempting to use the managed object would likely raise a CORBA::INV_OBJREF exception.

The _incref() method is implemented as follows:

```
void PolicyMO_Impl::_incref()
{
    IBOIMExtLocalToServer_IMixinPointerImpl mixinPointer(mixin());
    mixinPointer._incref();
}
```

The _decref() method is similar. The mixin pointer class delegates the _incref() and _decref() methods to the mixin, if it exists, or to the ORB otherwise. Thus, the _incref() and _decref() methods work correctly even before the setMixin() method is called and after the remove() method is called.

Handling Business Object Augmentation of OMG Services Methods

In some cases, you may need to augment the behavior of some OMG Services methods in the business object. This requires you to manually change the specific managed object method to invoke both the business object and the mixin. The following is an example of the augmented remove() method:

```
::CORBA::Void PolicyMO_Impl::remove()
{
    IBOIMExtLocalToServer_IMixinPointerImpl mixinPointer(getMixin());
    mixinPointer->remove();
}
```

The above example is simplified to remove any exception handling that should be done.

Managed Objects and Specialized Homes

In most cases, subclassing is done from a base class which is also created by the object provider. Some cases, however, such as the one described in this section, require extending objects which are part of the Component Broker server.

The interface for the home, used at the Managed Object Customization phase of developing an extended home, is named IManagedAdvancedServer::ISpecializedHome. This is located in the IManagedAdvancedServer.idl file.

The interface name that is inherited for implementation is IManagedAdvancedServer_ISpecializedHome_Impl and is located in the IManagedAdvancedServer_ISpecializedHome_Impl.ih files.

Managed Objects for Specialized Homes are very similar to other Managed Objects except for the interfaces that it inherits. The following IDL is for the Specialized Home for Policy.

```
#ifndef _PolicyHomeMO_idl
#define _PolicyHomeMO_idl


#include <PolicyHomeBO.idl>
#include <IManagedAdvancedServer.idll>
interface PolicyHomeMO : PolicyHomeBO,
                         IManagedAdvancedServer::ISpecializedHome
{
};
```

```
#endif
```

The implementation is very similar. The standard framework methods that have been previously described for Managed Objects are required along with the implementation of any business methods defined for that home. In this case:

```
virtual ::Policy_ptr createWithNumber(::CORBA::Long policyNo);
virtual ::Policy_ptr findByPolicyNumber(::CORBA::Long policyNo);
```

Sample Framework Flows

The following samples show the order in which the framework invokes the IManagedObject methods for a given scenario.

Create – Business Object has Cached Data Object

1. Framework calls internalizeFromPrimaryKey or internalizeFromCopyHelper on data object.
2. Framework calls initForCreation on business object passing data object.
3. Framework calls syncToDataObject on business object.
4. Framework tells “DataObject” to insert.
5. Object is created and can now be used by client.

Create – Business Object has Data Object (Delegating)

1. Framework calls internalizeFromPrimaryKey or internalizeFromCopyHelper on data object.
2. Framework calls initForCreation on business object passing data object.
3. Framework tells “DataObject” to insert.
4. Object is created and can now be used by client.

Reactivation – Business Object has Cached Data Object (first touch)

1. Framework calls internalizeFromPrimaryKey or internalizeFromCopyHelper on data object.
2. Framework calls initForReactivation on business object passing data object.
3. Object reference returned to client.
4. Client invokes a method.
5. Framework tells “DataObject” to retrieve.
6. Framework calls syncFromDataObject on business object.

Reactivation – Find Scenario with Cached Data Object

1. Framework calls internalizeFromPrimaryKey or internalizeFromCopyHelper on data object.
2. Framework calls initForReactivation on business object passing data object.
3. Framework tells “DataObject” to retrieve.
4. Framework calls syncFromDataObject on business object.
5. Object reference return to client.
6. Client invokes a method.

Remove Object

1. Framework calls uninitForDestruction on business object.
2. Framework tells “DataObject” to deleteFromDataStore.
3. Client can no longer use the object reference.

Passivation – Business Object with Cached Data Object

1. Framework calls syncToDataStore on business object.
2. Framework tells “DataObject” to update.
3. Framework calls uninitForPassivation.

Passivation – Business Object with Data Object

1. Framework tells “DataObject” to update.
2. Framework calls uninitForPassivation.

Data Object Customization

The PolicyDO which is described in Chapter 2, “Personal Life Insurance Application Example” on page 15 is just an interface to the essential state of the business object. PolicyDO is not ready to run yet. In order to get that ready to run, PolicyDO must inherit from one of the subclasses of the abstract IDataObject class. While tools generate the implementations of the data objects, the sections that follow describe what is going on under the covers.

The IManagedServer::IDataObject interface and its subclasses provided augmentation of the basic interface to the data which was described by the object provider. Additional methods are available on the data objects that the application adaptors need to use. These interfaces are oriented around what is necessary to interact with the underlying data store. This is encapsulated from the object provider. As much as possible, a key goal of Component Broker is to encapsulate the differences in the underlying data store.

Customization, therefore, is not necessarily the job of the object provider. It is more probably the job of an application or business object customizer. The customizer has some familiarity with the domain of the business objects, but is an expert on the particular environment and resource managers into which the applications and business objects are to be installed.

Data object Customization is a key part of turning a business object into a runnable managed object. Depending on the server environment into which the business object is installed, different options for the data objects present themselves. This section goes through each of the options available for Component Broker. These are presented starting at the simplest and working towards the more complex data object customizations.

For Component Broker, there are a number of choices for data object customization. These choices are based on the support available in the server. Refer to *Component Broker for Windows NT and AIX Online Documentation* and *Component Broker Application Development Tools* for further information on data object customization.

The choices include:

Persistent Data Object - Static SQL (production use)
This is the choice to make if the data object is being customized for installation into an application adaptor that is configured to use Static SQL backed by either DB2 on NT or DB2 on MVS. This choice provides data objects with SQL included in their implementations. Optionally, if the business object is developed to be queryable (that is, returned as part of the result set of a query), then the data object must be developed to support *query pushdown*. Query pushdown refers to queries on objects in a home that can be done in the database rather than object space. The Primary Keys in this case are a subclass of the IManagedClient::IPrimaryKey base

class. For customization activity, see “BOIM Data Object Customization – Static SQL” on page 210.

Persistent Data Object -- Cache Service (production use)
An alternative to Static SQL that uses the Component Broker cache service for improved concurrency, optimistic caching. See the discussion about cache service in the *Component Broker Advanced Programming Guide* for more details. This implementation replaces the SQL statements in the DataObject with calls to the cache service. The business object developed to work with this type of DataObject is expected to be queriable and to use the delegating pattern. For more information, see “BOIM Data Object Customization – Cache Service” on page 217.

Transient Data Object -UUID Key (production use)
This is the choice to make if the data object is going to be targeted for an application adaptor that supports transient managed objects. Primary Keys in this case are a subclass of the IBOIMExtLocal::IUUIDPrimaryKey base class. This type of data object customization is described in “Transient Data Object Customization – UUID Key (Production Use)” on page 223.

Transient Data Object - (production use)
This choice is similar to the previous choice, except that any domain specific key that ensures uniqueness can be used. This type of data object customization is described in “Transient Data Object – Any Key (Production Use)” on page 225.

Transient Data Object (Unit Test)
This is the tactical choice to make if the final target of the business object is an application adaptor that is configured for persistent data such as the first two choices in the list. Primary Keys in this case are a subclass of the IManagedClient::IPrimaryKey base class. This is the first type of customization explained in this chapter, starting at “Data Objects (Unit Test)” on page 207.

Table 4 on page 206 summarizes these data object customization choices and the interfaces they use as part of the customization.

Table 4. Interfaces Needed by Persistent Data Objects

Interface Required	Data Object Customization			Notes
	Static SQL data object	**Static SQL with query push down**	**Caching Cache Service data object with query push down**	
IBOIMExtLocalToServer::IQueryable Data Object	No	Yes	Yes	This interface adds the internalizeData() method that must be implemented.
IBOIMExtLocalToServer::IHomeAwareDataObject	No	Yes	Yes	This interface adds the setHome() method whose implementation is provided by the Component Broker implementation of the IRDBIMExtLocalToServer::ICachingServiceDataObject interface. For the Caching DAO data object with query pushdown, this interface is inherited by IRDGMIMExtLocalTOServer::ICachingServiceDataObject and so does not have to be explicitly inherited.
IRDBIMExtLocalToServer::IDataObject	Yes	Yes	No	This interface adds the setConnection() method that should be implemented in the persistent object. This persistent object is a companion object to the SQL data object that allows the database specific code, in this case SQL, to be isolated from non-specific data object code.
IRDBIMExtLocalToServer::ICachingServiceDataObject	No	No	Yes	This interface adds the getMOInterfaceName(), getMapExpression(), and setHome() methods. These method implementations are provided by the interface implementation.

Data Objects (Unit Test)

To test business logic using the unit test environment, the data object must have a simple and self-contained data object customization. For information on creating a self-contained data object, see Appendix D, “Unit Test Environment” on page 333.

Transient Data Object Interfaces

At the IDL level, the implementation diagram is simple. The following figure is an implementation diagram for a transient data object.

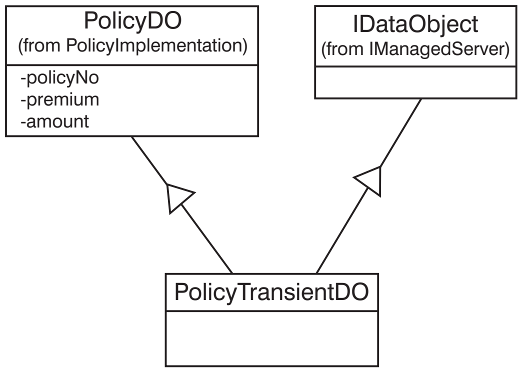


Figure 52. Transient Data Object Implementation

The PolicyDO interface is:

```
interface PolicyDO
{
    attribute float amount;
    attribute long policyNo;
    attribute float premium;
    #pragma meta PolicyDO localonly ,abstract
};
```

PolicyTransientDO is an interface that mixes together the interface to the essential state and the data object implementation which is desired. In this case, the IManagedServer::IDataObject interface is all that is required.

The IDL for this is:

```
#include "PolicyDO.idl"
#include <IManagedServer.idl>

interface PolicyTransientDO : PolicyDO, IManagedServer::IDataObject
{
    #pragma meta PolicyTransientDO localonly abstract
}
```

PolicyTransientDO is the interface that is implemented. The implementation must support the get and set methods that come from the attributes described in PolicyDO and must also support the interface required by IManagedServer::IDataObject. The implementation interface of PolicyTransientDO is:

```
#ifndef _PolicyTransientDO_ih_included
#define _PolicyTransientDO_ih_included

#include "PolicyTransientDO.hh"

class PolicyTransientDO_Impl : public virtual ::PolicyTransientDO_Skeleton
{
    public:
      PolicyTransientDO_Impl::PolicyTransientDO_Impl();
        ::CORBA::Long policyNo ();
        ::CORBA::Void policyNo (::CORBA::Long policyNo);
        ::CORBA::Float premium ();
        ::CORBA::Void premium (::CORBA::Float premium);
        ::CORBA::Float amount ();
        ::CORBA::Void amount (::CORBA::Float amount);

      virtual ::CORBA::Void internalizeFromPrimaryKey(::IManagedLocal::IPrimaryKey_ptr inKey);
      virtual ::CORBA::Void internalizeFromCopyHelper(::IManagedLocal::INonManageable_ptr inCopy);

    private:
    // Store the values of the public attributes transiently.

       ::CORBA::Long  tPolicyNo;  // Store value of "policyNo" attribute
       ::CORBA::Float tPremium;   // Store value of "premium"  attribute
       ::CORBA::Float tAmount;    // Store value of "amount"   attribute
};

#endif            /* _PolicyTransientDO_ih_included */
```

This is the minimal implementation interface required for this data object. In the transient case, there is no implementation to inherit.

The implementation of this interface is the final part of the transient data object customization. This is shown in the following sections.

Transient Data Object Implementation

This section includes the following topics:

- “Framework Required Method internalizeFromPrimaryKey”
- “Framework Required Method internalizeFromCopyHelper” on page 209
- “Framework Required create() function” on page 209
- “Methods To Support Attributes – Getters” on page 209
- “Methods to Support Attributes – Setters” on page 210
- “Additional Methods – Default Constructor” on page 210

Framework Required Method internalizeFromPrimaryKey: The MOFW requires the data object to implement the IManagedServer::IDataObject::internalizeFromPrimaryKey() method. The *in* parameter is an IManagedLocal::IPrimaryKey. This key is how the business object is uniquely identified. In the case of the example, the *in* parameter is narrowed to a PolicyKey and the policy number is extracted and stored as part of the data object. This method is important because it establishes the linkage between the key and the data object that supports the business object being constructed.

```
::CORBA::Void PolicyTransientDO_Impl::internalizeFromPrimaryKey(
              ::IManagedLocal::IPrimaryKey_ptr inKey)
{
    PolicyKey_var pk = PolicyKey::_narrow(inKey);
              /* Stores the key attribute in the DO data attribute */
    tPolicyNo = pk->policyNo();
              /* Stores the value of the "policyNo" attribute */
}
```

This method is called by the unit test environment during construction of objects and during reactivation of objects (although reactivation doesn't really make sense in the transient case).

Framework Required Method internalizeFromCopyHelper: The MOFW requires the data object to implement the IManagedServer::IDataObject::internalizeFromCopyHelper() method. The *in* parameter is an IManagedLocal::INonManageable. This copy contains state data, including the key that shows how the business object is uniquely identified. In the case of the example, the *in* parameter is narrowed to a PolicyCopy and the data is extracted and stored as part of the data object.

```
::CORBA::Void PolicyTransientDO_Impl::internalizeFromCopyHelper(
              ::IManagedLocal::INonManageable_ptr inCopy)
{
    PolicyCopy_var pc = PolicyCopy::_narrow(inCopy);
    tPolicyNo = pc->policyNo();        /* Stores the value of the "policyNo" attribute */
    tAmount = pc->amount();
    tPremium = pc -> premium();
}
```

This method is called by the unit test environment during construction of objects.

Framework Required create() function: This method is called by the unit test environment or application adaptor when an empty uninitialized data object is required. This happens during business object creation and reactivation.

The implementation must have an external entry point so that when the application adaptor loads the DLLs required for a particular business object (which happens dynamically), it can access the create() function. The code for the example looks like:

```
// The following extern "C" function is required so that a (generic) home can be
// configured to load the Policys DLL and create a PolicyTransientDO without knowing
// anything about the Policy business objects.

extern "C"
{
    SOMDLLEXPORT void* PolicyTransientDO_create()
    {
        return (void*) ( PolicyTransientDO::_create() );
    }
}
```

Methods To Support Attributes – Getters: All of the attributes need to have a getter method on them. A getter method for one of the attributes of the PolicyDO follows:

```
// * Method from the IDL attribute statement:  "attribute Float amount"

::CORBA::Float PolicyTransientDO_Impl::amount()
{
    return tAmount;
}
```

Methods to Support Attributes – Setters: All of the attributes need to have a set method on them. A setter method for one of the attributes of the PolicyDO follows:

```
// Method from the IDL attribute statement: "attribute Float amount"

::CORBA::Void PolicyTransientDO_Impl::amount(::CORBA::Float amount)
{
    tAmount = amount;
}
```

Additional Methods – Default Constructor: The transient data object implementation has values for each of the attributes that the business object is interested in. If the business object is caching, the values from the data object are loaded into the business object when the business object methods first require access to state data. In the delegating case, values in the data object are accessed directly, as needed. Either way, when the managed object is originally created, these default constructor values can be used. This ensures that uninitialized data does not get used during the period of time where the key or copy helper is known to the data object, but the data object has not yet been created into or refreshed from the data store. Therefore the values in the data object should be initialized properly in a default constructor. It is required that a default constructor be developed.

```
// Default constructor

PolicyTransientDO_Impl::PolicyTransientDO_Impl()
: tAmount(0),       /* initialization */
  tPolicyNo(0),
  tPremium(0)
{
}
```

Another Option

Beyond this transient view of a data object implementation, there are a number of possibilities for data objects that don't require any additional infrastructure from the server. A file-based implementation of a data object could be constructed by extending the transient business object that has been described. File naming might be based on the key to the object. Opening the file would be put in the internalize*XXXX* methods. Closing the file would be done in the constructor. Updating the file could be done at the end or after every setter. Reading the file could be done upon opening the file or when a getter is done. Various options exist. It is not the intention of this section to point out all of the possibilities.

More complex backing stores could be leveraged following the same pattern that would be appropriate for files. However, the unit test environment in which these sorts of data objects run does not provide the object server function that is available from a real application adaptor. There is no consideration for security and transactions in these scenarios. There is also limited support for reactivation. Finding objects is based only on those objects that have been activated. It might be necessary to activate all objects before running them in this sort of configuration.

BOIM Data Object Customization – Static SQL

If the data object customization is going to target Static SQL data objects that run in a DB2 application adaptor, then you should read this section. In this section, the PolicyDO interface is used as the basis for describing the customization necessary.

SQL Data Object Interfaces

The IDL-based picture for SQL-backed BOIM data objects follows:

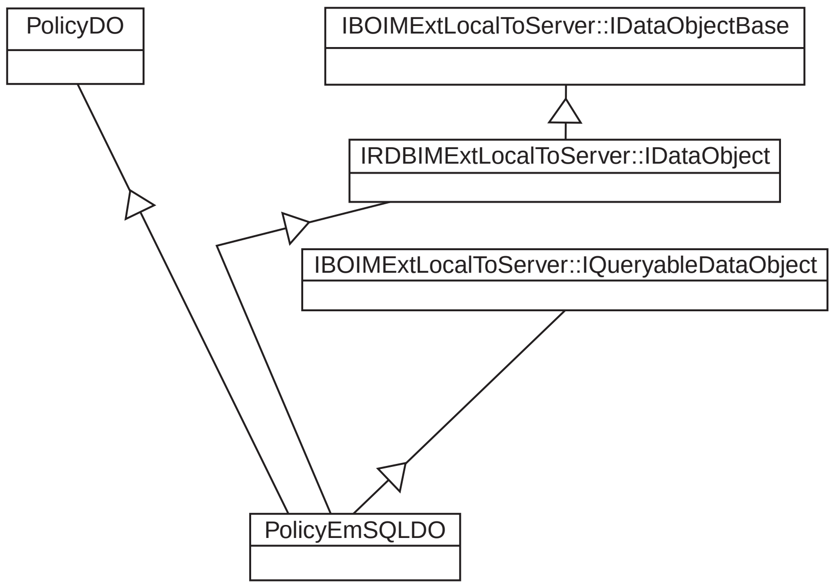

PolicyDO
IBOIMExtLocalToServer::IDataObjectBase
IRDBIMExtLocalToServer::IDataObject
IBOIMExtLocalToServer::IQueryableDataObject
PolicyEmSQLDO

Figure 53. SQL BOIM Data Object IDL Interface View

The example shows the case where the business object is also configured to be queryable. (that is, IBOIMExtLocalToServer::IQueryableDataObject should be inherited if the data object is to be used with queryable homes otherwise it should not be inherited.) Here is the IDL for this:

```
#include <PolicyDO.idl>
#include <IRDBIMExtLocalToServer.idl>
#include <IBOIMExtLocalToServer.idl>

interface PolicyEmSQLDOImpl : PolicyDO,
             IRDBIMExtLocalToServer::IDataObject,
             IBOIMExtLocalToServer::IQueryableDataObject
{
    #pragma meta PolicyEmSQLDO localonly
};
```

In fact there is more implementation inheritance as well. The interfaces added to the basic PolicyDO include those required for all two-level store application adaptors (BOIM) and those specific to the relational database adaptor (RDBIM) This is shown in the implementation view following:

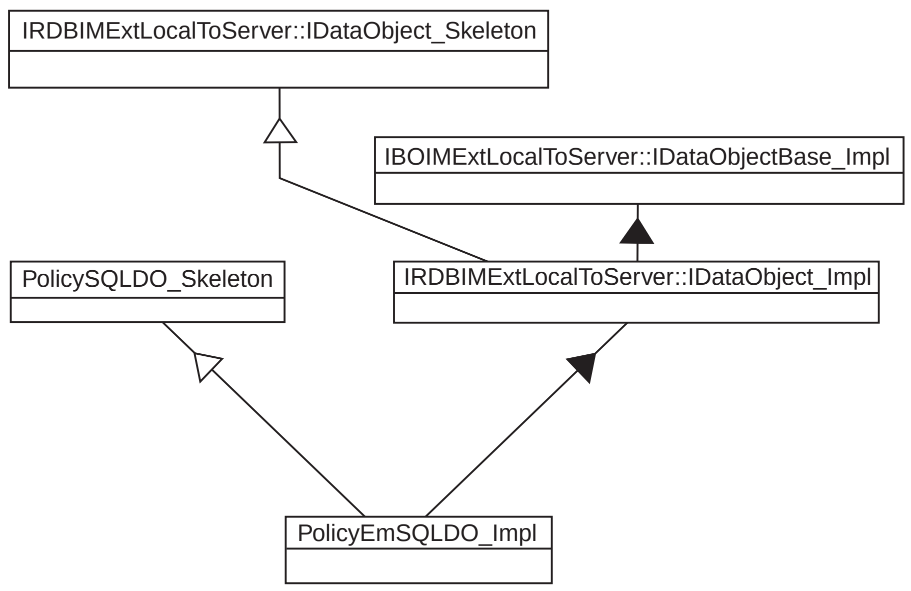

IRDBIMExtLocalToServer::IDataObject_Skeleton
IBOIMExtLocalToServer::IDataObjectBase_Impl
PolicySQLDO_Skeleton
IRDBIMExtLocalToServer::IDataObject_Impl
PolicyEmSQLDO_Impl

Figure 54. SQL BOIM Data Object Implementation Interface Inheritance

The IRDBIMExtLocalToServer::IDataObject_Impl in combination with IBOIMExtLocalToServer::IDataObjectBase_Impl provides an implementation for some of the methods that are described in the IBOIMExtLocalToServer::IDataObject and IRDBIMExtLocalToServer::IDataObject interfaces. The following methods have implementations which are inherited:

- string connection()
- void connection(string value)
- void markDirty()
- void clearDirty()
- boolean isDirty()
- void updateToDataStore()
- void insertToDataStore()
- void retrieveFromDataStore()
- void deleteFromDataStore()

These methods should not be overridden.

Given this base, the PolicyEmSQLDO_Impl implementation interface should look like this:

```
#include <PolicyEmSQLPO.hpp>

#ifdefSOMCBNOLOCALINCLUDES
#include <IRDBIMExtLocalToServer.ih>
#include <PolicyEmSQLDOImpl.hh>
#else
#include "IRDBIMExtLocalToServer.ih"
#include "PolicyEmSQLDOImpl.hh"
#endif

class PolicyEmSQLDOImpl_Impl : public virtual ::PolicyEmSQLDOImpl_Skeleton,
public virtual IRDBIMExtLocalToServer_IDataObject_Impl
{
    public:

   //default constructor doimpl
   PolicyEmSQLDOImpl_Impl();
```

```
    ::CORBA::Float amount();
    ::CORBA::Void amount( ::CORBA::Float amount);
    ::CORBA::Long policyNo();
    ::CORBA::Void policyNo( ::CORBA::Long policyNo);
    ::CORBA::Float premium();
    ::CORBA::Void premium( ::CORBA::Float premium);

    virtual ::CORBA::Void insert();
    virtual ::CORBA::Void update();
    virtual ::CORBA::Void retrieve();
    virtual ::CORBA::Void del();
    virtual ::CORBA::Void setConnection(const char* dataBaseName);
    virtual ::CORBA::Void internalizeFromPrimaryKey(
                   ::IManagedLocal::IPrimaryKey_ptr inKey);
    virtual ::CORBA::Void internalizeFromCopyHelper(
                   ::IManagedLocal::INonManageable_ptr inCopy);
    virtual ::CORBA::Void internalizeKeyAttributes(
                   ::IIMFLocalToServer::IKeyComponent_ptr keyComp);
    virtual ::CORBA::Void externalizeKeyAttributes(
                   ::IIMFLocalToServer::IKeyComponent_ptr & keyComp);
    virtual ::CORBA::Void internalizeData(
             const ::IBOIMLocalToServerMetadata::dataSequence & dataSeq);
    virtual ::CORBA::Boolean verifyKey();

    protected:
    private:
    ::PolicyEmSQLPO iPolicyEmSQLPO;
    ::CORBA::Boolean iKeyValueSet;
 };
```

Static SQL Data Object Implementation

The following methods need to be implemented in order for the Static SQL data object to work properly in the server.

Framework Required Method – internalizeFromPrimaryKey: See the description in “Framework Required Method internalizeFromPrimaryKey” on page 208. It is similar for Static SQL data objects.

In addition, the `tKeyValueSet` flag should be set to true when valid key values are successfully retrieved from the key.

```
::CORBA::Void PolicyEmSQLDOImpl_Impl::internalizeFromPrimaryKey
                (::IManagedLocal::IPrimaryKey_ptr inKey)
{
    // Insert Method modifications here
    PolicyKey_var iPolicyKey = PolicyKey::_narrow(inKey);
    long iPolicyNoTemp = iPolicyKey->policyNo();
    iKeyValueSet = 1;
    PolicyEmSQLPOKey iPolicyEmSQLPOKey;
    iPolicyEmSQLPOKey.policyNo(iPolicyNoTemp);
    iPolicyEmSQLPO.internalizeFromPrimaryKey(iPolicyEmSQLPOKey);

    // End Method modifications here
}
```

Framework Required Method – internalizeFromCopyHelper: See the description in “Framework Required Method internalizeFromCopyHelper” on page 209. It is the similar for static SQL data objects.

In addition, the tKeyValueSet flag should be set to true when valid key values are successfully retrieved from the key.

Framework Required Code – create() Function: See the description in “Framework Required create() function” on page 209. It is similar for static SQL-based data objects.

```
extern "C"
{
    SOMDLLEXPORT IBOIMExtLocalToServer::IDataObject_ptr PolicyEmSQLDOImpl_create()
    {
        return new PolicyEmSQLDOImpl_Impl();
    }
}
```

Methods To Support Attributes – Getters: See the description in “Methods To Support Attributes – Getters” on page 209. It is similar for static SQL-based data objects. For example, the getter for amount would be:

```
::CORBA::Float PolicyEmSQLDOImpl_Impl::amount()
{
    // Insert Method modifications here
    ::CORBA::Float iAmountTemp;
    iAmountTemp = iPolicyEmSQLPO.amount();
    return iAmountTemp;

    // End Method modifications here
}
```

Methods to Support Attributes – Setters: All of the attributes need to have a set method on them. A setter method for one of the attributes of the PolicyEmSQLDO follows:

```
::CORBA::Void PolicyEmSQLDOImpl_Impl::amount(::CORBA::Float amount)
{
    // Insert Method modifications here
    ::CORBA::Float iAmountTemp = amount;
    iPolicyEmSQLPO.amount(iAmountTemp);
    markDirty();

   // End Method modifications here
}
```

The difference here from the transient case is where the markDirty() method is run. This indicates to the application adaptor that it is necessary to update the underlying data store at prescribed times.

Additional Methods – Default Constructor: See the description in “Additional Methods – Default Constructor” on page 210. It is the same for BOIM-based data objects. Additionally, the iKeyValueSet should be initialized to false. The updated constructor looks like this:

```
PolicyEmSQLDOImpl_Impl::PolicyEmSQLDOImpl_Impl()
:iKeyValueSet(0)
{
}
```

Required Method – verifyKey: This method returns a boolean to indicate whether or not the key values in the data object have been acquired and are valid. This method is used by the application adaptor to validate the key before an object reference for the object is built.

```
::CORBA::Boolean PolicyEmSQLDOImpl_Impl::verifyKey ()
{
    //Insert Method modifications here
    return iKeyValueSet;
    //End Method Modifications here
}
```

Required Method – externalizeKeyAttributes: This method is used by the application adaptor as an object reference for the business object being created. The code in this method writes out the key values into the `outKey` that is provided on the parameter list.

```
::CORBA::Void PolicyEmSQLDOImpl_Impl::externalizeKeyAttributes
                (::IIMFLocalToServer::IKeyComponent_ptr& keyComp)
{
    // Insert Method modifications here
    long iPolicyNoTemp;
    PolicyEmSQLPOKey iPolicyEmSQLPOKey;
    iPolicyEmSQLPO.externalizeKeyAttributes(iPolicyEmSQLPOKey);
    iPolicyNoTemp = iPolicyEmSQLPOKey.policyNo();
    keyComp->write_long(iPolicyNoTemp);
    // End Method modifications here
}
```

Required Method – internalizeKeyAttributes: This method is used by the application adaptor for reactivating objects. This method should pull information out of the `inKey` provided and set the flag indicating that the key is valid to true. Additional logic may be placed in this method to further verify that the key value is good.

```
::CORBA::Void PolicyEmSQLDOImpl_Impl::internalizeKeyAttributes
                 (::IIMFLocalToServer::IKeyComponent_ptr keyComp)
{
    // Insert Method modifications here
    long iPolicyNoTemp = keyComp->read_long();
    iKeyValueSet = 1;
    PolicyEmSQLPOKey iPolicyEmSQLPOKey;
    iPolicyEmSQLPOKey.policyNo(iPolicyNoTemp);
    iPolicyEmSQLPO.internalizeKeyAttributes(iPolicyEmSQLPOKey);
    // End Method modifications here
}
```

Required Method – update: This is the code that puts data back into the database. This is .sqx code that needs to run through the SQL preprocessor before it is compiled.

```
::CORBA::Void PolicyEmSQLDOImpl_Impl::update ()
{
    // Insert Method modifications here
    iPolicyEmSQLPO.update();
    // End Method modifications here
}
```

Required Method – insert: The insert is called when a create is done. The code looks like this:

```
::CORBA::Void PolicyEmSQLDOImpl_Impl::insert ()
{
    // Insert Method modifications here
    iPolicyEmSQLPO.insert();
    // End Method modifications here
}
```

Required Method – retrieve: The retrieve gets the data from the database and looks like this:

```
::CORBA::Void PolicyEmSQLDOImpl_Impl::retrieve ()
{
    // Insert Method modifications here
    iPolicyEmSQLPO.retrieve();
    // End Method modifications here
}
```

Required Method – del

```
::CORBA::Void PolicyEmSQLDOImpl_Impl::del ()
{
    // Insert Method modifications here
    iPolicyEmSQLPO.del();
    // End Method modifications here
}
```

Required Method – setConnection: This method defines the database to which the SQL in the insert(), del(), retrieve(), and update() methods is directed.

```
::CORBA::Void PolicyEmSQLDOimpl_Impl::setConnection( const char* dataBaseName)
{
    // Insert Method modifications here
    iPolicyEmSQLPO.setConnection(dataBaseName);
    // End Method modifications here
}
```

Required Method – internalizeData: This method enables query to work on the data object.

```
::CORBA::Void PolicyEmSQLDOImpl_Impl::internalizeData(
                 const::IBOIMLocalToServerMetadata::dataSequence & dataSeq)
{
    // Insert Method modifications here
    PolicyEmSQLPOCopy iPolicyEmSQLPOCopy;

    long PolicyEmSQLPO_policyNoTemp;
    if (!((((dataSeq[0]))) >>= PolicyEmSQLPO_policyNoTemp))
    PolicyEmSQLPO_policyNoTemp = NULL;
    iPolicyEmSQLPOCopy.policyNo(PolicyEmSQLPO_policyNoTemp);

    double PolicyEmSQLPO_amountTemp;
    if (!((((dataSeq[1]))) >>= PolicyEmSQLPO_amountTemp))
    PolicyEmSQLPO_amountTemp = NULL;
    iPolicyEmSQLPOCopy.amount(PolicyEmSQLPO_amountTemp);

    double PolicyEmSQLPO_premiumTemp;
    if (!((((dataSeq[2]))) >>= PolicyEmSQLPO_premiumTemp))
    PolicyEmSQLPO_premiumTemp = NULL;
    iPolicyEmSQLPOCopy.premium(PolicyEmSQLPO_premiumTemp);

    iPolicyEmSQLPO.internalizeData(iPolicyEmSQLPOCopy);
    iKeyValueSet = 1;

    // End Method modifications here
}
```

Additional Considerations

The Static SQL data object could contain the SQL code within the update(), insert(), retrieve(), and delete() methods. It is more convenient to put this code into a C++ helper object that is referred to as the persistent object. The persistent object has many of the same methods as the data object and the data object just deals with the data in the data object and lets the persistent object take care of using SQL.

BOIM Data Object Customization – Cache Service

If the data object customization is going to target cache service data objects that run in a DB2 application adaptor, then you should read this section. In this section, the PolicyDO interface is used as the basis to describe the necessary customization.

To create a data object implementation that uses the cache service, you must use Object Builder to create the data object. When defining the data object implementation and persistent object, be sure to check the **Use Cache Service** check box.

Managed Objects that use the cache service must also be configured to use Containers with the **Use Cache Service** option.

Cache Service Data Objects Interfaces

The IDL-based picture for cache service data objects follows:

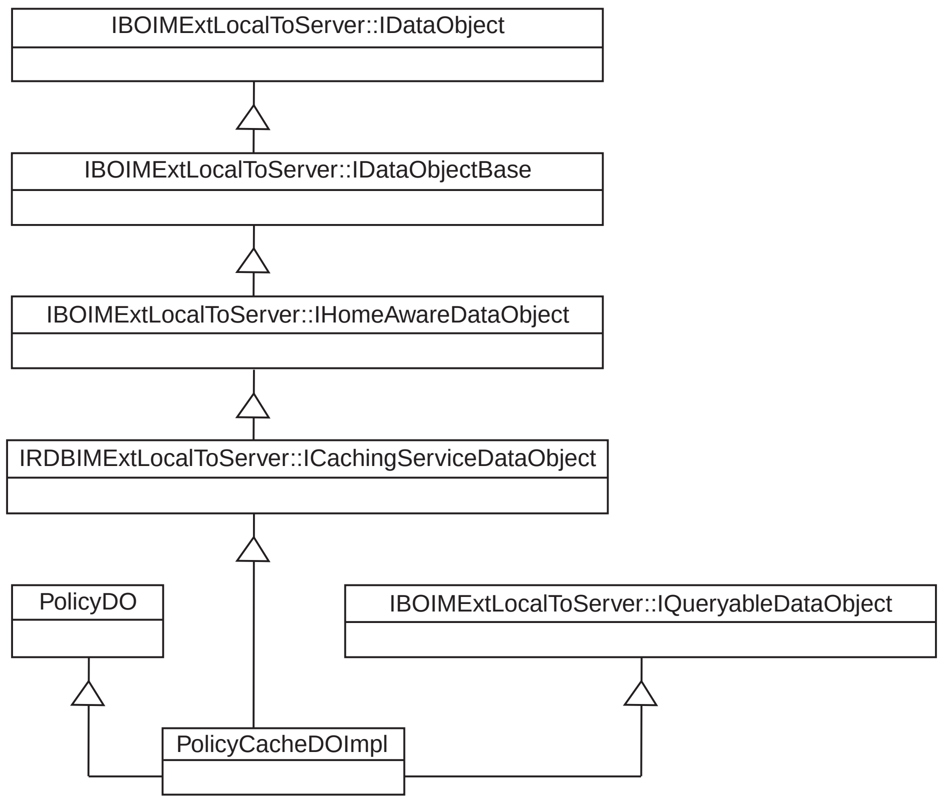

IBOIMExtLocalToServer::IDataObject
IBOIMExtLocalToServer::IDataObjectBase
IBOIMExtLocalToServer::IHomeAwareDataObject
IRDBIMExtLocalToServer::ICachingServiceDataObject
PolicyDO
IBOIMExtLocalToServer::IQueryableDataObject
PolicyCacheDOImpl

Figure 55. Cache Service Data Object IDL Interface View

The example shows the case where the business object is also configured to be queryable. (That is, IBOIMExtLocalToServer::IQueryableDataObject should be inherited because the data object is to be used with queryable homes.) The IDL is:

```
#include <PolicyDO.idl>
#include <IRDBIMExtLocalToServer.idl>
#include <IBOIMExtLocalToServer.idl>
interface PolicyCacheDOImpl : PolicyDO,
   IRDBIMExtLocalToServer::ICachingServiceDataObject,
   IBOIMExtLocalToServer::IQueryableDataObject
{
   #pragma meta PolicyCacheDOImpl localonly
};
```

From this, it can be seen that at the IDL level, the difference between the embedded SQL case and the cached DAO case is whether to inherit IRDBIMExtLocalToServer::IDataObject or IRDBIMExtLocalToServer::ICachingServiceDataObject.

These differences are also apparent from looking at the implementation interfaces and how they interact to provide the basis for the PolicyCacheDOImpl class which is used to implement the PolicyCacheDOImpl interface shown in the previous figure. The implementation interface inheritance picture follows.

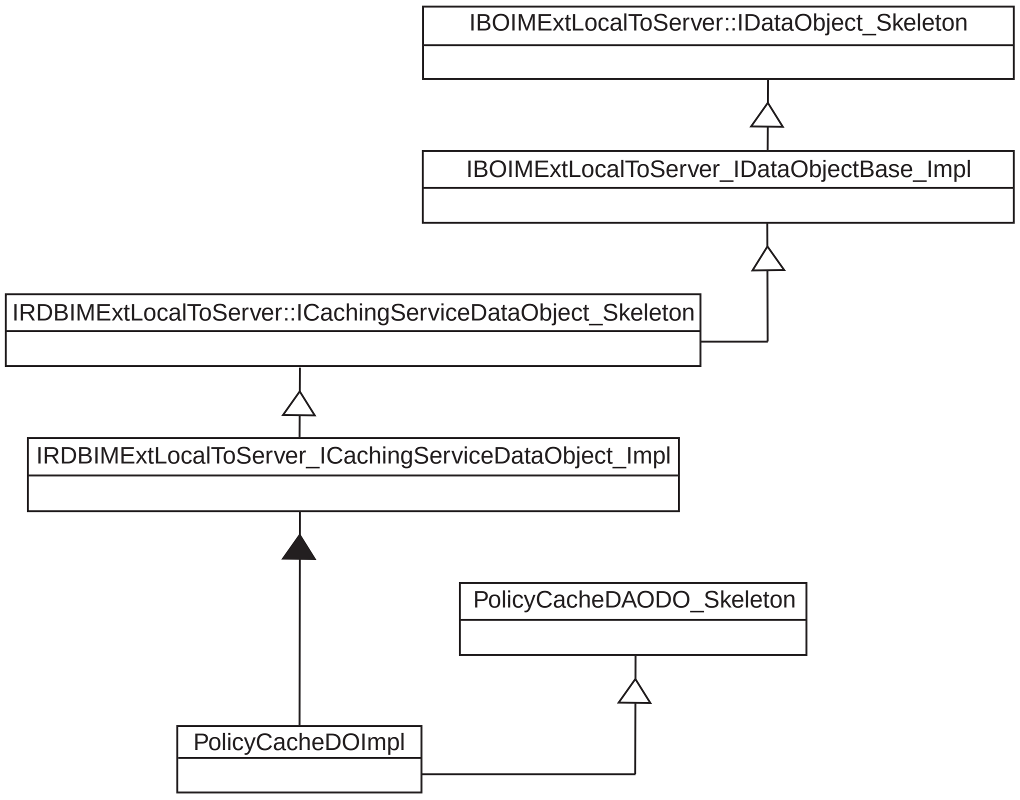

IBOIMExtLocalToServer::IDataObject_Skeleton
IBOIMExtLocalToServer_IDataObjectBase_Impl
IRDBIMExtLocalToServer::ICachingServiceDataObject_Skeleton
IRDBIMExtLocalToServer_ICachingServiceDataObject_Impl
PolicyCacheDAODO_Skeleton
PolicyCacheDOImpl

Figure 56. BOIMDO Implementation Interface Inheritance

The IBOIMExtLocalToServer_IDataObjectBase_Impl provides an implementation for some of the methods that are described in the IBOIMExtLocalToServer::IDataObject interface which is represented in C++ in the previous figure by the IBOIMExtLocalToServer::IDataObject_Skeleton. The following methods have implementations which are inherited:

- string connection()
- void connection(string value)
- void markDirty()
- void clearDirty()
- boolean isDirty()
- void updateToDataStore()

- void insertToDataStore()
- void retrieveFromDataStore()
- void deleteFromDataStore()

These methods should not be overridden.

Given this base, the PolicyCacheDOImpl_Impl implementation interface should look like this:

```
#include "PolicyCachePO.hpp"

#ifdefSOMCBNOLOCALINCLUDES
#include <IRDBIMExtLocalToServer.ih>
#include <PolicyCacheDOImpl.hh>
#else
#include "IRDBIMExtLocalToServer.ih"
#include "PolicyCacheDOImpl.hh"
#endif

class PolicyCacheDOImpl_Impl : public virtual ::PolicyCacheDOImpl_Skeleton,
public virtual IRDBIMExtLocalToServer_ICachingServiceDataObject_Impl
{
   public:

   //default constructor doimpl
   PolicyCacheDOImpl_Impl();

   ::CORBA::Float amount();
   ::CORBA::Void amount( ::CORBA::Float amount);
   ::CORBA::Long policyNo();
   ::CORBA::Void policyNo( ::CORBA::Long policyNo);
   ::CORBA::Float premium();
   ::CORBA::Void premium( ::CORBA::Float premium);

   virtual ::CORBA::Void insert();
   virtual ::CORBA::Void update();
   virtual ::CORBA::Void retrieve();
   virtual ::CORBA::Void del();
   virtual ::CORBA::Void internalizeFromPrimaryKey(
                  ::IManagedLocal::IPrimaryKey_ptr inKey);
   virtual ::CORBA::Void internalizeFromCopyHelper(
                  ::IManagedLocal::INonManageable_ptr inCopy);
   virtual ::CORBA::Void internalizeKeyAttributes(
                  ::IIMFLocalToServer::IKeyComponent_ptr keyComp);
   virtual ::CORBA::Void externalizeKeyAttributes(
                  ::IIMFLocalToServer::IKeyComponent_ptr & keyComp);
   virtual ::CORBA::Void internalizeData(
            const ::IBOIMLocalToServerMetadata::dataSequence & dataSeq);
   virtual ::CORBA::Boolean verifyKey();

   protected:
   private:
   ::PolicyCachePO iPolicyCachePO;
   ::CORBA::Boolean iKeyValueSet;
};
```

Most of the methods are the same as the Static SQL implementation. Only the methods that differ are described here.

Required Method - default constructor:: This method uses a different initialization technique by using the PO capabilities of attribute initialization.

```
PolicyTransDOImpl_Impl::PolicyTransDOImpl_Impl()
:iKeyValueSet(0)
{
   policyNo(0);
   amount(0);
   premium(0);
}
```

Required Method - internalizeFromPrimaryKey:: This method must handle the exceptions throw by the PO and map them to exceptions that are accepted by the framework. Since this method is called for both the create and find scenarios, additional logic is required to either initialize the PO or retrieve the already existing values. The isNew() method is used to make this determination.

```
::CORBA::Void PolicyCacheDOImplCache_Impl::internalizeFromPrimaryKey(
::IManagedLocal::IPrimaryKey_ptr inKey)
{
   // Insert Method modifications here
   PolicyKey_var iPolicyKey = PolicyKey::_narrow(inKey);
   long iPolicyNoTemp = iPolicyKey->policyNo();
   iKeyValueSet = 1;
   PolicyCacheDOImplCachePOKey iPolicyCacheDOImplCachePOKey;
   iPolicyCacheDOImplCachePOKey.policyNo(iPolicyNoTemp);
   try
   {
      iPolicyCacheDOImplCachePO.internalizeFromPrimaryKey(
      iPolicyCacheDOImplCachePOKey);
      if (isNew())
      {
         iPolicyCacheDOImplCachePO.create();
         amount(0);
         premium(0);
      }
      else
      {
         iPolicyCacheDOImplCachePO.retrieve();
      }
   }
   catch (IBOIMException::IDataKeyAlreadyExists &dkae)
   {
      throw IManagedClient::IDuplicateKey();
   }
      catch (IBOIMException::IDataKeyNotFound &dknf)
   {
      throw IManagedClient::INoObjectWKey();
   }
   catch (IBOIMException::IDataObjectFailed &dof)
   {
      throw CORBA::PERSIST_STORE(0, CORBA::COMPLETED_NO);
   }
   catch (...)
   {
      throw; // home will handle
   }
   // End Method modifications here
}
```

Required Method - internalizeFromCopyHelper:: This method must handle the exceptions throw by the PO and map them to exceptions that are accepted by the framework.

```
::CORBA::Void PolicyCacheDOImplCache_Impl::internalizeFromCopyHelper(
::IManagedLocal::INonManageable_ptr inCopy)
{  // Insert Method modifications here
   PolicyCopy_var iPolicyCopy = PolicyCopy::_narrow(inCopy);
   long iPolicyNoTemp = iPolicyCopy->policyNo();
   float iAmountTemp = iPolicyCopy->amount();
   float iPremiumTemp = iPolicyCopy->premium();
   iKeyValueSet = 1;
   PolicyCacheDOImplCachePOCopy iPolicyCacheDOImplCachePOCopy;
   iPolicyCacheDOImplCachePOCopy.policyNo(iPolicyNoTemp);
   iPolicyCacheDOImplCachePOCopy.amount(iAmountTemp);
   iPolicyCacheDOImplCachePOCopy.premium(iPremiumTemp);
   try
   {
      iPolicyCacheDOImplCachePO.internalizeFromCopyHelper(
      iPolicyCacheDOImplCachePOCopy);
   }
   catch (IBOIMException::IDataKeyAlreadyExists &dkae)
   {
      throw IManagedClient::IDuplicateKey();
   }
   catch (IBOIMException::IDataKeyNotFound &dknf)
   {
      throw IManagedClient::INoObjectWKey();
   }
   catch (IBOIMException::IDataObjectFailed &dof)
   {
      throw CORBA::PERSIST_STORE(0, CORBA::COMPLETED_NO);
   }
   catch (...)
   {
      throw; // home will handle
   }
   // End Method modifications here
}
```

Required Method - getters:: Many of the methods are required to handle exceptions that may be thrown by the PO and mapping them to exceptions that are supported by the calling framework. For getters the exception that is used as the CORBA standard exception PERSIST_STORE.

```
::CORBA::Long PolicyCacheDOImplCache_Impl::amount()
{
   // Insert Method modifications here
   ::CORBA::Float iAmountTemp;
   try
   {
      iAmountTemp = iPolicyCachePO.id();
   }
   catch (CORBA::UserException &cue)
   {
      throw CORBA::PERSIST_STORE(0, CORBA::COMPLETED_NO);
   }
   return iAmountTemp;
   // End Method modifications here
}
```

Required Method - setters:: Many of the methods are required to handle exceptions that may be thrown by the PO and mapping them to exceptions that are supported by the calling framework. For getters the exception that us used is the CORBA standard exception PERSIST_STORE.

```
::CORBA::Void PolicyCacheDOImplCache_Impl::amount( ::CORBA::Float amount)
{
   // Insert Method modifications here
   ::CORBA::Float iAmountTemp = amount;
   try
   {
      iPolicyCache.amount(iAmountTemp);
   }
   catch (CORBA::UserException &cue)
   {
      throw CORBA::PERSIST_STORE(0, CORBA::COMPLETED_NO);
   }
   markDirty();
   // End Method modifications here
}
```

Required Method - internalizeKeyAttributes:: This method must also handle the exceptions throw by the PO and map them to exceptions that are accepted by the framework.

```
::CORBA::Void PolicyCacheDOImplCache_Impl::internalizeKeyAttributes(
::IIMFLocalToServer::IKeyComponent_ptr keyComp)
{
   // Insert Method modifications here
   iKeyValueSet = 1;
   try
   {
      iPolicyCacheDOImplCachePO.internalizeKeyAttributes(keyComp);
   }
   catch (IBOIMException::IDataKeyAlreadyExists &dkae)
   {
      throw IManagedClient::IDuplicateKey();
   }
   catch (IBOIMException::IDataKeyNotFound &dknf)
   {
      throw IManagedClient::INoObjectWKey();
   }
   catch (IBOIMException::IDataObjectFailed &dof)
   {
      throw CORBA::PERSIST_STORE(0, CORBA::COMPLETED_NO);
   }
   catch (...)
   {
      throw; // home will handle
   }
   // End Method modifications here
}
```

Required Method - externalizeKeyAttributes:: This method must also handle the exceptions throw by the PO and map them to exceptions that are accepted by the framework.

```
::CORBA::Void PolicyCacheDOImplCache_Impl::externalizeKeyAttributes(
::IIMFLocalToServer::IKeyComponent_ptr & keyComp)
{
   // Insert Method modifications here
   try
```

```
      {
         iPolicyNoDOImplCachePO.externalizeKeyAttributes(keyComp);
      }
      catch (CORBA::UserException &cue)
      {
         throw IBOIMException::IExternalizeKeyAttributesFailed();
      }
      // End Method modifications here
   }
```

Required Method - internalizeData:: This method must also handle the exceptions throw by the PO and map them to exceptions that are accepted by the framework but is also much simpler due to the support provided by the PO.

```
::CORBA::Void PolicyCacheDOImplCache_Impl::internalizeData(
const ::IBOIMLocalToServerMetadata::dataSequence & dataSeq)
{
   // Insert Method modifications here
   iPolicyCacheDOImplCachePO.internalizeData(dataSeq[0]);
   iKeyValueSet = 1;
   // End Method modifications here
}
```

Transient Data Object Customization – UUID Key (Production Use)

This section includes the following topics:

- “Interfaces”
- “Implementation” on page 224
- “Additional Considerations” on page 225

Interfaces

The interface view follows. The IBOIMExtLocalToServer::IUUIDDataObject is the interface that should be supported for data object implementations using UUID Keys for Transient Objects.

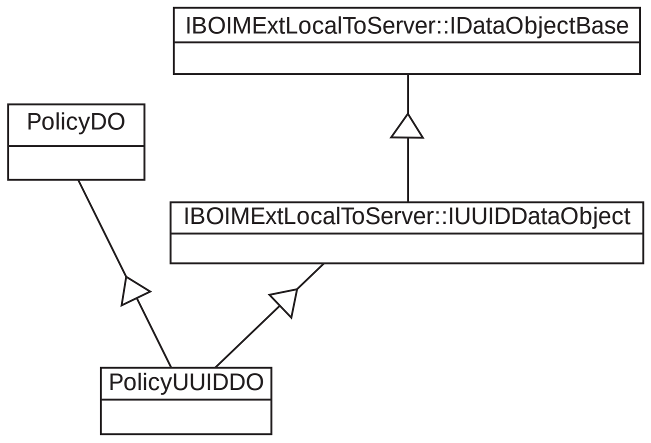

IBOIMExtLocalToServer::IDataObjectBase
PolicyDO
IBOIMExtLocalToServer::IUUIDDataObject
PolicyUUIDDO

Figure 57. UUID Data Object Interface View

The implementation interface follows.

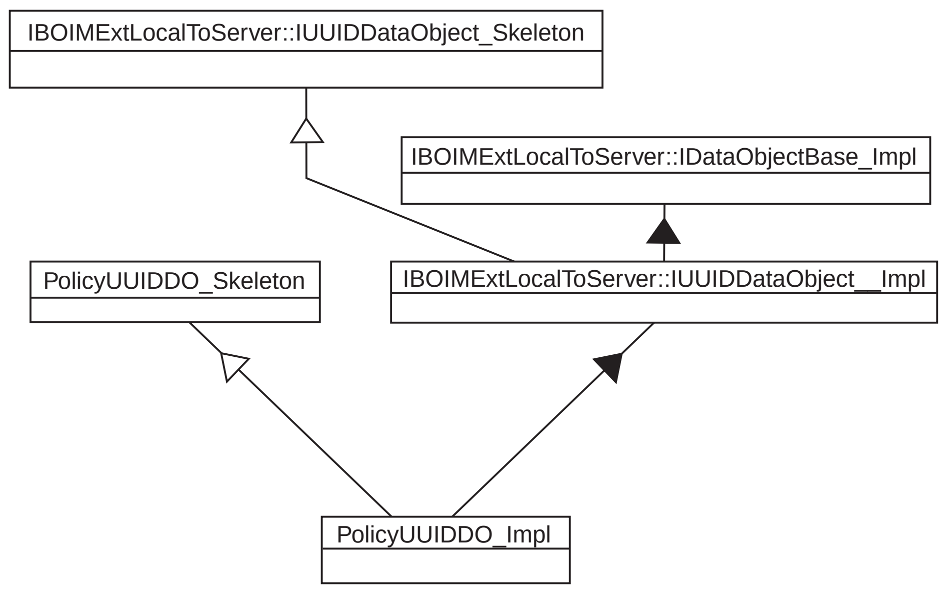

IBOIMExtLocalToServer::IUUIDDataObject_Skeleton
IBOIMExtLocalToServer::IDataObjectBase_Impl
PolicyUUIDDO_Skeleton
IBOIMExtLocalToServer::IUUIDDataObject__Impl
PolicyUUIDDO_Impl

Figure 58. UUID Data Object Implementation Interface Inheritance

Implementation

This section includes the following topics:

- "Framework Required Method – internalizeFromPrimaryKey"
- "Framework Required Method – internalizeFromCopyHelper"
- "Framework Required Code – create() function"
- "Methods To Support Attributes – Getters"
- "Methods to Support Attributes – Setters"
- "Additional Methods – Default Constructor" on page 225
- "Required Method – verifyKey" on page 225
- "Required Method – externalizeKeyAttributes" on page 225
- "Required Method – internalizeKeyAttributes" on page 225
- "Required Method – update" on page 225
- "Required Method – insert" on page 225
- "Required Method – retrieve" on page 225
- "Required Method – del" on page 225

Framework Required Method – internalizeFromPrimaryKey: This method should not be implemented. It is implemented by the framework. It works with a IBOIMExtLocal::IUUIDPrimaryKey.

Framework Required Method – internalizeFromCopyHelper: This method must be implemented. It should copy its attributes and call it's parent's internalizeFromCopyHelper method which copy the UUID primary key value.

Framework Required Code – create() function: See the description in "Framework Required create() function" on page 209. It is the same for UUID based data objects.

Methods To Support Attributes – Getters: See the description in "Methods To Support Attributes – Getters" on page 209.

Methods to Support Attributes – Setters: See the description in "Methods to Support Attributes – Setters" on page 214.

Additional Methods – Default Constructor: See the description in “Additional Methods – Default Constructor” on page 210.

Required Method – verifyKey: This method should not be implemented. The framework takes care of it.

Required Method – externalizeKeyAttributes: This method should not be implemented. The framework takes care of it.

Required Method – internalizeKeyAttributes: This method should not be implemented. The framework takes care of it.

Required Method – update: This method should not be implemented. The framework takes care of it.

Required Method – insert: This method should not be implemented. The framework takes care of it.

Required Method – retrieve: This method should not be implemented. The framework takes care of it.

Required Method – del: This method should not be implemented. The framework takes care of it.

Additional Considerations

Using the UUIDDO and UUIDKey limits the Programming Model for the business objects. FindByPrimaryKeyString is not meaningful because the UUIDKey does not contain business logic attributes. UUID support is most useful when there are short-lived business objects that do not need to be found after they are created.

Transient Data Object – Any Key (Production Use)

If the data object customization is targeting transient data objects that run in a BOIM application adaptor and leverage business object information for making up the key, then you should read this section. In this section, the PolicyDO interface is used as the basis to describe the necessary customization.

BOIM Data Object Interfaces

The IDL-based picture for BOIM data objects follows:

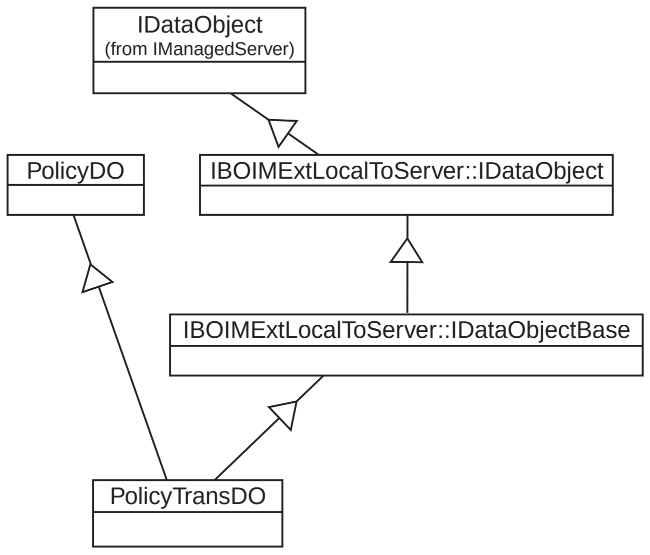

IDataObject
(from IManagedServer)
PolicyDO
IBOIMExtLocalToServer::IDataObject
IBOIMExtLocalToServer::IDataObjectBase
PolicyTransDO

Figure 59. BOIM Data Object IDL Interface View

The example does not show objects capable of being included in queries. The PolicyDO_Impl class implements the PolicyDO interface shown in the previous figure. The implementation interface inheritance picture follows.

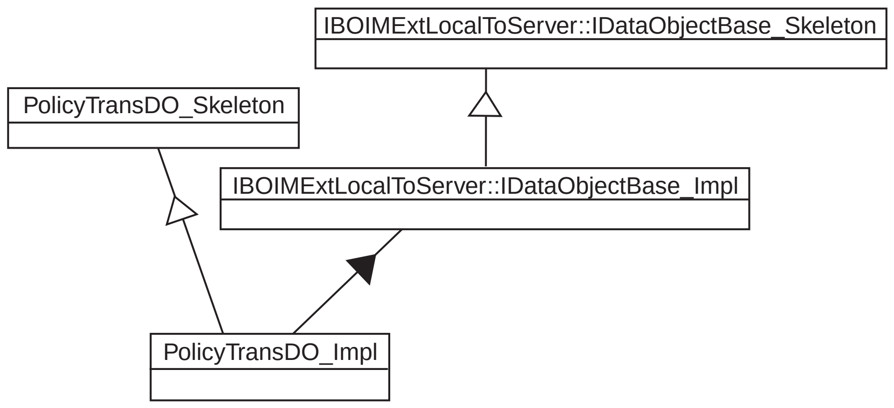

IBOIMExtLocalToServer::IDataObjectBase_Skeleton
PolicyTransDO_Skeleton
IBOIMExtLocalToServer::IDataObjectBase_Impl
PolicyTransDO_Impl

Figure 60. BOIM Data Object Implementation Interface Inheritance

The IBOIMExtLocalToServer::IDataObjectBase_Impl provides an implementation for some of the methods that are described in the IBOIMExtLocalToServer::IDataObject interface. The following methods have implementations which are inherited:

- string connection()
- void connection(string value)
- void markDirty()
- void clearDirty()
- boolean isDirty()
- void updateToDataStore()
- void insertToDataStore()
- void retrieveFromDataStore()
- void deleteFromDataStore()
- initDO()

These methods should not be overridden.

Given this base, the PolicyDO_Impl implementation interface should look like this:

```
#ifdef SOMCBNOLOCALINCLUDES
#include <IBOIMExtLocalToServer>
#include <PolicyTransDOImpl.hh>
#else
#include "IBOIMExtLocalToServer.ih"
#include "PolicyTransDOImpl.hh"
#endif

class PolicyTransDOImpl_Impl : public virtual ::PolicyTransDOImpl_Skeleton,
public virtual IBOIMExtLocalToServer_IDataObjectBase_Impl
{
   public:

   //default constructor doimpl
   PolicyTransDOImpl_Impl();

   ::CORBA::Float amount();
   ::CORBA::Void amount( ::CORBA::Float amount);
   ::CORBA::Long policyNo();
```

```
    ::CORBA::Void policyNo( ::CORBA::Long policyNo);
    ::CORBA::Float premium();
    ::CORBA::Void premium( ::CORBA::Float premium);

    virtual ::CORBA::Void insert();
    virtual ::CORBA::Void update();
    virtual ::CORBA::Void retrieve();
    virtual ::CORBA::Void del();
    virtual ::CORBA::Void internalizeFromPrimaryKey(
                   ::IManagedLocal::IPrimaryKey_ptr inKey);
    virtual ::CORBA::Void internalizeFromCopyHelper(
                   ::IManagedLocal::INonManageable_ptr inCopy);
    virtual ::CORBA::Void internalizeKeyAttributes(
                   ::IIMFLocalToServer::IKeyComponent_ptr keyComp);
    virtual ::CORBA::Void externalizeKeyAttributes(
                   ::IIMFLocalToServer::IKeyComponent_ptr & keyComp);
    virtual ::CORBA::Boolean verifyKey();

    protected:
    private:
    ::CORBA::Float iAmount;
    ::CORBA::Long iPolicyNo;
    ::CORBA::Float iPremium;
    ::CORBA::Boolean iKeyValueSet;
 };
```

BOIM Data Object Implementation

The following methods need to be implemented in order for the BOIM data object to work properly on the server.

Framework Required Method – internalizeFromPrimaryKey: See the description in “Framework Required Method internalizeFromPrimaryKey” on page 208. It is the same for BOIM-based data objects.

In addition, the `keyValueSet` flag should be set to true when valid key values are successfully retrieved from the key.

Framework Required Method – internalizeFromCopyHelper: See the description in “Framework Required Method internalizeFromCopyHelper” on page 209. It is the same for BOIM-based data objects.

In addition, the `iKeyValueSet` flag should be set to true when valid key values are successfully retrieved from the key.

Framework Required Code – create() Function: See the description in “Framework Required create() function” on page 209. It is the same for BOIM based data objects.

Methods To Support Attributes – Getters: See the description in “Methods To Support Attributes – Getters” on page 209. It is the same for BOIM based data objects.

Methods to Support Attributes – Setters: All of the attributes need to have a set method on them. A setter method for one of the attributes of the PolicyDO follows.

```
 ::CORBA::Void PolicyTransDOImpl_Impl::amount( ::CORBA::Float amount)
 {
    // Insert Method modifications here
    iAmount = amount;
    markDirty();
```

```
   // End Method modifications here
}
```

The difference from the transient case is where the markDirty() method is run. This indicates to the application adaptor that it is necessary to update the underlying datastore at prescribed times.

Additional Methods – Default Constructor: The default constructor needs to initialize the attributes and synchronize the DataObject with the PO. This is best done by calling the setters for these attributes. However, a side effect of calling the setters is that the 'dirty' bit is set. Leaving this bit set would cause invalid updates to the datastore when the business object is passivated. To solve this problem the clearDirty() method is called at the end of the method. Also, the iKeyValueSet() attribute is set to a value that indicates the primary key attributes have not yet been set.

```
PolicyEmSQLDOImpl_Impl::PolicyEmSQLDOImpl_Impl()
:iKeyValueSet(0)
{  policyNo(0);
   amount(0);
   premium(0);
   clearDirty();
}
```

Required Method – verifyKey: This method returns a boolean to indicate whether or not the key values in the data object have been acquired and are valid. This method is used by the application adaptor to validate the key before it builds reference data for the object.

```
::CORBA::Boolean PolicyTransDOImpl_Impl::verifyKey()
{
   // Insert Method modifications here
   return iKeyValueSet;
   // End Method modifications here
}
```

Required Method – externalizeKeyAttributes: This method is used by the application adaptor as it creates reference data for the object reference that is being created. The code in this method writes out the key values into the `outKey` that is provided on the parameter list.

```
::CORBA::Void
PolicyTransDOImpl_Impl::externalizeKeyAttributes(
                      ::IIMFLocalToServer::IKeyComponent_ptr & keyComp)
{
   // Insert Method modifications here
   keyComp->write_long(iPolicyNo);
   // End Method modifications here
}
```

Required Method – internalizeKeyAttributes: This method is used by the application adaptor for reactivating objects. This method should pull information out of the `inKey` provided and set the flag indicating that the key is valid to true. Additional logic may be placed in this method to further verify that the key value is good.

```
::CORBA::Void
PolicyTransDOImpl_Impl::internalizeKeyAttributes(
                      ::IIMFLocalToServer::IKeyComponent_ptr keyComp)
{
   // Insert Method modifications here
   iPolicyNo = keyComp->read_long();
   iKeyValueSet = 1;
   // End Method modifications here
}
```

Required Methods – del, insert, retrieve, and update: For true transient objects, the del(), insert(), retrieve(), and update() methods are empty. For situations where the transient object adaptor is being used to manage the object references, but not the persistent state data, these methods contain code that removes, creates, retrieves, or updates a new entry in the persistent store.

Summary of DataObject Customization

Table 5 on page 230 enumerates the general strategy for each data object method or data object method type as it applies to each of the kinds of data object customization that are possible using Component Broker.

Table 5 (Page 1 of 3). Summary of Data Object Customization Methods.

Method	Transient data object (unit test)	Persistent data object Static SQL	Persistent Cache Service data object Cache	Transient (production) - UUID	Transient - Any Key (production)
IManagedServer::IDataObject:: internalizeFromPrimary	Required	Required, uses local cache	Required, talks to Cache Service data object	Implemented by framework	Required, uses local cache
IManagedServer::IDataObject:: internalizeFromCopyHelper	Required	Required, uses local cache	Required, talks to Cache Service data object	Required	Required, uses local cache
_create	Required	Required	Required	Required	Required
Attribute getters	Required	Required, uses local cache	Required, talks to Cache Service data object	Required	Required, uses local cache
Attribute setters	Required	Required, uses markDirty(), uses local cache	Required, talks to Cache Service data object	Required, uses markDirty(), uses local cache	Required, uses markDirty(), uses local cache
Default constructor	Required	Required	Required	Required	Required
IBOIMExtLocalToServer::IDataObject::connection()	n/a	Implement by framework	Implemented by framework	Implemented by framework	Implemented by framework
IBOIMExtLocalToServer::IDataObject:: connection(string)	n/a	Implemented by framework	Implemented by framework	Implemented by framework	Implemented by framework
IBOIMExtLocalToServer::IDataObject::markDirty()	n/a	Implemented by framework	Implemented by framework	Implemented by framework	Implemented by framework
IBOIMExtLocalToServer::IDataObject::isDirty()	n/a	Implemented by framework	Implemented by framework	Implemented by framework	Implemented by framework
IBOIMExtLocalToServer::IDataObject:: updateToDataStore()	n/a	Implemented by framework	Implemented by framework	Implemented by framework	Implemented by framework
IBOIMExtLocalToServer::IDataObject:: insertToDataStore	n/a	Implemented by framework	Implemented by framework	Implemented by framework	Implemented by framework

Table 5 (Page 2 of 3). Summary of Data Object Customization Methods.

Method	**Transient data object (unit test)**	**Persistent data object Static SQL**	**Persistent Cache Service data object Cache**	**Transient (production) - UUID**	**Transient - Any Key (production)**
IBOIMExtLocalToServer::IDataObject:: retrieveFromDataStore	n/a	Implemented by framework	Implemented by framework	Implemented by framework	Implemented by framework
IBOIMExtLocalToServer::IDataObject:: deleteFromDataStore	n/a	Implemented by framework	Implemented by framework	Implemented by framework	Implemented by framework
IBOIMExtLocalToServer::IDataObject::verifyKey()	n/a	Required	Required	Implemented by framework	Required
IBOIMExtLocalToServer::IDataObject:: externalizeKeyAttributes	n/a	Required	Required, talks to Cache Service data object	Implemented by framework	Required
IBOIMExtLocalToServer::IDataObject:: internalizeKeyAttributes	n/a	Required	Required, talks to Cache Service data object	Implemented by framework	Required
IBOIMExtLocalToServer::IDataObject::update()	n/a	Required, has SQL	Required, null implementation Cache Service data object handles	Implemented by framework	Required
IBOIMExtLocalToServer::IDataObject::insert()	n/a	Required, has SQL	Required, null implementation Cache Service data object handles	Implemented by framework	Required
IBOIMExtLocalToServer::IDataObject::retrieve()	n/a	Required, has SQL	Required, null implementation	Implemented by framework	Required
IBOIMExtLocalToServer::IDataObject::del()	n/a	Required, has SQL	Required, does markDelete on Cache Service data object	Implemented by framework	Required
IRDBIMExtLocalToServer::setConnection	n/a	Required	n/a	n/a	n/a

Table 5 (Page 3 of 3). Summary of Data Object Customization Methods.

Method	**Transient data object (unit test)**	**Persistent data object Static SQL**	**Persistent Cache Service data object Cache**	**Transient (production) - UUID**	**Transient - Any Key (production)**
IBOIMExtLocalToServer::IQueryableDataObject::internalizeData()	n/a	Optional	Optional	Optional	Optional
IBOIMExtLocalToServer::IDataObject::initDO()	n/a	Optional	Optional	Optional	Optional

Data Object Data Management Patterns

In the MOFW, decisions are made about how to deal with the data object. This is done when choosing between IManagedServer::IManagedObjectWithCachedDataObject and IManagedServer::IManagedObjectWithDataObject. Programming style influences the choice. Performance and other design considerations also come into play.

Data objects also do a level of "caching." This is not a programming style issue. It is related to the underlying store, or lack thereof, for which the data object is the abstraction that is used by the business object. Transient objects definitely need to have some cache in the data object as that is the only way that they can work correctly. Data objects that are abstractions over persistent storage, however, have choices to make about the caching pattern that is chosen.

Data Object Customization and Inheritance

It is necessary to inherit the parent data object interface. It is not necessary and may not even be desirable to inherit implementation of the base class data object. However, if you are inheriting data object implementation, then parent methods need to be called for internalizeFromPrimaryKey, internalizeFromCopyHelper, verifyKey, and externalizeKeyAttributes. Call the parent method first, before executing the subclass function. Other methods may also need to call parent methods. This varies based upon the kind of data object being used.

The following example assumes that the PolicyEmSQLDOImpl interface is inherited. (See "Transient Data Object – Any Key (Production Use)" on page 225 for details on the BOIM data object type of data object customization.)

CarPolicy BOIM Data Object Interfaces

The IDL-based picture for the CarPolicy BOIM data object follows:

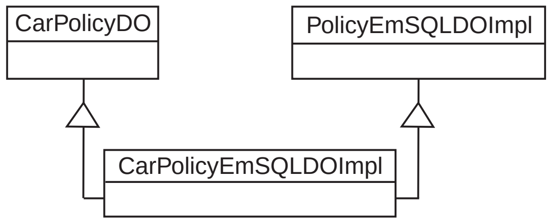


Figure 61. CarPolicy BOIM Data Object IDL Interface View

The PolicyEmSQLDOImpl_Impl class implements the Policy interface as shown in Figure 61. The implementation interface inheritance is shown in Figure 62.

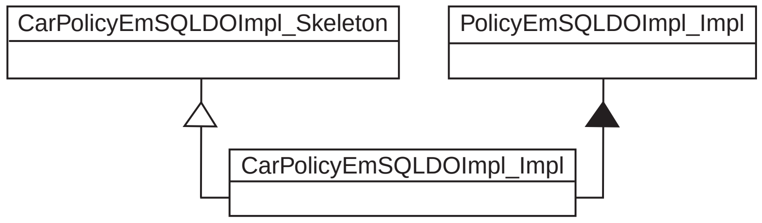


Figure 62. BOIMDO Implementation Interface Inheritance

Given this base, the CarPolicyEmSQLDOImpl_Impl implementation interface should look like this:

```
#include "CarPolicyEmSQLPO.hpp"

#ifdef SOMCBNOLOCALINCLUDES
#include <PolicyEmSQLDOImpl.ih>
#include <CPEmSQLDOImpl.hh>
#else
#include "PolicyEmSQLDOImpl.ih"
#include "CPEmSQLDOImpl.hh"
#endif

class CarPolicyEmSQLDOImpl_Impl : public virtual::CarPolicyEmSQ LDOImpl_Skeleton
,public virtual PolicyEmSQLDOImpl_Impl
{
public:
   //default constructor doimpl
   CarPolicyEmSQLDOImpl_Impl();

   ::CORBA::Long year();
   ::CORBA::Void year( ::CORBA::Long year);
   char* make();
   ::CORBA::Void make(const char* make);
   char* model();
   ::CORBA::Void model(const char* model);
   ::CORBA::Long serialNumber();
   ::CORBA::Void serialNumber( ::CORBA::Long serialNumber);
   ::CORBA::Float collisionDeductible();
   ::CORBA::Void collisionDeductible( ::CORBA::Float collisionDeductible);
   ::CORBA::Boolean glassCoverage();
   ::CORBA::Void glassCoverage( ::CORBA::Boolean glassCoverage);
   ::CORBA::Long policyNo();
   ::CORBA::Void policyNo( ::CORBA::Long policyNo);

   virtual ::CORBA::Void insert();
   virtual ::CORBA::Void update();
   virtual ::CORBA::Void retrieve();
   virtual ::CORBA::Void del();
   virtual ::CORBA::Void setConnection(const char* dataBaseName);
   virtual ::CORBA::Void internalizeFromPrimaryKey(
                  ::IManagedLocal::IPrimaryKey_ptr inKey);
   virtual ::CORBA::Void internalizeFromCopyHelper(
                  ::IManagedLocal::INonManageable_ptr inCopy);
   virtual ::CORBA::Void internalizeKeyAttributes(
                  ::IIMFLocalToServer::IKeyComponent_ptr keyComp);
   virtual ::CORBA::Void externalizeKeyAttributes(
                  ::IIMFLocalToServer::IKeyComponent_ptr & keyComp);
   virtual ::CORBA::Void internalizeData(
                    const ::IBOIMLocalToServerMetadata::dataSequence & dataSeq);
   virtual ::CORBA::Boolean verifyKey();

protected:
   ::CarPolicyEmSQLPO iCarPolicyEmSQLPO;

private:
   ::CORBA::Boolean iKeyValueSet;
};
```

CarPolicy BOIM Data Object Implementation

The following methods need to be implemented for the data object to work properly on the server.

Framework Required Method – internalizeFromPrimaryKey

The `policyNo` attribute needs to be internalized. The `keyValueSet` flag should be set to true when valid key values are successfully retrieved for both attributes.

```
::CORBA::Void
CarPolicyEmSQLDOImpl_Impl::internalizeFromPrimaryKey(::IManagedLocal::IPrimaryKey_ptr inKey)
{
   // Insert Method modifications here
   PolicyKey_var iPolicyKey = PolicyKey::_narrow(inKey);
   long iPolicyNoTemp = iPolicyKey->policyNo();
   iKeyValueSet = 1;
   PolicyEmSQLPOKey iPolicyEmSQLPOKey;
   CarPolicyEmSQLPOKey iCarPolicyEmSQLPOKey;
   iPolicyEmSQLPOKey.policyNo(iPolicyNoTemp);
   iCarPolicyEmSQLPOKey.policyNo(iPolicyNoTemp);

   iPolicyEmSQLPO.internalizeFromPrimaryKey(iPolicyEmSQLPOKey);
   iCarPolicyEmSQLPO.internalizeFromPrimaryKey(iCarPolicyEmSQLPOKey);

   // End Method modifications here
}
```

Framework Required Method – internalizeFromCopyHelper

internalizeFromCopyHelper() should first be called using PolicyEmSQLDOImpl_Impl. Then copy the attributes from the copy helper object to those stored in the CarPolicyEmSQLDOImpl_Impl. In addition, the `keyValueSet` flag should be set to true when valid key values are successfully retrieved from the key.

```
::CORBA::Void
CarPolicyEmSQLDOImpl_Impl::internalizeFromCopyHelper(
                         ::IManagedLocal::INonManageable_ptr inCopy)
{
   // Insert Method modifications here

   CarPolicyCopy_var iCarPolicyCopy = CarPolicyCopy::_narrow(inCopy);
   long iYearTemp = iCarPolicyCopy->year();
   ::CORBA::String_var iMakeTemp = iCarPolicyCopy->make();
   ::CORBA::String_var iModelTemp = iCarPolicyCopy->model();
   long iSerialNumberTemp = iCarPolicyCopy->serialNumber();
   float iCollisionDeductibleTemp = iCarPolicyCopy->collisionDeductible();
   ::CORBA::Boolean iGlassCoverageTemp = iCarPolicyCopy->glassCoverage();
   float iAmountTemp = iCarPolicyCopy->amount();
   long iPolicyNoTemp = iCarPolicyCopy->policyNo();
   float iPremiumTemp = iCarPolicyCopy->premium();
   iKeyValueSet = 1;
   PolicyEmSQLPOCopy iPolicyEmSQLPOCopy;
   CarPolicyEmSQLPOCopy iCarPolicyEmSQLPOCopy;
   iPolicyEmSQLPOCopy.amount(iAmountTemp);
   iPolicyEmSQLPOCopy.policyNo(iPolicyNoTemp);
   iCarPolicyEmSQLPOCopy.policyNo(iPolicyNoTemp);
   iPolicyEmSQLPOCopy.premium(iPremiumTemp);
   iCarPolicyEmSQLPOCopy.year(iYearTemp);
   {
      DB2VarCharMap maketemp;
      CARPOLICYEMSQLPO_MAKEDB2VARCHAR maketemp2;
      maketemp.maxLen = 2000;
      maketemp.dataP = maketemp2.data;
      if (iMakeTemp)
      DB2MappingHelper::stringToVarChar(iMakeTemp, maketemp);
      else
      maketemp.length = 0;
      maketemp2.length = maketemp.length;
      iCarPolicyEmSQLPOCopy.make(maketemp2);
   }
   {
      DB2VarCharMap modeltemp;
      CARPOLICYEMSQLPO_MODELDB2VARCHAR modeltemp2;
      modeltemp.maxLen = 2000;
      modeltemp.dataP = modeltemp2.data;
      if (iModelTemp)
      DB2MappingHelper::stringToVarChar(iModelTemp, modeltemp);
      else
      modeltemp.length = 0;
      modeltemp2.length = modeltemp.length;
      iCarPolicyEmSQLPOCopy.model(modeltemp2);
   }
   iCarPolicyEmSQLPOCopy.serialNumber(iSerialNumberTemp);
   iCarPolicyEmSQLPOCopy.collisionDeductibl(iCollisionDeductibleTemp);
   iCarPolicyEmSQLPOCopy.glassCoverage(iGlassCoverageTemp);
   iPolicyEmSQLPO.internalizeFromCopyHelper(iPolicyEmSQLPOCopy);
   iCarPolicyEmSQLPO.internalizeFromCopyHelper(iCarPolicyEmSQLPOCopy);

   // End Method modifications here
}
```

Framework Required Code – create() Function

See the description in “Framework Required create() function” on page 209. It is the same for this data object.

Methods To Support Attributes – Getters

All of the attributes need to have a getter method for them. Methods that support getting attributes work in the usual way. For example, the getter for make would be:

```
char* CarPolicyEmSQLDOImpl_Impl::make()
{
   // Insert Method modifications here
   ::CORBA::String_var iMakeTemp;
   {
      DB2VarCharMap maketempR;
      maketempR.maxLen = 2000;
      maketempR.length = iCarPolicyEmSQLPO.make().length;
      maketempR.dataP = (char *)iCarPolicyEmSQLPO.make().data;
      iMakeTemp = new char[2001];
      DB2MappingHelper::varCharToString(maketempR, iMakeTemp);
   }
   return CORBA::string_dup(iMakeTemp);
   // End Method modifications here
}
```

Methods to Support Attributes – Setters

All of the attributes need to have a setter method for them. Methods that support setting attributes work in the usual way. For example, the setter for make would be:

```
::CORBA::Void CarPolicy_Impl::make(const char* make)
{
   fMake = make;
   markDirty();
}
```

Additional Methods – Default Constructor

The implementation for this method is similar to other data objects.

```
CarPolicyEmSQLDOImpl_Impl::CarPolicyEmSQLDOImpl_Impl()
:iKeyValueSet(0)
{
   year(0);
   make(CORBA::string_dup(""));
   model(CORBA::string_dup(""));
   serialNumber(0);
   collisionDeductible(0);
   policyNo(0);
   clearDirty();
}
```

Required Method – externalizeKeyAttributes

The implementation for this method is similar to other data objects, except that it first calls `externalizeKeyAttributes` on the parent data object.

```
::CORBA::Void
CarPolicyEmSQLDOImpl_Impl::externalizeKeyAttributes(
      ::IIMFLocalToServer::IKeyComponent_ptr & keyComp)
{
   // Insert Method modifications here
   long iPolicyNoTemp;
   PolicyEmSQLPOKey iPolicyEmSQLPOKey;
   CarPolicyEmSQLPOKey iCarPolicyEmSQLPOKey;
   iPolicyEmSQLPO.externalizeKeyAttributes(iPolicyEmSQLPOKey);
   iCarPolicyEmSQLPO.externalizeKeyAttributes(iCarPolicyEmSQLPOKey);
   iPolicyNoTemp = iPolicyEmSQLPOKey.policyNo();
   iPolicyNoTemp = iCarPolicyEmSQLPOKey.policyNo();
    keyComp->write_long(iPolicyNoTemp);
   // End Method modifications here
}
```

Required Method – internalizeKeyAttributes

The implementation for this method is similar to other data objects, except that it first calls internalizeKeyAttributes on the parent data object. In addition, all keys need to be verified that the values are good.

```
::CORBA::Void
CarPolicyEmSQLDOImpl_Impl::internalizeKeyAttributes(
                         ::IIMFLocalToServer::IKeyComponent_ptr keyComp)
{
   // Insert Method modifications here
   long iPolicyNoTemp = keyComp->read_long();
   iKeyValueSet = 1;
   PolicyEmSQLPOKey iPolicyEmSQLPOKey;
   CarPolicyEmSQLPOKey iCarPolicyEmSQLPOKey;
   iPolicyEmSQLPOKey.policyNo(iPolicyNoTemp);
   iCarPolicyEmSQLPOKey.policyNo(iPolicyNoTemp);
   iPolicyEmSQLPO.internalizeKeyAttributes(iPolicyEmSQLPOKey);
   iCarPolicyEmSQLPO.internalizeKeyAttributes(iCarPolicyEmSQLPOKey);
   // End Method modifications here
}
```

Required Method —del

The del method is called when the object is removed. Both of the POs need to be called in this method.

```
::CORBA::Void CarPolicyEmSQLDOImpl_Impl::del()
{
   // Insert Method modifications here
   iPolicyEmSQLPO.del();
   iCarPolicyEmSQLPO.del();
   // End Method modifications here
}
```

Required Method —insert

The insert is called when a create is done. Both of the POs need to be called in this method.

```
::CORBA::Void CarPolicyEmSQLDOImpl_Impl::insert()
{
   // Insert Method modifications here
   iPolicyEmSQLPO.insert();
   iCarPolicyEmSQLPO.insert();
   // End Method modifications here
}
```

Required Method —retrieve

The retrieve gets data from the database. Both of the POs need to be called in this method.

```
::CORBA::Void CarPolicyEmSQLDOImpl_Impl::retrieve()
{
   // Insert Method modifications here
   iPolicyEmSQLPO.retrieve();
   iCarPolicyEmSQLPO.retrieve();
   // End Method modifications here
}
```

Required Method —update

The update is called when data is put into the database. Both of the POs need to be called in this method.

```
::CORBA::Void CarPolicyEmSQLDOImpl_Impl::update()
{
   // Insert Method modifications here
   iPolicyEmSQLPO.update();
   iCarPolicyEmSQLPO.update();
   // End Method modifications here
}
```

Required Method —setConnection

The setConnection is called when initializing connections. Both of the POs need to be called in this method.

```
::CORBA::Void CarPolicyEmSQLDOImpl_Impl::setConnection(const char* dataBaseName)
{
   // Insert Method modifications here
   iPolicyEmSQLPO.setConnection(dataBaseName);
   iCarPolicyEmSQLPO.setConnection(dataBaseName);
   // End Method modifications here
}
```

Data Object Customization for Cardinality Relations

The data object implementation for a business object that contains a reference to another business object requires getters and setters that are more complex than those that implement attributes with simple mappings to back-end resource managers such as SQL tables. This section describes the reference mapping patterns, what the purpose of each pattern is, and how best to apply the patterns to specific relationship situations.

Top-Down Versus Bottom-Up Relations

The business object interface that contains references does not know how the relationship is implemented. This is a key part of the encapsulation provided by the Managed Object Framework. In this section, implementation strategies and how they appear on the data object implementation are introduced.

There are two basic strategies that are applied to implementing object references which appear in the business object:

Top-down This approach allows alteration or definition of the database schema that underly the class of business object that has references to other business objects.

Bottom-up This approach implies preservation of an existing schema.

While there are times when the object resolution approach characterized as bottom-up can be used for new top-down applications, the inverse is not true. The two strategies apply equally well to cardinality-1 and cardinality-n types of relationships.

Top-Down Customizations

The top-down approach is characterized by the presence of a string in the database table of the containing object. This string contains information which the data object can use to produce the object reference required by the business object interface. Data object getter methods take on the general form of:

```
Claim_ptr PolicyDO_Impl::currentClaim()
{
   // retrieve the stored value for persistent store
   ...
   // convert the retrieved value into a pointer and return it
   ...
}
```

The Data object setter methods take on the general form of:

```
::CORBA::Void PolicyDO_Impl::currentClaim(Claim_ptr claim)
{
   // map the claim pointer into a storable form
   ...
   // invoke and/or notify the application adaptor regarding
   //     the change to persistent data
   ...
}
```

The implementation of these methods is variable in the following dimensions:

- Mapping or conversion design pattern dimension
- Persistent storage dimension

CORBA provides conversion interfaces that assist in the mapping. These interfaces are string_to_object and object_to_string. The string form of an object reference (often called a Stringified Object Reference, or SOR), must stand alone and may be quite large, because it must contain enough information to locate and materialize a remote object. It is, however, a simple way to get the string representation of an object that can then be stored persistently and later used to bring an object reference back to life.

Conceptually, storing a stringified reference sounds reasonable. However, more efficient mechanisms with dimensions that possess different levels of robustness are also possible. These patterns take advantage of abstractions introduced by the managed object framework. The patterns are generally preferable to SORs for the following reasons:

- They require less storage to implement.
- They are based on the applications object model, instead of the objects physical location in the network, making them more robust and allowing them to be maintained more easily than SORs. For example, moving a Home from one container to another would not break the Home/key reference (this pattern is described later), but would break an outstanding SOR.

There are many different combinations of CORBA and Component Broker abstractions that could be used to map object references. The following useful patterns have been identified by the programming model:

Stringified Object Reference
Stores the Stringified Object Reference as a variable length string. It is the simplest form in structure and the largest in size. Multiple references to the same object or other objects in the same application adaptor environment store duplicate prefix data.

Object Name
Stores a Name Service name of an object as a variable length string. This string is much shorter than the SOR. Only objects that are named in the Name Service can be referenced using this pattern.

Home Name and Key
Used for objects that are not named in the Name Service. Instead of the object name, it stores the name of the object home and its primary key within the home. The stored representation for this pattern is a pair of variable length strings: the home name and the stringified primary key.

Queryable Collection Name and Query String

Used for objects that are contained in a queryable home or named collection. It stores a pair of variable length strings: the collection name and a query string that uniquely returns the object.

This is not an exhaustive list. Many other patterns and variants of these patterns are possible. The intent here is to describe some of the more generally applicable patterns.

Bottom-Up Customizations

In the bottom-up case, in addition to knowing that the business object interface has a getter and a setter, there is also a known value in an existing database table or other resource to which the data object is mapped. The value in the existing resource manager is not a mapped object reference as described in "Top-Down Customizations" on page 239. It is most probably a foreign-key, a value that can be used, along with other Component Broker system function, to render the object reference which is being rrquested in the upper level (business object) interface. The methods take on the same form as in the top-down case excepting this additional restriction of the value used to do the mapping.

Conceptually, there are a number of patterns that can be used to map the foreign-key into the object reference and back again:

FindByPrimaryKeyString

Using this pattern, a value (or values) from the underlying resource manager is used to formulate a key. The home for the kind of object being found is determined. The key is used to do a findByPrimaryKeyString on the object. The resulting reference is returned back to the business object. On the setter side, the object reference primary key is extracted and stored for later use when the containing object is again activated and the referred to object is requested. A basic structure is shown in the following segments:

```
Claim_ptr PolicyDO_Impl::currentClaim()
{
   // retrieve the stored foreign key from the PolicySQL table
   // create a claim key
   ClaimKey_var theKey = Claim::_create();
   // set foreign key from table into primary key for claim
   theKey->claimNo(n);
   // find the object via its home
   IMananagedClient::IManageable_var aMgbl;
   aMgbl = claimHome->findByPrimaryKeyString(*theKey->toString());
   return Claim::narrow(aMgbl);
   ...
}


::CORBA::Void PolicyDO_Impl::currentClaim(Claim_ptr claim)
{
   // map the claim pointer into a storable form
   // n is part of a struct used to talk to the database
   n=claim->claimNo;
   // invoke and/or notify the application adaptor regarding
   //      the change to persistent data
   markDirty();
   ...
}
```

The above segments have exception handling and NULL checks removed to allow a more simplified view. They are also incomplete in how they deal with the conditionality aspect of the relationship. If, for example, there is always total ownership of the referenced object, then the setter has to consider removing the referenced object.

The other issue is that of finding the home that is used in the getter method. This is discussed in later sections.

Query

This pattern leverages the query service. In this case, the foreign-key is used as the basis to formulate a query into the table that contains the data for the referred to objects. The result set can then be used to return values to the business object interface. This pattern works for both 1-to-1 and 1-to-n relationships with various exception handling required.

The following segment shows conceptually what happens in the 1-to-1 case:

```
Policy_ptr ClaimDOImpl::policy()
{
   PolicyKey_var iPolicyKey;
   iPolicyKey=PolicyKey::_create();
   // set the foreign key into a policy key
   iPolicyKey->policyNo(iClaimPO.policyNo());
   ::ByteString_var*iPolicyKeyString iPolicyKey->toString();

   // find the home to use
   char buf[128];
   sprintf(buf, "/host/resources/servers/%s/collections/%s", serverName(),
         "Insurance::Policy");
   CORBA::Object_var obj = nameService()->resolve_with_string(buf);
   iPolicyHome=IManagedClient::IHome::_narrow(obj);

   // find the object using the key
   temp = iPolicyHome->findByPrimaryKeyString(*iPolicyKeyString);
   return Policy::_narrow(temp);
}

::CORBA::Void ClaimDOImpl::policy(Policy_ptr policy)
{
   // get the primary key string from the in object
   ::ByteString_var iPolicyKeyString = policy->getPrimaryKeyString();

   // make a key
   PolicyKey_var iPolicyKey = PolicyKey::_create();

   // set string into the key
   iPolicyKey->fromString(*iPolicyKeyString);

   // get number from key and tell PO about it
   iPolicyPolicyNoFK = iPolicyKey->policyNo();
   iClaimPO.policyNo(iPolicyPolicyNK);
```

In the example above, the getter method creates the reference to return by issuing findByPrimaryKeyString with the key that is created from the foreign key value. On the setter, the foreign key determined by examining the key that comes with a reference is extracted and pushed back down to the database. The value for the home Insurance::Policy represents the value that is used by client programs for the factory finder and is part of the management information that is found in the Home Image under the name of managed object interface.

The general patterns that can be used for top-down and bottom-up relationships are described in previous sections. The following sections detail what is done for each specific combination.

Cardinality-1 Relationships

For the top-down case, using the handle-pattern that is supported by Object Builder is recommended. The handle concept of Component Broker encapsulates the actual pattern that is used to store the object reference. From this perspective, each 1-to-1 relationship in the top-down case stores a handle. Using the handle pattern requires that some decisions be made about the nature of the handle.

When a business object that will be referred to by other business objects is constructed, the handle-pattern that it supports must be decided. If no choice is made, then the handle implementation that encapsulates a stringified object reference is the default. This means that it is essential that the Stringified Object Reference pattern be selected on Object Builder whenever constructing relationships to this business object from other business objects. If, when constructing the referenced business object, the managedObjectName or Home Name and Key handles are selected, then the corresponding mapping pattern must be used when making relationships to these objects.

The Home Name and Key pattern should be used when appropriate. This allows creation of database tables that hold shorter handles than is possible with the stringified reference version of handles. For example, the Home Name and Key can generally be stored in 200 bytes (plus or minus 20 bytes) but the SOR version of handles can require more than 1000 bytes.

An optimization for this pattern that can be used when the object being linked is always in a specific known home, is to store only the key string in the database and to hard code the home name (for example, with an installation-time settable environment variable) in the access methods. This reduces the storage overhead per link tremendously but can be used only to link to objects in the same home. Attempts to set the link to point to an object of the same type in another home cannot be statically checked and would therefore need to fail at run time.

Alternatively, a factory finder can be used to implement this pattern. This variant of the pattern also eliminates the home name from the stored representation by using a factory finder instead of the Name Service to retrieve the home object in the get method. This approach can be used only if the factory finder can be guaranteed to always return the same home object (for example, when there is known to be one and only one home for a certain managed object type).

The following figure summarizes what happens at the object (top of figure) and database level (bottom of figure) to make this relationship happen.

For the bottom-up case, Object Builder supports the foreign key pattern suggested and described previously. This is the recommended pattern.

The following figure summarizes what happens at the object (top of figure) and database level (bottom of figure) to make this relationship happen.

Cardinality-N Relationships

For the top-down case, two patterns are supported by Object Builder for Cardinality-N relationships. Conceptually they involve either keeping a reference collection of the object relationships or resolving the object relationships using the query service.

The reference collection method is useful if there is no query predicate from which a results set can be calculated. This option creates a reference collection of object relationships. A handle to this reference collection is stored in the business object that contains the reference to the objects. This reference collection is then accessed whenever the members of the object relationship are used.

The query solution for 1-to-M (described following) is also useful in some top-down scenarios.

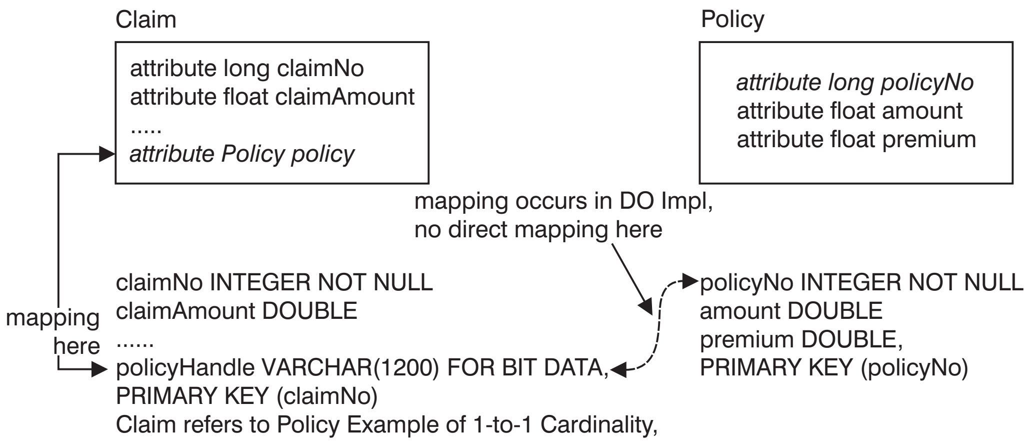


Figure 63. Claim refers to Policy Example of 1-to-1 Cardinality (Top Down)

For the bottom-up case, the solution for 1-to-M relationships is to use the query service. Objects are retrieved using a code segment similar to the following:

```
::IManagedCollections::IIterator_ptr PolicyDOImple::myClaims()
{
   char buf[1024];
   sprintf(buf2, "/host/resources/servers/%s/query-evaluators/default",
       serverName());
   ::CORBA::Object_var obj = nameService()->resolve_with_string(buf2);
   IExtendedQuery::QueryEvaluator_var qe = IExtendedQuery::QueryEvalutor::_narrow(obj);
   ICollectionsBase::IIterator_ptr qIter;
   IExtendedQuery::MemberList_var members;
   char buf[1024];
   sprintf(bufm,"select a from %s a where
       a.\policy\"..policyNo=d%;","Insurance::Policy",policyNo());
   qe->evaluate_to_iterator(buf,NULL,NULL,NULL,0,members,qIter);
   return qIter;
}
```

Object Builder currently supports a pattern similar to this. The difference is that the code to do the query is actually located in the business object implementation. This is a tactical statement. Either way, the general concept of the query code being used to resolve the relationship is the same.

This option allows query to be used to determine membership. This is required in cases where there are other legacy programs that could affect membership in any given relationship. This also has the advantage of using less storage in the business object that has a relationship to many other objects of a different type.

Summarizing Relationships Implementations

The patterns described above are summarized in the following table.

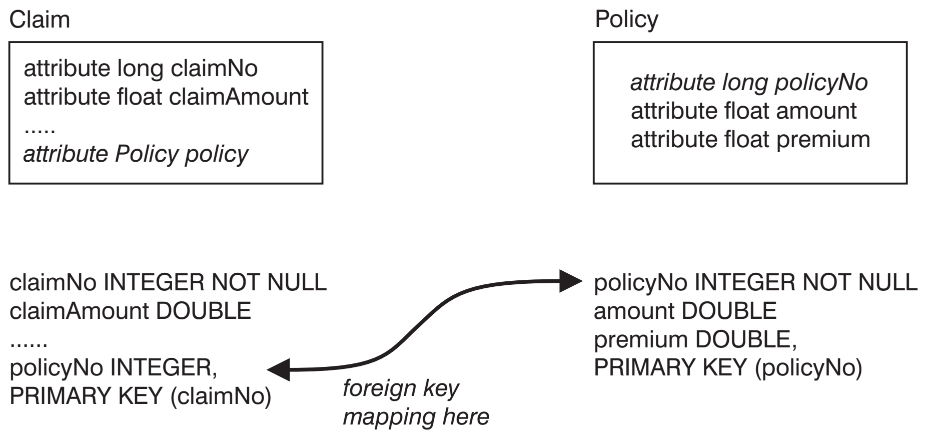


Figure 64. Claim refers to Policy Example of 1-to-1 Cardinality (Bottom Up)

	Top-Down	**Bottom-Up**
Cardinality-1	Store the reference • Uses handles support to store a stringified version of the object reference	Foreign key • Uses findByPrimaryKeyString to locate referenced object
Cardinality-n	Reference collection • Uses handles support to store a reference to a collection that contains the "object relationships"	Foreign key • Uses query service to return the "object relationships" • Requires a 1-to1 reference to implement the other way

Additional Customizations

Additional Data Object Customizations are possible. If the customization options that are optimal for a given relationship are not supported directly, it is possible to alter the mapping patterns that are used in such a way that Object Builder round-tripping is still preserved. This is done through the use of a mapping helper.

Mapping Helpers

A mapping helper is a class that contains mapping methods. Mapping methods provide the conversion between the attribute types of the two objects. You can either use the mapping helpers provided by Object Builder, or you can define your own. Object Builder provides the default mapping helper (the class and its methods) in the following cases:

- When a Stringified Object Reference (SOR) of the data object is mapped to a persistent object attribute of type VARCHAR.
- When a data object attribute of type string is mapped to a persistent object attribute of type VARCHAR. A data object attribute of type string is normally mapped to a persistent object attribute of C++ string type, for example, a string of length 20 is mapped to char[21].

Object Builder does not provide the default mapping between complex data types (any, wchar, and wstring and types defined as constructs, that include typedefs, structures, and unions) and DB2 database types. You must provide your own helper class for these mappings.

You cannot use a mapping helper to map many data object attributes to either one or many persistent object attributes; however you can use one to map many data object attributes to one persistent object attribute. Refer to *Component Broker for Windows NT and AIX Online Documentation* and *Component Broker Application Development Tools* for further information on mapping helpers.

Example Usage of a Mapping Helper

Object Builder integrates the usage of Mapping Helpers into the Data Object implementation files as part of the customization of the application. Following is an example of a VARCHAR-to-string conversion:

```
::CORBA::String_var iAStringTemp = aString;
{
   DB2VarCharMap aStringtemp;
   MAPPINGHELPERPO_ASTRINGDB2VARCHAR aStringtemp2;
   aStringtemp.maxLen = 10;
   aStringtemp.dataP = aStringtemp2.data;
   if (iAStringTemp)
       DB2MappingHelper::stringToVarChar(iAStringTemp, aStringtemp);
   else
      aStringtemp.length = 0;
   aStringtemp2.length = aStringtemp.length;
   iMappingHelperPO.aString(aStringtemp2);
}
```

Expanding the Client Programming Interface

In Chapter 4, “MOFW Client Programming Model” on page 33 there was one interface described that could be used to access the business logic functions of a managed object. That is the easiest way to get started and should be used when possible. Object providers, when they are building business objects, need to subclass from the appropriate interfaces in MOFW and follow a set of rules for building business objects. The client interface that has been discussed so far is the one that is the first subclass of IManageable or one of its Component Broker Frameworks provided dependents. The figure below shows this interface.

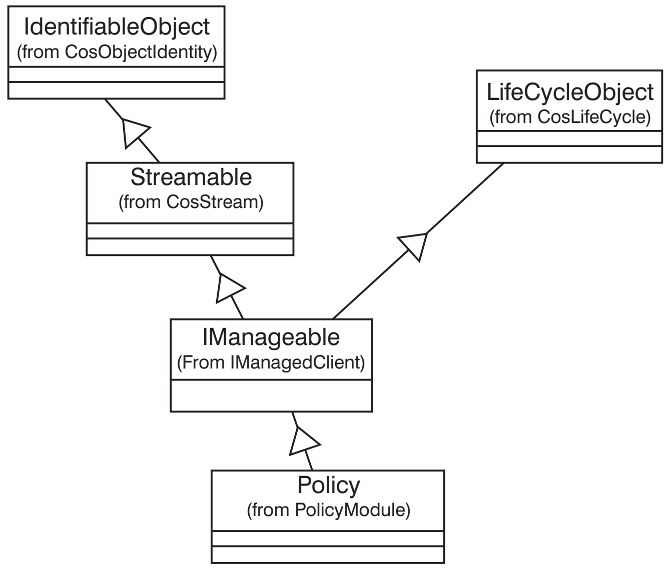


Figure 65. Primary Client Programming Interface

This figure shows the interface of a managed object. There is much to this interface that is described in other chapters. The important thing to recognize here is that the Policy interface introduces the business logic methods and provides access to other methods that can be useful.

Quality of Service Interfaces

Each business object that is constructed by the object provider is installed and run in a particular application adaptor or container. These are Component Broker server concepts that deal with how resource managers are used beneath the MOFW. The MOFW provides a consistent programming model for clients and for object providers, encapsulating the details of the underlying resource managers when possible and practical. However, there are cases where accessing the additional capabilities afforded by a particular implementation of the Component Broker server application adaptor may be desirable. To accommodate this, a different client programming interface is available to you. The intent is to call this the client *Quality Of Service* (QOS) interface. Some clients are considered to be “friends” and might have access to additional methods. Friends are probably other business objects playing the role of clients.

The following interfaces fall into this category:

- IBOIMManagedObjectQOS::IMMixin
- IBOIMManagedObjectFriendQOS::IMMixin

These interfaces collect a number of interfaces which are supported by the BOIM Container. IBOIMManagedObjectQOS::IMMixin brings in CosTransactions::TransactionalObject; IBOIMManagedObjectFriendQOS::IMMixin brings in the checkpointToDataStore() and refreshFromDataStore() methods.

Normally, a client program deals with a client interface by specifying it when the object is found in the IHome, retrieving from some other collection, creating, or reviving from a stringified object reference.

```
Policy *myPolicy;
ByteString *theKeyString;
IManagedClient::IManageable           _ptr          mptr;
IManagedClient::IHome_var myPolicyHome;

// Get the primary key (string) of the object in "theKeyString"
// Get the home configured for Policy objects in "myPolicyHome"

mptr = myPolicyHome->findByPrimaryKeyString(*theKeyString);
myPolicy = Policy::_narrow(mptr);

// invoke business logic or IManageable methods on "myPolicy"
myPolicy->amount(25000.00);
```

Narrow the QOS interface to access specific methods introduced by that interface:

```
IBOIMManagedObjectFriendQOS::IMMixin_ptr myPolicyAsMOFQOS;
ByteString *theKeyString;
IManagedClient::IManageable _ptr mptr;
IManagedClient::IHome_var myPolicyHome;

// Get the primary key (string) of the object in "theKeyString"
// Get the home configured for Policy objects in "myPolicyHome"
mptr = myPolicyHome->findByPrimaryKeyString(*theKeyString);
myPolicyAsMOFQOS =
   IBOIMManagedObjectFriendQOS::IMMixin::_narrow(mptr)

// now use a method from this quality of service interface
```

```
myPolicyAsMOFWQOS->checkpointToDatastore();
```

The checkpointToDatastore() method is used as an example of a method that is available on the QOS interface. Exactly which methods are available on this interface depends on the particular Component Broker Installation and the container in which the managed object resides.

If you already have an object reference to a Policy and want to get at the QOS interface, use the following code segment to narrow it down.

```
myPolicy->....; // run some business logic
myPolicy->....; // run some more business logic

// Now ManagedObjectFriendQOS capability is needed.
myPolicyAsMOFQOS =
   BOIMManagedObjectFriendQOS::IMMixin::_narrow(mypolicy);
// now use a method from this quality of service interface
myPolicyAsMOFWQOS->checkpointToDatastore();
```

While the previous code-segments are understandable, they leave you saddled with the responsibility of maintaining two references to a single managed object. The first reference, myPolicy, has the business logic and IManageable interfaces, while the second, myPolicyAsMOFQOS, has the special QOS interface that relates to the quality of service available on this particular type of managed object.

An interface that combines the QOS quality of service interface with the business logic interface can be introduced. This makes it possible for clients to code to one interface for the duration of an application. A code segment of the IDL follows:

```
#include <IBOIMManagedObjectFriendsQOS.idl>
#include <Policy.idl>

interface PolicyMOFQOS: Policy, IBOIMManagedObjectFriendQOS::IMMixin
{
}
```

If the PolicyMOFQOS interface exists and is usable, then the previous code segments can be simplified as follows:

```
PolicyMOFQOS_ptr myPolicy;
ByteString *theKeyString;
IManagedClient::IManageable _ptr mptr;
IManagedClient::IHome_var myPolicyHome;

// Get the primary key (string) of the object in "theKeyString"
// Get the home configured for Policy objects in "myPolicyHome"

mptr = myPolicyHome->findByPrimaryKeyString(*theKeyString);
myPolicy = PolicyMOFQOS::_narrow(mptr);

// invoke business logic or IManageable methods on "myPolicy"
myPolicy->amount(25000.00);

// invoke any ManagedObjectFriendQOS methods that you want to ...
myPolicy->checkpointToDatastore();
```

You should make a decision for your applications and remain consistent. If you expect to use QOS interfaces rarely, then choose the option implied by the first set of code segments. This has the advantage of letting the application code be as independent as possible from the underlying implementation of the business objects. Only surfacing the QOS type when it is needed means that changes to the methods supported on this interface have minimal impacts on the application code. This interface should not often change. One reason to change is the movement of a particular business object from one container to another. If application code is dependent on a specific container's quality of service interface, it must change. The design challenge is to minimize and isolate these usages to be able to deal with change easily when it occurs.

The advantage of using the combined interface (PolicyMOFQOS in the example) exclusively is one of housekeeping and simplicity. When declared, it provides complete access to all methods associated with the object. Narrowing or casting from one to the other and keeping track of two sets of references to the same object is not necessary.

Using QOS Interfaces for Non-transactional Support

The server supports the concept of transactions in a number of ways. Containers are configured so that all objects within them have the same transactional semantics. Besides the variations that leverage the transaction service, one option exists in which there is no transactional capabilities.

In cases where objects live in a container that does not use transactions, use the following general programming model:

1. Find or create objects.
2. Make changes to those objects.
3. Use QOS interface to checkpointToDataStore.
4. If you want a refresh, then do refreshFromDataStore.

Assembling the Pieces

This section describes how to pull together all of the pieces and package them for installation on the server or various Component Broker clients.

This section recommends a packaging strategy for managed objects. Alternate packaging possibilities exist. The Policy example is used here as the example.

Packaging for Client and Server (VA C++)

This section describes packaging recommendations for business objects.

DLL Packaging

The Component Broker server uses DLLs created with the IBM Visual-Age C++ compiler. DLLs that represent business object implementations are dynamically loaded by the server.

The general strategy for business object DLLs is to have two DLLs. One of the DLLs contains items needed on clients (that is, clients running Visual-Age C++) and on the server, while a separate DLL is created for the business object implementation on the server. The following example packages only one business object into the DLL, although multiple business objects can be put into each DLL. For simplicity, the example uses only Policy. There is a DLL/LIB for both client and server named `policyc` and a server-side DLL/LIB named `policies`.

The following table shows the .o files that must be placed into one of the two LIBs that are created.

Note: In this book, ".o" means that it is an object file targeted for a LIB/DLL. An ".obj" is something that is targeted for an .exe. For example, a client.cpp might generate a client.obj that would go into a client.exe while all of the components associated with a managed object are .o's targeted for LIBs/DLLs.

Table 6. Object Files Targetted for a LIB or DLL

Object File	LIB/DLL	Notes
PolicyKey_C	policyc	Bindings for the key class.
PolicyKey_I	policyc	Implementation for the key class.
PolicyCopy_C	policyc	Bindings for the copy helper.
PolicyCopy_I	policyc	Implementation for the copy helper.
Policy_S	policyc	Bindings to be used by clients for the business object. **Note:** This is an _S and not an _C so that the same LIB/DLL can work on both the client and the server. The _S has all the function of the _C, plus the ability to dispatch on the server. The overhead of the _S versus the _C is minimal.
PolicyBO_S	policys	Bindings for business object implementation.
PolicyBO_I	policys	Implementation of business logic.
PolicyDO_C	policys	Data Object interface bindings.
PolicyDOBOIM_C	policys	Data Object implementation bindings.
PolicyDOBOIM_I	policys	Data Object implementation code.
PolicyMO_S	policys	Manage Object bindings.
PolicyMO_I	policys	Managed Object implementation.

Client programs should be able to link the policy.lib into their programs (.exe or .dll) and successfully communicate with business objects on the server. Business objects that wish to directly use references to policies should link to policy.lib.

See systems management and application installation information about distributing and installing these DLL files on clients and on servers.

Create Functions for Dynamic DLL Loading

The server dynamically loads the DLLs that contain the business object function. Each business object is outfitted with a set of external functions that allow the server to create the various pieces of a managed object. The following table summarizes the requirement for these create functions.

Table 7. Function Summmary

Class	Function Name	Implementation	Return Type
DataObject_Imple (for example, Policy_Impl)	Policy_create	return new Policy_Impl();	IBOIMManagedObjectCustom-ization::IDataObject_ptr
ManagedObject_Impl (for example, PolicyMO_Impl)	PolicyMO_create	return new PolicyMO_Impl();	IManagedServer:: ImangedObject_ptr
Key_Impl (for example, PolicyKey_Impl)	PolicyKey_create	return PolicyKey::_create();	IManagedLocal:: IPrimaryKey_ptr
CopyHelper_Impl (for example, PolicyCopy_Impl)	PolicyCopy_create	return PolicyCopy::_create	IManagedLocal:: INonManageable_ptr

The general form of these functions is:

```
extern "C"
{
    __declspec(dllexport) return type functionname()
    {
        return implementation;
     }
}
```

Notice that the two shaded boxes for the local-only objects are using the same _create function that is used by clients when they are making one of these objects. The others are made only by the server and thus do not need to have similar _create methods. Do not get the _create methods that some local-only objects have confused with the create functions that are used by the server.

Exposing Interfaces to Clients

Given the .LIB files shown previously, the next step is to list and determine which interfaces are needed by clients of the policy class. Pure clients need access to the following interfaces:

PolicyKey.hh	To create and use the key class.
PolicyCopy.hh	To create and use the copy helper class.
Policy.hh	To interact with Policy objects that are on the server.

Exposing Interfaces to Business Object Builders

If an object provider wishes to have a reference to a business object from another business object, then the interfaces described previously are sufficient. If however, the business object provider wants to extend or override the business object and create a new business object as described in “Extending a Business Object” on page 105, then some additional interfaces are necessary. These are listed next:

- All of the IDL files associated with the Policy would be needed to make a subclass of policy.
- By implication of the previous item, all of the .hh files associated with Policy would be needed to make the bindings for the new class.
- Furthermore, all of the .ih files associated with Policy would be needed if the implementation of the new business object was to make full use of the implementation of Policy.

Packaging the DLL for the ActiveX Visual C++ Client

Packaging for a pure Visual C++ client could be as simple as taking the policyc DLL and rebuilding it for the Microsoft Visual C++ client, and providing the same interfaces to clients. For a minimum footprint client, the Policy_S could be replaced with Policy_C as the Microsoft version of policyc is not used on the server.

To get from a pure Microsoft Visual C++ client to a full ActiveX/COM enabled client, COM wrappers must be created. Run the idl2com tool on those interfaces that are to be exposed to the ActiveX/COM programmer. The idl2com tool produces a series of files that include all of the source code necessary for the COM wrapper for a particular IDL file, as well as a makefile for building a DLL which contains the COM wrapper.

By default, the makefile compiles all COM wrapper source, and all _C files needed by the COM wrapper. Because it is possible that any method inherited by Policy from other classes could be invoked by a COM wrapper user, all _C files for any inherited classes are also compiled into the ActiveX/COM wrapper DLL. This allows the COM wrapper to expose all methods/attributes defined by Policy, as well as those which are inherited from other classes. Once this DLL is built, it must be registered in the Windows system registry, so that the DLL that implements the COM wrapper can be found. Running regsvr32 dllname.dll against the produced DLL, when it has been placed into its final destination directory, accomplishs this task.

A set of basic LIBs must be linked to in order to get the client DLLs or EXEs made properly. This list includes:

- somororm.lib - ORB
- sompmcim.lib - Programming Model
- somosa1m.lib - object services
- somax00m.lib (the ActiveX client run time)

Packaging the Java Client Code

Packaging for the Java Client involves taking the IDL for Policy, PolicyKey and PolicyCopy and running them through the IDL–to–Java emitter. This generates the necessary bindings. These bindings can then be packaged into a .zip file or exist as .class files in the CLASSPATH of the clients.

Enabling Additional Clients

Component Broker provides special support for the clients listed in the previous sections. For additional clients such as Orbix, the appropriate set of Component Broker IDL files would have to be run through the emitter of the client ORB. Then the IDL files for the business object (policy) would need to be run through the emitter of the client ORB.

Note that clients not provided by IBM do not have facilities for handling INonManageable Objects. This means that alternative means of client access are necessary. Using Stringified Object references to business objects, creating specialized homes or wrappering homes appear as the most preferable options for providing access to other clients.

The Local-Only Development Process

The local-only development process includes the following topics:

- "C++ Local-Only"
- "Java Local-Only" on page 254

C++ Local-Only

This development process applies to all keys and copy helpers. The basic process is as follows:

1. Create the IDL file inheriting from the base class specified by the programming model. This might be IManangedLocal::IPrimaryKey (for a primary key) or IManagedLocal::INonManageable (for a copy helper) or some other base class.
2. Introduce the appropriate #pragma to tell the tools and the world that the bindings that are to be generated do not need to contain things that assist CORBA objects in becoming remote. These are objects that are addressed from within the process. The #pragma to be used is: #pragma meta <interface name> localonly. The following example code shows how to do this for a PolicyPrimaryKey interface:

```
#ifndef PolicyPrimaryKey_idl
#define PolicyPrimaryKey_idl
#include <ILocalOnly.idl>
interface PolicyPrimaryKey : IManagedLocal::IPrimaryKey {
#pragma meta PolicyPrimaryKey localonly /* This makes it generate local only bindings */

      /* put your attributes methods here... */
      attribute long policyNo;
};
```

Note: There is an abstract keyword supported by the emitters as well. When combined with local-only, this keyword suppresses the generation of the _create() method declaration and implementation stub. The abstract keyword should not be used for key and helper interfaces that are to have an implementation. It should only be used for abstract interfaces such as the case where you would have a PolicyKeyBase and a couple of subclasses of this for various specific keys. Do not use abstract when you plan to implement the notifies at the same level at which it is introduced.

3. Run the emitter against the local-only idl file.

```
idlc -s"uc;hh;ih;ic"
```

This generates the following files:

- PolicyPrimaryKey.hh - usage bindings containing PolicyPrimaryKey and PolicyPrimaryKey_Skeleton class. The PolicyPrimaryKey class introduces the PolicyPrimarKey::_create() interface that is implemented later.
- PolicyPrimaryKey_C.cpp - implementation of client side usage bindings
- PolicyPrimaryKey.ih - implementation class interface
- PolicyPrimaryKey_I.cpp - implementation class stubs. Included in this file is the method classname::_create(); which should be implemented to new up and return an instance of the proper _Impl class (for example, PolicyPrimaryKey_Impl).

Table 8 contains a summary of the artifacts involved in this activity.

Table 8 (Page 1 of 2). Artifacts of the Local-only Development Process

Artifact Name (across) ArtifactType (down)	**Non-Remoteable Abstration**
IDL Put these under configuration management control.	Abstraction.idl (Use Local-Only and Abstract #pragmas to indicate if this is a regular local-only or an abstract local-only).
.hh (language usage binding) Never modify these – have the makefile generate them.	Abstraction.hh (contains the Abstraction_Skeleton class and the Abstraction C++ class). This also contains the definition of _create() if the local-only #pragma is used.
_C.cpp (client side binding) Never modify these – have the makefile generate them.	Abstraction_C.cpp (The #pragma ensures that only the necessary stuff is in here, no remotable and dispatcher-related pieces).
.ih (implementation interface – emit once or enter manually).	Abstraction.ih (defines Abstraction_Impl that inherits only from Abstration_Skeleton.
_I.cpp (implementation code – emit once or enter manually).	Abstraction_I.cpp (implementation of real logic – the code). When the local-only #pragma is used, this file contains an implementation of the Abstraction::_create() method.
_C.obj (from _C.cpp – need rule in makefile).	Abstraction_C.o (binding used by clients).

Table 8 (Page 2 of 2). Artifacts of the Local-only Development Process

Artifact Name (across) ArtifactType (down)	**Non-Remoteable Abstration**
_I.o (place on servers) (from _I.cpp – need rule in makefile).	Abstraction_I.o
Abstraction.LIB (group one more abstraction into a LIB).	Abstraction_I.o and Abstraction_C.o can get packaged together into one LIB that goes on both the client (C++) and the server. See “Packaging for Client and Server (VA C++)” on page 249 for more information on how to assemble all of the pieces needed for a managed object.

4. Create a Makefile - Use Table 8 on page 253 to determine proper packaging strategy.
5. Write the Code - Use the section in this book that maps to the kind of local-only object being constructed. The implementation of the code is in the _I.cpp file with the definition of any private variables or methods being done in the .ih file.
6. Run `make` from the environment with the correct paths set
7. Test and debug.
8. Install.

Java Local-Only

Most of the process is the same for Java. The emitter is IDL-to-Java. The results are .class files. Final packaging might involve .zip or .jar files that go onto the Web server for downloading.

Configuring Managed Objects into Servers

This section is intended to describe what is necessary to take the artifacts that are created by the application development process and properly configure them to run on Component Broker servers.

The container is defined as the qualities of service that are provided to managed objects. The BOIM application adaptor provides a single container type that can be configured to provide various qualities of service. The configuration values are described.

Memory Management

There is a classical trade off between performance and space with respect to having objects preloaded into memory. The BOIM and PAA application adaptors have a configuration attribute on the container that allows an application to control this trade off. This attribute is called the memory management attribute, and it is expressed in terms of when an object can be taken from memory.

Those objects that will be used constantly should be placed in a container that has a memory management policy to never passivate. Those objects that are used in transactions should be placed in a container with the memory management policy of passivate at the end of transaction. These objects that are used in sessions should be placed in a container with the memory management policy set to passivate at end of session. Those objects that are used where a client is controlling the synchronization of the object and the back end store (using the checkpointToDatastore and refreshFromDatastore methods) should be placed in a container that has a memory management policy to passivate after checkpoint.

Synchronizing with the Back End

The application adaptor controls the synchronization of the data, state or objects (with the data or state stored in back end stores). The PAA and BOIM application adaptors have many attributes in the configuration of containers that will alter the schemes used to control this synchronization. These attributes are dataCachedInManagedObject, dataCachedInDataObject, useCachingService, terminationPolicy, defaultTransactionPolicy, and sessionPolicy.

Persistence

An object has two dimensions to persistence, persistent state and persistent references. While both of these dimensions refer to the durability, persistent state affects the business object while persistent references affects the user of a business object. All objects are persistent with the exception of the UUID objects.

For the UUID objects their references are either persistent or transient depending on the duration of the usage of the reference. If the reference is expected by the client to always be valid, then the reference is persistent. If however, once the object is passivated then the reference is not expected to be valid any longer, then the reference is transient.

Behavior in the Absence of a Transaction or Session

The BOIM AA and PAA provide the capability of putting UUID objects and persistent objects in the same container. For this to work then the defaultTransactionalPolicy or the sessionPolicy must be set to ignoreCondition.

Some applications may desire a transaction to last for the duration of a method. BOIM application adaptor provides this capability by starting a transaction prior to dispatching the method on an object, and committing the transaction after the method has completed. For this behavior the defaultTransactionPolicy needs to be set to Atomic.

Summary of Configuration Options on Container

Data Object Type	Memory Management Policy	Synchronization Policy	Persistence vs. Transient	Include UUID, roll your own, or objects in the same container
Static embedded SQL for DB2	Passivate at end of transaction	noSession, Data is Cached in Data Object	persistent references and persistent objects	default transaction policy set to ignore condition
DB2 with Caching Service	Passivate at end of transaction	noSession, usesCachingService	persistent references and persistent objects	default transaction policy set to ignore condition
CICS and IMS	Passivate at end of session	noTransaction, Data is not Cached in Data Object	persistent references and persistent objects	default transaction policy set to ignore condition
Oracle with Caching Service	Passivate at end of transaction	noSession	persistent references and persistent objects	default transaction policy set to ignore condition
Roll your own passivation	neverPassivate or Passivate after checkpoint	noTransaction, noSession persistent refs	persistent references and persistent objects	
UUID - transient object refs	1. neverPassivate or 2. noTransaction and passivate at end of transaction	N/A	not persistent objects and not persistent refs	
UUID - persistent object refs	1. neverPassivate or 2. noTransaction and passivate at end of transaction		not persistent objects and persistent refs	

Configuring Application Adaptors – RDB

Business objects run in containers as described previously. Containers are part of the bigger component of Component Broker known as application adaptors. Each application adaptor provides a different quality of service. This quality of service manifests itself in a number of ways, one of which is through the containers that are surfaced by any given adaptor. This section enumerates specifics about the relational database adaptor provided by Component Broker.

Updating the Database Manager Configuration for the Transaction Processor Monitor

To use Object Transaction Service (OTS) with DB2, the transaction processor monitor must be configured with the Transaction Service DLL. To perform the configuration, the following should be typed at a DB2 prompt:

```
update database manager configuration using TP_MON_NAME somtrx1i
```

The request updates the configuration for the transaction processor monitor. The change does not take effect until the database is stopped and started. To validate that the configuration has been updated, type the following at a DB2 prompt:

```
get dbm cfg
```

Look at the transaction processor monitor name to verify that it shows as `somtrxli`.

Starting the Database before Executing an Application

If the application interacts with DB2, the database manager must be started before the application is run. Either:

```
DB2Start
```

or

```
start database manager
```

should be typed at a DB2 command line. Once this is done, the application can communicate with DB2.

Configuring Homes

Homes that can be queried and iterated can be configured for managed objects that are using embedded SQL data objects or for data objects that use the caching service. Data objects that use any other data object cannot be configured into homes that can be queried or iterated, or into specialized homes that can be queried or iterated.

The four default homes that are provided by Component Broker are as follows:

BOIMHomeOfNotRegHomes This home is used for system objects and should not be used for business objects. Objects that are stored in this home are not registered in the Name Space and cannot be found using Factory Finding techniques.

BOIMHomeOfRegHomes This home is for objects that are not Workload Managed and do not have Queryable or Iterable interfaces. This home should be used for standard business objects if they do not configured for WLM and have not be configured for Query.

BOIMHomeOfRegQIHomes This home is for objects that are not Workload Managed and are configured for Query. This home should be used for those business objects that match this criteria.

BOIMHomeOfRegWLMHomes This home is for objects that are Workload Managed. Business objects that have been configured for WLM support must be stored in these homes.

Object Builder allows configuration of objects into the appropriate home based on their configuration criteria. Only those homes that match the configuration of the business objects are allowed as selections on the appropriate configuration windows.

An Overview of Application Adaptors

An application adaptor provides a place for managed objects, much like a database system provides a home for data or records. An application adaptor is similar to an object-oriented database; it is responsible for providing systems capabilities (for example, identity, caching, and persistence) for its managed objects. By providing such capabilities, an application adaptor provides a certain “quality of service” to its managed objects. Different application adaptors may differ in the trade-off between cost and their quality of service.

An application adaptor provides the following benefits:

- It provides a higher level of abstraction than the object services interfaces for the systems capabilities that it provides. This allows the flexibility that is very important for providing efficient support for objects.

- It enables object services to be packaged in a more integrated and efficient manner than is possible with object services that are provided separately.
- It can delegate many of its object services to existing resource managers, such as database systems, that typically provide systems capabilities in a very integrated way.
- It provides a boundary for administrative functions.

A specific type of application adaptor can only handle one type of legacy store. This means that a DB2 application adaptor provides support for managed objects that are persistent in DB2 tables. However, a different type of application adaptor would be required for managed objects that are stored through CICS.

An Overview of BOIM

Business Object Application Adaptor (BOIM) is the term used to identify IBM's first application adaptor. BOIM fulfils the following needs:

- As an application adaptor it forms a framework for applications
- It is an implementation of the application adaptor framework

The BOIM application adaptor is targeted at a two-level store implementation model for servers and a single-level store model for clients.

An object is fully functional only when in memory; the object returns to a dormant state when removed from memory. The BOIM application adaptor provides memory management schemes for bringing objects into memory and removing them from memory. Such schemes mean that clients do not need to perform special memory management tasks. For example, the BOIM retrieves objects from the database or disk transparently to the clients.

The environment for managed objects supported by the BOIM application adaptor is geared toward having each data object provide its own access to the underlying back-end permanent data store.

Adapting Applications

The simplest applications to convert to the environment targeted by the BOIM application adaptor are those that have the following characteristics:

- The application is not widely distributed. That is, it is built with a traditional client-server model.
- The application has basic back-end access requirements, for example, accessing files or having all the activity for an entire transaction within one database and server process.

Managed Object Assembly

The managed object assembly pattern supported by the BOIM application adaptor has a delegated mixin, a delegated data object, and a managed object that extends the business object implementation.

The BOIM application adaptor provides interfaces (methods) to the managed object without requiring the business object to provide implementation for those methods.

Managed Object

Adapting a business object to the BOIM application adaptor requires development of a managed object and a function that instantiates a managed object. The basic design of the managed object is to give it the appearance of being the managed object assembly. Developing the managed object requires support for the following types of interfaces:

- An interface used by the BOIM application adaptor to communicate with the managed object instead of the managed object assembly
- The business interfaces implemented in the business object
- The interface for which the BOIM application adaptor provides the implementation
- The interface for which no implementation is provided by the BOIM application adaptor but for which the environment requires a default implementation for which no implementation is provided by the BOIM application adaptor

Business Object

The BOIM application adaptor supports business objects that support either of the two interfaces `IManagedServer::IManagedObjectWithDataObject` or `IManagedServer::IManagedObjectWithCachedDataObject` from the Component Broker programming model. The interface supported by the business object is detected by the BOIM application adaptor and no special configuration is required.

Data Object

The BOIM application adaptor provides a framework for data object development by providing a base class with implementation. All data objects are expected to extend this implementation and provide the implementation of the methods that it has defined.

The Life Cycle of Managed Objects

Because the BOIM application adaptor is based on a two-level store, there is more to the life cycle of a managed object than creation and deletion. There are the following additional states for a managed object:

A passivated state
There is no representation of the object in memory, however the state data for the object is saved.

An activated state
There is no representation of the object in memory.

Qualities of Service

The BOIM provides the following qualities of service:

Sharing Data with the Database
The BOIM application adaptor provides control information that helps data objects manage their synchronization with the database.

Objects are synchronized with the underlying data store on the following occasions:

- When the object is created, its corresponding data needs to be inserted into the underlying data store.

- When the object is removed, its corresponding data needs to be deleted from the underlying data store.
- When the data in the object becomes stale, the data needs to be retrieved from the underlying data store.
- When the underlying data store becomes stale, the underlying data store needs to be updated.

Simplifying Interaction with the Data Store
The BOIM application adaptor simplifies interaction with the underlying data store in the following ways:

- A base class for data objects that provides a check to ensure that the key is valid before invoking any of the database access commands
- A performance improvement by updating the database only if data has been changed

Memory Management
The BOIM application adaptor determines when objects are activated in memory and removed from memory.

Tolerance of Programming Errors
The BOIM application adaptor tolerates programming errors within applications.

Locking

One very useful quality of service provided by the BOIM application adaptor is an ability to have multiple copies of a single object. This allows multiple clients access the same object simultaneously by giving each client its own copy of the object. However, the fact that there are multiple copies is hidden from the client and each client thinks it has the only copy of the object. Another useful quality of service provided by the application adaptors is obtaining the locks in the database. All of this sounds great because locks are obtained and business objects can be used simultaneously and yet do not need to be thread safe, so what could be better? All of this does come with a price. There is a possibility of creating applications that result in dead locks as they are accessed by multiple clients.

The following coding practices will help prevent these dead lock situations from occurring.

Identify and label those methods that are read only. Especially when data is cached in the data object. When a method is not identified as read only, then a write lock is obtained as the transaction is committed. This can lead to multiple clients all trying to upgrade locks form read locks to write locks yet not being able to because of the others holding the read locks.

Delegate all data management to the data object (do not cache data in the business object). When the management of the data is delegated to the data object then it can detect when values are changed and only upgrade the lock when necessary.

Ensure that there is recovery code in the client to cover the case where the transaction will not commit. When a dead lock occurs in the database, or a time out is detected in the Component Broker runtime, then all but one of the transactions involved in the deadlock should be rolled back. Leaving one to succeed and the others to fail. Those that failed should try again.

Use the optimistic mode of the cache. This will detect clashes immediately rather than wait for a timer to expire to determine that a deadlock condition exists.

Assembling and Installing Java Business Objects

Once the unit test activity has been completed successfully the next step is to generate and build the extra components necessary to adapt a business object to a server environment and to a particular persistent store, then to package and install all the necessary pieces into a server.

In Component Broker, most of the extra code that is required is C++ code and the Component Broker tools generate almost all of it. As a result, the discussion earlier in this chapter applies with few or no changes when the business object is implemented in Java. In addition, there are some extra steps only in the Java business object case.

This section details the differences in the procedures and code described elsewhere in this chapter when they are applied to a Java business object. As was the case previously, it is not normally necessary for a business object developer to understand the tool-generated code. It is described in case unusual circumstances require debugging.

Create the C++ Client and Server Bindings

Previously Java has been used exclusively, but in order to adapt a Java business object to the Component Broker server environment it is necessary to create some C++ code. If you are using Object Builder to create the business object, it generates a make utility file that does all this for you.

The first group of required C++ files are the bindings generated from the IDL that has already been produced. These are created using the `idlc` command with the `-shh;uc;sc` option. In the case of the Policy example this command would be used on all of Policy.idl, PolicyKey.idl, PolicyCopy.idl, PolicyBO.idl, and PolicyDO.idl. It would also be repeated for the IDL created later in this chapter for the PolicyMO class and the specialized data object.

Create the Managed Object Class and Implementation

This Object Builder-generated class differs only slightly from that described earlier in this chapter. Its IDL is identical, and there are only two differences in the implementation:

- Inheritance
- Construction and destruction

Inheritance

The PolicyMO_Impl class constructed for a C++ business object inherits from the PolicyBO_Impl C++ class, but when the business object is implemented in Java there is no PolicyBO_Impl. Instead, the inheritance is from a generated IOM C++ proxy class that delegates method calls to the Java business object:

```
class PolicyMO_Impl : public virtual ::PolicyMO_Skeleton,
                      public virtual Object_SOMProxyRemotable,
                      public virtual PolicyBO_SOMProxy
```

The Object_SOMProxyRemotable base class is added to resolve some ambiguous overrides of methods introduced in CORBA::Object, and does not contribute any new function.

Inside the business object methods of the PolicyMO_Impl, this same renaming happens for each call to the parent BO business logic:

```
::CORBA::Float PolicyMO_Impl::amount()
{
   ::CORBA::Float retval;
   IBOIMExtLocalToServer_IMixinPointerImpl mixinPointer(mixin());
   CALL_MIXIN_BEFORE(mixinPointer);

   #ifdef CBS_TRACE_DEBUG
         void *trc_handle = BOSS_TRACE_SERVER_START_2(this, "name");
   #endif

   retval = PolicyBO_SOMProxy::amount();

   #ifdef CBS_TRACE_DEBUG
         BOSS_TRACE_SERVER_STOP(trc_handle, "name");
   #endif

   CALL_MIXIN_AFTER(mixinPointer);
   return retval;
}
```

Construction and Destruction

The PolicyMO_Impl for a Java BO has a more complex constructor and destructor than in the C++ case. The constructor calls through the IOM interlanguage run time in order to create the Java business object and to connect it to the C++ Managed Object:

```
static SOMRef* _PolicyMO_Impl_helper()
{
   SOMException e = { 0,0 };
   SOMRef* sr;
   SOMClassRef* scr = SOM_FindClass("PolicyBO", &e);
   if(scr == NULL || (sr=scr->NewObject(&e)) == NULL)
   {
      throw CORBA::NO_IMPLEMENT(0, CORBA::COMPLETED_NO);
   }
   return sr;
}

PolicyMO_Impl::PolicyMO_Impl()
   : Object_SOMProxy(_PolicyMO_Impl_helper()) , mixin_(this)    // See note 1
{
   m_target_type_id = (char*)Policy_RID;
}

PolicyMO_Impl:: PolicyMO_Impl()                                  // See note 2
{
   if(m_somref2 != m_somref && m_somref != NULL)
   {
      SOMRef* tmp = m_somref;
      m_somref = m_somref2;
      setWrapper(this);
      tmp->Destroy();
    }
}
```

Notes about the example:

1. A helper routine creates the corresponding Java business object and returns an interlanguage handle that is used to tie together the C++ MO and Java BO objects.
2. The objects are eventually untied in the MO destructor, allowing them to be separately destroyed.

Data Object Customization

Data objects created for use with a Java business object are identical to those created for a C++ business object, and the same variety of customization choices is available. For more information, see “Data Object Customization” on page 204.

Do Not Mark Data Object Interfaces as "Abstract"

Interfaces of customized data objects, such as the PolicyEmSQLDO interface illustrated earlier in this chapter are marked with the abstract meta information pragma in their IDL. This was done to suppress the declaration of a static _create() method, which is not needed.

In Component Broker, this generates incorrect bindings, and such a data object is not usable with a Java business object. Interfaces that do have corresponding implementations must not be marked abstract. Instead, if the function of the static _create() method is not needed, you can just define a _create that returns NULL.

Generating Server-Specific Java Classes

Two Java classes are used on the server to wrap the Java business object class and adapt it to run in the server environment. Both classes are generated from the business object IDL by running the `idlc` command with the `-sbj` option:

```
idlc -sbj PolicyBO.idl
```

The command generates the _PolicyBOWrapper and _PolicyBOImpl classes. _PolicyBOWrapper extends _PolicyBOBase and is used by IOM to dispatch calls from C++ to the business logic. _PolicyBOImpl extends _PolicyBOWrapper and delegates calls from Java code over to the C++ Managed Object class, which is then able to add manageability and quality of service factors when the business object is called from Java, just as when it is called from C++.

Generating Other C++ Classes

If you have not already done so, a C++ Primary Key class must now be generated and compiled. The server infrastructure requires this, even if the business object is implemented in Java. Chapter 5, “MOFW Server Programming Model” on page 57 describes how to do this.

If the business object makes use of a Copy Helper class, you also need to generate and compile a C++ version of the class.

In both cases, it is essential that the internalize_from_stream() and externalize_to_stream() methods of the C++ and Java versions be consistent with one another. If the Java version reads and writes the policyNo as a CORBA::Long, followed by the amount as a CORBA::float, the C++ version must do the same.

Debugging Java Code Running on the Server

As initially delivered, Component Broker does not contain a Java debugger capable of debugging code running in a server. There are two options available for server debugging.

The System.out.println() standard Java method can be used to produce debugging output in the server console log. A _PolicyBOBase class can be quickly modified to add or delete these statements, recompiled, and replaced, and the server restarted to use it, without regenerating or replacing any of the other Java classes or C++ DLLs.

A second option is to use the jdb debugger supplied as part of the JavaSoft Java Development Kit. This requires that you first use the Component Broker System Manager User Interface to enable the debug option in the server. The procedure to do this follows:

1. In System Manager, from the **View** menu, select **User Level** → **Super User**. This causes the System Manager window to display more detailed information.
2. Open the Host Images folder and expand the host image that corresponds to the name of the server.
3. Expand Server Images and find the server image where the application resides. Click this image with mouse button 2. A pop-up menu appears.
4. Select **Edit**. A notebook opens.
5. Select the **Main** tab.
6. Change the debug enabled attribute to yes.
7. Select the **ORB** tab.
8. Change the request timeout value to 0.
9. Select the **Log Controls** tab.
10. Change the **Console Disposition** to “Console.”
11. Click the **OK** button to exit the server image notebook.

You are now ready to start the Component Broker daemon, name server, and application server. To do so, follow these steps:

1. From the Host Images folder, find the host image that corresponds to the name of the server computer.
2. Click this image with mouse button 2. A pop-up menu appears.
3. Select **Activate**. This starts the communication daemon and the name server. Monitor the Action Console window for completion status.
4. When activation is complete, select the application server and click with mouse button 2. A pop-up menu appears.
5. Select **Run Immediate**. Monitor the Action Console window for completion status.

Note: If the server fails to start, it is possible that the CLASSPATH environment variable may not be configured to include the path to your JDK installation's classes.zip file. To correct the CLASSPATH and restart the server, perform the following steps:

1. Logon to the user ID specified for Systems Management.
2. Change the CLASSPATH environment variable to include the path to the classes.zip file either in the **System Variable** or in the **User Variable** section of the environment setting (**System Variable** is recommended to ensure that other users logging on also pick up the CLASSPATH environment variable changes).

3. Stop the CBConnector service.
4. Restart the CBConnector service.
5. Reactivate the applications using the System Manager User Interface.

You can now run a client program that uses the Java business objects. The Java environment initializes as soon as the first Java class is used in the server, and at that time a single line of output appears in the server console window:

```
  Agent password=password
```

Where *password* is a random sequence of letters and digits. You can now start the jdb debugger:

```
jdb -host hostname -password password
```

Where *hostname* is the Internet-style name of the server host (for example, server1.ibm.com) and *password* is the agent password. The server continues to run throughout this procedure, so you should set breakpoints in the Java business object or to explicitly halt the server using the `jdb` command set. See the JDK documentation on the jdb debugger for more information.

The Managed Object for a Java Specialized Home

The Managed Object class for a Java specialized Home has the responsibility for routing method calls to the correct implementation. Calls to methods from the IManagedAdvancedServer::ISpecializedHome interface are passed to a C++ implementation, while calls to extension methods are forwarded to Java. To accomplish this, the managed object class, which is implemented in C++, inherits from both the C++ base implementation and from several IOM proxy classes:

```
  class PolicyHomeMO_Impl : public virtual ::PolicyHomeMO_Skeleton,
                 public virtual Object_SOMProxyRemotable,
                 public virtual PolicyHomeBO_SOMProxy,
                 public virtual IManagedAdvancedServer_ISpecializedHome_Impl,
                 public virtual IManagedServer::IWrappable_SOMProxy
```

The second last of these is the managed object class for the base C++ Home implementation, and provides implementations of the IManagedClient::IHome interface as well as all the usual framework methods. The two surrounding it, PolicyHomeBO_SOMProxy and IManagedServer::IWrappable_SOMProxy, forward their respective methods over to the Java side. Because many of the framework methods are implemented in more than one of these classes, the managed object class PolicyHomeMO_Impl has to override all methods and explicitly call up to the correct base class or classes.

As a result, PolicyHomeMO_Impl has a total of 50 methods. Only a representative sample is shown here. The constructor and destructor, and all of the framework methods that delegate to the mixin object are no different than for any Java Business Object, so they are not discussed.

Some of the more interesting Managed Object methods correspond to methods implemented in the C++ Home class IManagedAdvancedServer_ISpecializedHome_Impl. These are delegated to that base class:

```
IManagedClient::IManageable_ptr  PolicyHomeMO_Impl::findByPrimaryKeyString(const ::ByteString & key)
{
  IManagedClient::IManageable_ptr temp=NULL;
  IBOIMExtToLocalServer_IMixinPointerImpl mixinPointer(mixin());

  CALL_MIXIN_BEFORE2(mixinPointer,"findByPrimaryKeyString" );

  temp= IManagedAdvanceServer_ISpecializedHome_Impl::findByPrimaryKeyString(key);

  CALL_MIXIN_AFTER(mixinPointer);

  return temp;
}
```

This Managed Object method only calls up to the C++ base class, and ignores any alternate definition of the function that may have been provided in Java. Other Managed Object methods, though, correspond to methods introduced in the specialized PolicyHome interface, and they are delegated through the IOM proxy class over to the Java class _PolicyHomeBOBase:

```
::Policy_ptr PolicyHomeMO_Impl::default_create()
{
  ::Policy_ptr retval;
  IBOIMExtLocalToServer_IMixinPointerImpl mixinPointer(mixin());
  CALL_MIXIN_BEFORE2(mixinPointer,"findSomething");

  #ifdef CBS_TRACE_DEBUG
    void *trc_handle = BOSS_TRACE_SERVER_START_2(this, "default_create",
"Java");
  #endif

  // call the real business logic
  retval = PolicyHomeBO_SOMProxy::default_create();

  #ifdef CBS_TRACE_DEBUG
   BOSS_TRACE_SERVER_STOP(trc_handle, "default_create");
  #endif

  CALL_MIXIN_AFTER(mixinPointer);

}
```

Finally, the following framework methods are special cased so that both the C++ and Java implementations are called:

- initForCreation()
- uninitForDestruction()
- initForReactivation()
- uninitForPassivation()
- syncToDataObject()
- syncFromDataObject()

This ensures that both the C++ and Java portions of the composite implementation will be able to correctly initialize and shut down, and allows them both access to the Home's Data Object:

```
::CORBA::Void PolicyHomeMO_Impl::initForReactivation( ::IManagedServer::IDataObject_ptr theDO)
{
  #ifdef CBS_TRACE_DEBUG
   BOSS_TRACE_CREATE(this,"PolicyHome");
  #endif

  IManagedAdvanceServer_ISpecializedHome_Impl::initForReactivation(theDO);

  PolicyHomeBO_SOMProxy::initForReactivation(theDO);

}
```

Appendix A. Artifacts Produced in Building Objects

The following table summarizes the artifacts of the development process for object components (interface, business, and managed objects).

Table 9 (Page 1 of 2). Artifacts of the Development Process for Object Components

Artifact Type	**Artifact Name**		
	Abstraction Interface	**AbstractionBusiness Object**	**AbstractionManaged Object**
IDL	Abstraction.idl	AbstractionBO.idl	AbstractionMO.idl (Object Builder builds these automatically for BOIM managed objects that customers write.)
.hh (language usage binding) Never modify these – have the makefile generate them.	Abstraction.hh (contains the Abstraction_Skeleton class and the Abstraction C++ class)	AbstractionBO.hh (contains the AbstractionBO_Skeleton class and the Abstractionbusiness object C++ class)	AbstractionMO.hh (contains the AbstractionMO_Skeleton class and the Abstractionmanaged object C++ class)
_C.cpp (client side binding) Never modify these – have the makefile generate them.	Abstraction_C.cpp	AbstractionBO_C.cpp	AbstractionMO_C.cpp
_S.cpp (server side bindings) These #include the _C.cpp. Never modify these – have the makefile generate them.	Abstraction_S.cpp	Abstraction_S.cpp	Abstraction_S.cpp
.ih (implementation interface - emit once or enter manually)	Not needed	AbstractionBO.ih (defines AbstractionBO_Impl that inherits only from AbstractionBO_Skeleton) (and inherits from IManageable_Impl)	AbstractionMO.ih (defines AbstractionMO_Impl that inherits from AbstractionBO_Impl and AbstractionMO_Skeleton) (Object Builder builds these automatically for BOIM managed objects that customers write.)
_I.cpp (implementation code - emit once or enter manually)	Not needed	AbstractionBO_I.cpp (implementation of real logic – the code)	AbstractionMO_I.cpp (delegator, traffic cop) (Object Builder builds these automatically for BOIM managed objects that customers write.)
_C.obj (from _C.cpp – need rule in makefile)	Abstraction_C.o (primary binding used by remote clients. See note.)	Not needed	Not needed

Table 9 (Page 2 of 2). Artifacts of the Development Process for Object Components

Artifact Type	**Artifact Name**		
	Abstraction Interface	**AbstractionBusiness Object**	**AbstractionManaged Object**
_S.obj (placed on servers; from _S.cpp – need rule in makefile)	Abstraction_S.o	AbstractionBO_S.o	AbstractionMO_S.o
_I.obj (placed on servers; from _I.cpp – need rule in makefile)	Not possible	AbstractionBO_I.o	AbstractionMO_I.o
CLIENT.LIB (group one or more abstractions into an LIB)	contributes its _C.o only (see note)	no contribution	no contribution
ASERVER.LIB (group one or more abstractions into an LIB)	contributes _S.o	contributes _S.o and _I.o	contributes _S.o and _I.o

Note: If the same DLL is to be used on both client and server (which makes sense in the VisualAge for C++ case), then using the Abstraction_S.o for the client DLL allows this same DLL to be used on both client and server.

Appendix B. Interface Definition Language

The interface to a class of objects contains the information that a caller must know to use an object, specifically, the names of its attributes and the signatures of its operations. The interface is described in a formal language independent of the programming language used to implement the object's operations. The formal language used to define object interfaces is the Interface Definition Language (IDL), standardized by CORBA.

The implementation of a class of objects (that is, the procedures that implement operations and the variables used to store an object's state) is written in the implementor's preferred programming language (for example, C++ or Java).

A completely implemented class definition consists of the following parts:

- An IDL specification of the interface to instances of the class: the interface definition file (or IDL file).
- Method procedures written in the implementor's language of choice: the implementation file(s).

The IDL compiler takes as input an object interface definition file (the IDL file) and produces binding files that make it convenient to implement and use objects that support the defined interface within a particular programming language.

Note: Component Broker is based on CORBA Version 2.0. All IDL used with Component Broker must be CORBA 2.0-compliant without IDL extensions.

IDL Name Scoping

The IDL file forms a naming scope (or scope). Modules, interface statements, structures, unions, operations, and exceptions form nested scopes. An identifier can only be defined once in a particular scope. Identifiers can be redefined in nested scopes.

Names can be used in an unqualified form within a scope, and the name will be resolved by successively searching the enclosing scopes. Once an unqualified name is defined in an enclosing scope, that name cannot be redefined.

Fully qualified names are of the form:

```
scope-name::identifier
```

For example, operation name meth defined within interface Test of module M1 would have the fully qualified name:

```
M1::Test::meth
```

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

Qualified names can also take the form:

```
::identifier
```

These names are resolved by locating the definition of *identifier* within the outermost name scope.

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

- Before the IDL Compiler scans the IDL file, the name of the current root and the name of the current scope are empty. As each module is encountered, the string "::" and the module name are appended to the name of the current root. At the end of the module, they are removed.
- As each interface, struct, union, or exception definition is encountered, the string "::" and the associated name are appended to the name of the current scope. At the end of the definition, they are removed. While parameters of an operation declaration are processed, a new unnamed scope is entered so that parameter names can duplicate other identifiers.
- The global name of an IDL definition is then the concatenation of the current root, the current scope, a "::", and the local name for the definition.

The names of types, constants, and exceptions defined by base interfaces are accessible in a derived interface. References to these names must be unambiguous. Ambiguities can be resolved by using a scoped name (prefacing the name with the name of the interface that defines it, and the characters "::", as in *base-interface::identifier*). Scope names can also be used to refer to a constant, type, or an exception name defined by a base interface but redefined by a derived interface.

Type and Constant Declarations

IDL specifications may include type declarations and constant declarations as in C and C++, with the restrictions and extensions described below. IDL supports the following declarations.

Integral Types

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

- short -2**15 .. (2**15)-1
- long -2**31 .. (2**31)-1
- unsigned short 0 .. (2**16)-1
- unsigned long 0 .. (2**32)-1

Floating Point Types

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

Since returning floats and doubles by value may not be compatible across Microsoft Windows® compilers, client programs should return floats and doubles by reference.

Character Type

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

Boolean Type

IDL supports a boolean type for data items that can take only the values zero (FALSE) and one (TRUE).

Octet Type

IDL supports an octet type, an 8-bit quantity guaranteed not to undergo conversion when transmitted between a client and server process. The octet type can be used in place of the unsigned char type.

Any Type

IDL supports an any type, which permits the specification of values of any IDL type. Conceptually, an any consists of a value and a TypeCode that represents the type of the value. The TypeCode class provides functions for obtaining information about an IDL type.

Constructed Types (struct, union, enum)

In addition to the above basic types, IDL also supports three constructed types: struct, union, and enum. The structure and enumeration types are specified in IDL just as they are in C and C++, with the following restrictions:

- Unlike C/C++, recursive type specifications are allowed only through the use of the sequence template type (see below).
- Unlike C/C++, structures, discriminated unions, and enumerations in IDL must be tagged. For example, struct { int a; ... } is an invalid type specification (because the tag is missing). The tag introduces a new type name.
- In IDL, constructed type definitions need not be part of a typedef statement; furthermore, if they are part of a typedef statement, the tag of the struct must differ from the type name being defined by the typedef. For example, the following are valid IDL struct and enum definitions:

  ```
  struct myStruct {
                 long x;
                 double y;
            };                              /* defines type name myStruct */
            enum colors { red, white, blue };  /* defines type name colors */
  ```

  The following IDL definitions are **not** valid:

  ```
  typedef struct myStruct {                      /*  NOT VALID  */
                 long x;                       /*  Tag myStruct is the same */
                 double y;                     /*  as the type name below;  */
            } myStruct;                        /*  myStruct has been redefined */
            typedef enum colors { red, white, blue } colors;  /* NOT VALID */
  ```

Union Type

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

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

The *identifier* following the union keyword defines a new legal type. (Union types may also be named using a typedef declaration.)

The *switch-type* specifies an integral, character, boolean, or enumeration type, or the name of a previously defined integral, boolean, character or enumeration type.

Each *case* of the union is specified with the following syntax:

```
case-label+ type-spec declarator;
```

Where

- Each *caselabel* has one of the following forms:

  `case` *`const-expr`*`:`

  default: The *const-expr* is a constant expression that must match or be automatically castable to the *switch-type*. A default case can appear no more than once.
- *type-spec* is any valid type specification.
- *declarator* is an identifier or an array declarator (such as, foo[3][5]).

Template Types (sequences and strings)

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

- `sequence < simple-type [, positive-integer-const] >`

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

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

- `string [ <` *`positive-integer-const`* `> ]`

  where the optional *positive-integer-const* is a constant expression that specifies the maximum size of the string (as a positive integer, which does not include the extra byte to hold a NULL as required in C/C++).
- CORBA does not prescribe specific rules on how to process blanks contained within strings. Thus, Component Broker needs help in determining whether keys “ABC” and “ABC” (Insert three trailing blanks after the second “ABC.”) refer to the same or different managed objects. To aid with this decision, Component Broker offers multiple semantic choices for processing string attributes when they are defined on a business object. When multiple string attributes exist within a single business object, mixing and matching of the various semantics will be allowed. The three semantic choices are:
  – CORBA - this is consistent with the support provided through release 1.3 of Component Broker. No rules are defined for processing blanks within these strings.
  – Trailing blanks stripped - Object Builder will generate code that removes trailing blanks.
  – Pad with blanks to fixed length - Object Builder will generate code that adds trailing blanks to some bounded string length.

Customers are discouraged from using the CORBA semantics for strings to be used as key attributes. Customers must follow similar semantic rules as enforced by our generated code for any string extensions they implement.

Arrays

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

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

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

Object Types

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

Constants

Constants are declared in IDL just as in C++, except that the type of the constant must be a valid IDL type. IDL Constant declarations take the following form:

const <*const-type*> *identifier*=<*constant-expression*];

The *const-type* must be a valid IDL integer, char, boolean, floating point, string, or user-defined type name. The *identifier* is the name of the constant being defined. The *constant-expression* is a constant expression as in C/C++, and can include the usual C/C++, unary and binary operators (|, ^, &, >>, <<, +, -, *, /, %, ˜), parentheses for controlling operator precedence, literal values (integer, string, character, and floating point) as in C/C++, and the boolean literal values TRUE and FALSE.

Interface Declarations

The IDL specification for a class of objects must contain a declaration of the interface these objects will support. When objects are implemented using classes, the interface name is used as a class name as well. In addition to the interface name and its base interface names, an interface indicates new methods (operations), and any constants, type definitions, and exception structures that the interface exports. An interface declaration has the following syntax:

```
interface  interface-name [: base-interface1, base-interface2, ...]
{
   constant declarations  (optional)
   type declarations       (optional)
   exception declarations  (optional)
   attribute declarations  (optional)
   operation declarations  (optional)
};
```

The base-interface names specify the interfaces from which *interface-name* is derived. Parent-interface names are required only for the immediate base interface(s). Each base interface must have its own IDL specification (which must be #included in the IDL file). A base interface cannot be named more than once in the interface statement header.

In general, an interface header must precede any subsequent references to . For a discussion of multiple interface statements, see Multiple IDL Interfaces and Modules.

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

- “Attribute Declarations” on page 278
- “Comments” on page 281
- “Type and Constant Declarations” on page 272
- “Operation Declarations”

Constant, Type, and Exception Declarations Within an Interface

The form of a constant, type, or exception declaration within the body of an interface declaration is the same as described in “Interface Declarations” on page 275. Constants and types defined within an interface are transferred by the IDL compiler to the binding files it generates for that interface.

Types, constants, and exceptions defined in a base interface are accessible to the derived interface. References to them, however, must be unambiguous. Ambiguities can be resolved by using a fully scoped name, (prefacing a name with the name of the interface that defines it) separated by the characters "::" as illustrated below:

```
  MyBaseInterface::myType
```

A leading "::" can be used to fully-qualify a reference starting from the outermost name scope.

The derived interface can redefine any of the type, constant, and exception names that were inherited. The derived interface cannot, however, redefine attributes or operations. To refer to a constant, type, or exception "name" defined by a base interface and redefined by "interface-name," use the "parent-name::name" syntax.

Operation Declarations

Operation declarations define the interface of each operation introduced by the interface. (An IDL operation is typically implemented by a method in the implementation programming language. Hence, the terms operation and method are often used interchangeably.) An operation declaration is similar to a C++ virtual function definition:

```
  [ oneway ] type-spec
identifier ( parameter-list ) [ raises-expr ] [ context-expr ] ;
```

where

identifier is the name of the operation.

type-spec is any valid IDL type, except a sequence, or the keyword void, indicating that the operation returns no value. (Although the return type cannot be a sequence, it can be a user-defined type that is a sequence.) Unlike C and C++ procedures, operations that do not return a result must specify void as their return type.

The remaining syntax of an operation declaration is elaborated in the following subtopics.

"oneway" Keyword

The optional oneway keyword specifies that when a caller invokes the operation, no reply is expected or received. The invocation semantics of a oneway operation are best-effort, which does not guarantee delivery of the call. Best-effort implies that the operation will be invoked at most once. A oneway operation must not have any output parameters and must have a return type of void. A oneway operation also must not include a raises expression (see below).

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

Parameter List

The parameter-list contains zero or more parameter declarations for the operation, delimited by commas. (The target object for the operation is not explicitly specified as an operation parameter in IDL.) If there are no explicit parameters, the syntax "( )" must be used, rather than "(void)". A parameter declaration has the following syntax:

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

where *type-spec* is any valid IDL type (except a sequence), and *declarator* is an identifier or an array declarator. Although the type of a parameter cannot be a sequence, it can be a user-defined type that is a sequence.

The required in|out|inout directional attribute indicates whether the parameter is to be passed from caller to callee (in), from callee to caller (out), or in both directions (inout). The following are examples of valid operation declarations:

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

An operation's implementation should not modify an in parameter. If a change must be made by the implementation, the implementation should copy the parameter and only modify the copy.

If an operation raises an exception, the values of the return result and the values of the out and inout parameters (if any) are undefined.

"raises" Expression

The optional raises expression in an IDL operation declaration indicates which exceptions the operation may raise. A raises expression is specified as follows:

```
raises ( identifier1, identifier2, ... )
```

where each *identifier* is the name of a previously defined exception. In addition to the exceptions listed in the raises expression, an operation may also signal any of the standard exceptions. Standard exceptions, however, should not appear in a raises expression. If no raises expression is given, then an operation can raise only the standard exceptions. “Exception Declarations” on page 278 contains further information on defining exceptions and the list of standard exceptions.

"context" Expression

The optional context expression (context-expr) in an operation declaration indicates which elements of the caller's context the operation's implementation may consult. A context expression is specified as follows:

```
context ( identifier1, identifier2, ... )
```

where each *identifier* is a string literal made up of alphanumeric characters, periods, underscores and asterisks. The first character must be alphabetic, and an asterisk can only appear as the last character, where it serves as a wildcard matching any characters. If convenient, identifiers may consist of period-separated valid identifier names, but that form is optional.

The Context is a special object that is specified by the CORBA standard. It contains a property list: a set of property-name/string-value pairs that the caller can use to store information about its environment that operations may find useful. It is used in much the same way as environment variables. It is passed as an additional parameter to operations that are defined as context-sensitive in IDL.

The context expression of an operation declaration in IDL specifies which property names the operation uses. If these properties are present in the Context object supplied by the caller, they will be passed to the object implementation, which can access them via the interface of the Context object.

The argument that is passed to the operation having a context expression is a Context object, not the names of the properties. The caller must create a Context object and use the interface of the Context object to set the context properties. The caller then passes the Context object in the operation invocation. The CORBA standard allows properties in addition to those in the context expression to be passed in the Context object.

Attribute Declarations

Declaring an attribute as part of an interface is equivalent to declaring one or two accessor operations: one to retrieve the value of the attribute (a get or read operation) and (unless the attribute specifies readonly) one to set the value of the attribute (a set or write operation).

Attributes are declared as follows:

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

where:

type-spec specifies any valid IDL type (except a sequence).

declarators is a list of identifiers, delimited by commas. An array declarator cannot be used directly when declaring an attribute, but the type of an attribute can be a user-defined type that is an array. Although the type of an attribute cannot be a sequence, it can be a user-defined type that is a sequence. The optional readonly keyword specifies that the value of the attribute can be accessed but not modified. (In other words, a readonly attribute has no set operation.) Below are examples of attribute declarations, which are specified within the body of an interface statement:

```
interface Goodbye: Hello
{
   void  sayBye();
   attribute short xpos;
   attribute char c1, c2;
   readonly attribute float xyz;
};
```

Attributes are inherited from base interfaces. An inherited attribute name cannot be redefined to be a different type.

Exception Declarations

IDL specifications can include exception declarations, which define data structures to be returned when an exception occurs during the execution of an operation. A name is associated with each type of exception. Optionally, a struct-like data structure for holding error information can also be associated with an exception. Exceptions are declared as follows:

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

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

```
type-spec declarators ;
```

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

If an exception is returned as the outcome of an operation, the exception *identifier* indicates which exception occurred. The values of the members of the exception provide additional information specific to the exception. “Operation Declarations” on page 276 describes how to indicate that a particular operation may raise a particular exception.

The following is an example showing the declaration of a BAD_FLAG exception:

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

In addition to user-defined exceptions, there are several predefined exceptions for system run-time errors. The standard exceptions as prescribed by CORBA are subclasses of CORBA::SystemException. These exceptions correspond to standard run-time errors that may occur during the execution of any operation (regardless of the list of exceptions listed in the operation's IDL specification).

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

```
enum completion_status
{
   COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
};
exception NO_MEMORY
{
   unsigned long minor;
     completion_status completed;
};
```

The "completion_status" value indicates whether the operation was never initiated (COMPLETED_NO), if the operation completed its execution prior to the exception (COMPLETED_YES), or if the operation's completion status is indeterminate (COMPLETED_MAYBE).

IDL Syntax

This section describes the syntax of the Interface Definition Language (IDL), as specified by the CORBA standard. This section describes the syntax and semantics of IDL using the following conventions:

bold	Indicates Literals (such as keywords).
italics	Indicate user-supplied elements.
{ }	Groups related items together as a single item.
[ ]	Encloses an optional item.
*	Indicates zero or more repetitions of the preceding item.

+	Indicates one or more repetitions of the preceding item.
|	Separates alternatives.
_	Within a set of alternatives, an underscore indicates the default, if defined.

IDL is a formal language used to describe object interfaces. An IDL definition specifies, for a class of objects, what methods (operations) are available, their return types, and their parameter types. For this reason, we often speak of an IDL specification for a class (as opposed to simply an object interface).

IDL generally follows the same lexical rules as C and C++. Exceptions to C++ lexical rules include:

- IDL uses the ISO Latin-1 (8859.1) character set.
- White space is ignored except as token delimiters.
- C and C++ comment styles are supported.
- IDL supports standard C/C++ preprocessing, including macro substitution, conditional compilation, and source file inclusion.
- Identifiers (user-defined names for operations, attributes, instance variables, and so on) are composed of alphanumeric and underscore characters (with the first character alphabetic) and can be of arbitrary length, up to an operating-system limit of about 250 characters.
- Identifiers must be spelled consistently with respect to case throughout a specification.
- Identifiers that differ only in case yield a compilation error.
- Within a particular name scope, there is a single name space for all identifiers, regardless of their type. For example, using the same identifier for a constant and an interface name within the same name scope yields a compilation error.
- Integer, floating point, character, and string literals are defined as in C and C++.

The terms listed in Table 10 are reserved keywords and may not be used otherwise. Keywords must be spelled using upper- and lower-case characters exactly as shown in the table. For example, "void" is correct, but "Void" yields a compilation error.

Table 10. Reserved Keywords for IDL

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

A typical IDL specification for a single interface, residing in a single IDL file, has a form which includes the following specifications listed.

- “Interface Declarations” on page 275 (optional)
- “Exception Declarations” on page 278 (optional)
- “Include Directives” on page 281 (optional)
- “Type and Constant Declarations” on page 272 (optional)

The order is unimportant, except that interface names must be declared (or forward referenced) before they are referenced.

For more information on the CORBA standard for IDL, see *The Common Object Request Broker: Architecture and Specification* .

Comments

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

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

IDL also supports throw-away comments. They may appear anywhere in an IDL specification. Throw-away comments start with the string "//#" and end at the end of the line. Use throw-away comments to comment out portions of an IDL specification.

Include Directives

The IDL specification for an interface normally contains #include statements that tell the IDL compiler where to find the interface definitions (the IDL files) for each of the interface's parent (direct base) interfaces and for other referenced types.

The file orb.idl can be included to access IDL types defined by the CORBA specification that are not IDL keywords.

As in C and C++, if a filename is enclosed in angle brackets ([ ]), the search for the file begins in system-specific locations. If the filename is in double quotation marks (""), the search for the file begins in the current working directory, before searching the system-specific locations.

In addition to the #include directive, other preprocessor directives can be used in IDL.

Pragma Directives

Component Broker supports the following pragmas.

- localonly
- localonly abstract
- cpponly
- init
- ID
- Prefix
- version

localonly Pragma

This Component Broker unique pragma supports the generation of bindings for objects that are known to be local (not distributed). This pragma may occur at any point in the IDL file following the definition or forward declaration of the designated interace.

The syntax is:

```
#pargma meta interface-name localonly
```

The IDL interface identified by *interface-name* is treated by generated bindings as stricktly local to the caller's process. No calls to the CORBA ORB occur when invoking the operations defined in this interface. *interface-name* may be a simple name of an interface in the current scope or a fully- or partially-qualified interface name. The interface must be previously defined or forward declared when the pragma statement is encountered.

localonly abstract Pragma

This Component Broker unique pragma is like the localonly pragma, but it does not produce a _create() function.

The syntax is:

```
#pragma meta interface-name localonly abstract
```

Like the localonly pragma, but the client bindings contain no _create() function.

cpponly Pragma

This Component Broker unique pragma suppresses the generation of IOM interlanguage bindings.

The syntax is:

```
#pragma meta interface_name cpponly
```

In the default case, without this pragma, two sets of bindings are produced:

- The standard CORBA C++ bindings suitable for use with the ORB component.
- IOM bindings suitable for interlanguage interaction.

Without this pragma, only the standard CORBA C++ bindings are produced.

init Pragma

This Component Broker unique pragma specifies a function to use to initialize newly created objects.

The syntax is:

```
#pragma mets method-name init
```

This pragma allows the IDL to specify the name of a function to be used to initialize the newly created method. When this pragma is not used, the emitters produce a _create() function that takes no parameters and does no initialization after the new object is created.

For example, if the IDL contains:

```
interface A
{
   void initFunction(int);
};
#pragma meta A::initFunction init
```

the C++ class A that implements interface A will have a _create() function that takes an int parameter (because initFunction takes an int). Also, the code inside _create(int) creates a new instance of class A and then call initFunction(int) on the newly created object, passing along its int parameter.

ID Pragma

This CORBA-defined pragma overrides the default RepositoryID for an IDL entity.

The syntax is:

```
#pragma ID scoped-name literal-string
```

which sets the RepositoryID of *scoped-name* to *literal-string* instead of the default Repository ID.

Prefix Pragma

This CORBA-defined pragma sets the RepositoryID prefix

The syntax is:

```
#pragma prefix string
```

which sets the current prefix used in generating OMG IDL format RepositoryIDs. The specified prefix applies to RepositoryIDs generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered.

version Pragma

This CORBA-defined pragma sets the RepositoryID version number.

The syntax is:

```
#pragma version scoped-name major.minor
```

which uses the *major.minor* as the version number for RepositoryID of the *scoped-name*.

Multiple IDL Interfaces and Modules

A single IDL file can define multiple interfaces. When a file defines two or more interfaces that reference one another, forward declarations can be used to declare the name of an interface before it is defined. This is done as follows:

```
interface interfaceName ;
```

The actual definition of the interface for *interfaceName* must appear later in the same IDL file.

If multiple interfaces are defined in the same IDL file, they can be grouped into modules, by using the following syntax:

```
module moduleName { definition+ };
```

where each definition is a type declaration, constant declaration, exception declaration, interface statement or nested module statement. Modules are used to scope identifiers.

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

The idlc Command

Creates usage and implementation bindings for interfaces described in IDL files.

The Interface Definition Language Compiler (idlc) command compiles one or more files containing CORBA 2.0-compliant IDL statements, and optionally produces generated language bindings appropriate to each named input file.

The syntax for the idlc command is:

```
idlc [options] <filename>...
```

Where:

[options]

See Options for the idlc Command for a complete list of the options, their usage, and their restrictions.

<**filename**>

<filename> may be specified without a file name extension; if no file name extension is supplied, it is assumed to be “.idl.” The wildcard character “*” is permitted to appear once in the non-path portion of the file name. For example, the following are acceptable ways to refer to the file “xyz.idl” in directory “E:\idl\src”:

```
E:\idl\src\xyz.idl
E:\idl\src\xyz
E:\idl\src\*.idl (this may refer to additional files as well)
E:\idl\src\x*.idl (all files starting with x)
xyz.idl (if E:\idl\src is the current directory)
xyz
x*
```

When all specified input files are compiled, the idlc command returns a value of zero if no errors were detected; otherwise, a non-zero value is returned.

If no emitters are specified for the idlc command, then only the syntax of the named files is checked, and any errors reported. (See the discussion of the -e option (IDLC Command options) for how to specify emitters.) When a compilation error (but not a warning) is detected for a particular input file, the emit phase for that file is skipped.

For additional information on emitted files, see Emitted File Names.

Options for the idlc Command

Options for the idlc command are preceeded with a dash (-) character and may be specified individually or run together. For example, `-p -v -V` or `-pvV` is acceptable.

Some options accept an argument. These options must either be specified individually or as the last option in a run-together grouping (for example, `-p -m tie` or `-pm tie`). The space between this type of option and its argument is optional. For example, either `-mtie` or `-m tie` is equally acceptable.

All options are case-sensitive, even on platforms where file names are not case-sensitive.

Table 11 describes each available option:

Table 11 (Page 1 of 4). idlc Command Options

Option	Description
-d <directory-name>	Specifies the directory in which to place emitted output files and directories. If none is specified, the default is the current directory.
-V	Shows the version number of the idlc command.
-v	Specifies verbose mode. This shows all internal commands (and their arguments) issued by the idlc command.
?	Writes a brief description of the idlc command syntax to standard output.
-h	Synonymous with -?.
-D <define-expression>	Predefines a preprocessor variable for the IDL compiler.
-I <include-directory>	Adds a directory to the list of directories used by the IDL compiler to find #include files. In addition to the -I option, the IDLC_INCLUDE environment variable can be used to specify a list, with <include-directory> names separated by the PATH separator character.
-i <file-name>	Specifies the name of a file to be compiled that does not have the .idl extension. The <file-name> should not have an implicit .idl suffix added to its name.
-p	Used as a shorthand for `-D__PRIVATE__`.

Table 11 (Page 2 of 4). idlc Command Options

Option	Description
-e <emit-list>	Specifies a list of emitters to run. Emitters are specified with a short 2- or 3-character designator. Each emitter in the list should be separated from the others with a colon (:) or semicolon (;) character. Valid emitter names are: **hh** Produces C++ usage bindings. If no modifiers are present, bindings with support for remotable cross-language operation are produced. The cpponly, localonly, and somthis modifiers cause specialized bindings to be produced (see -m<name[=value]>). **sc** Produces a C++ skeleton for the Basic Object Adapter of the ORB. If no modifiers are present, bindings with support for remotable cross-language operation are produced. The cpponly, localonly, and somthis modifiers cause specialized bindings to be produced (see -m<name[=value]>). **uc** Produces local implementations needed by the C++ usage bindings. If no modifiers are present, bindings with support for remotable cross-language operation are produced. The cpponly, localonly, and somthis modifiers cause specialized bindings to be produced (see -m<name[=value]>). **ih** Produces a C++ implementation header. If the mo modifier (see -m<name[=value]>) is specified, bindings that support Component Broker managed objects are produced; otherwise, pure CORBA C++ bindings, suitable for use with a standalone ORB, are produced. **ic** Produces a Component Broker C++ managed object implementation template. The mo modifier has the same effect as the ih emitter. **uj** Produces the cross-language Java usage bindings. **sj** Produces a Java implementation skeleton. **bj** Creates files needed to support business objects written in Java. The files are • _<interface_name>Wrapper.java that replaces _<interface_name>Skeleton.java • _<interface_name>Impl.java that is the implementation-side proxy for the C++ managed object associated with the Java business object. **ir** Updates the CORBA Interface Repository with the interfaces in this compilation unit. The idlc command looks for emitters specified by the -e or -s flags and looks for an <emit-list> in an environment variable named IDLC_EMIT. If no <emit-list> can be found from any source, no emitters are run, but the IDL compiler is invoked to check for syntax errors in the input files.
-s <emit-list>	Synonymous with -e.

Table 11 (Page 3 of 4). idlc Command Options

Option	Description
-m <name[=value]>	Specifies an output modifier. A modifier may be given as a name or a name=value expression. The emitters are sensitive to the following modifiers: **LINKAGE=<value>** Used to insert customized C++ linkage modifiers into the generated bindings. **notcconsts** Eliminates the generation of C++ TypeCode constants and overloaded any operators. **tie** Generates "tie-style" bindings that assume delegation rather than inheritance. **cpponly** Suppresses the production of cross-language bindings and produces standard CORBA C++ bindings suiitable for use with a standalone ORB. cpponly affects the bindings produced by the hh, sc, and uc emitters. **localonly** Generates bindings that can only be used to access a local object for all of the most-derived interfaces in the IDL file. For an alternative mechanism that also offers finer control over the affected interfaces, see "localonly" Pragmas. **nointerface** Suppresses the generation of the <interface_name>.java file that would otherwise be created by the uj emitter. **nohelper** Suppresses the generation of the <interface_name>Helper.java file that would otherwise be created by the uj emitter. **noholder** Suppresses the generation of the <interface_name>Holder.java file that would otherwise be created by the uj emitter. **nostub** Suppresses the generation of the _<interface_name>Stub.java file that would otherwise be created by the uj emitter. **noimplbase** Suppresses the generation of the _<interface_name>ImplBase.java file that would otherwise be created by the sj emitter. **noskeleton** Suppresses the generation of the _<interface_name>Skeleton.java file that would otherwise be created by the sj emitter. **noimpl** Suppresses the generation of the _<interface_name>Impl.java file that would otherwise be created by the bj emitter. **nowrapper** Suppresses the generation of the _<interface_name>Wrapper.java file that would otherwise be created by the bj emitter. **orbadapter** Generates C++ bindings that allow the C++ ORB to dispatch Java implementations. **IRforce** Forces the IR emitter to destroy objects already present in the IR with the same name as in the IDL being produced. **dllname=<value>** Puts NT import/export specifications into classes contained in the DLL named by <value>. **preInclude=<file-name>** Adds the line: `#include <file-name>` to the .hh file, just before the line that includes corba.h. **postInclude=<file-name>** Adds the line: `#include <file-name>` just before the end of the .hh file.

Table 11 (Page 4 of 4). idlc Command Options

Option	Description
-J	Passes options through to the Java interpreter used internally. For example: `-J"-mx32m"` sets the heap size for the interpreter to 32M.

Emitted File Names

The idlc command process IDL files and produces output files that contain language-specific usage and implementation bindings for the IDL interface. Each emitter (see -e*emit-list* in Options for the idlc Command) produces one or more output files. The rules used to generate the names of these output files are described in the following sections.

C++ Emitters

The names of the generated output files are derived frrom the file name of the corresonding IDL file. For a file named *filestem*.idl, the following list of output files may be emitted when the idlc command is run. The list contains the emitter and its corresponding output file name.

hh	*filestem*.hh
sc	*filestem*_S.cpp
uc	*filestem*_C.cpp
ih	*filestem*.ih
ic	*filestem*_I.cpp

Java Emitters

The Java emitters produce multiple output files as defined by the CORBA Java bindings. These file names are unrelated to the name of the corresponding IDL file, and depend instead on the name of the IDL elements being mapped. In keeping with the required correspondence between Java package names and the directory structure where Java source files reside, subdirectories of the current directory (or the directory specifiec by the -d command line option) are created dynamically to hold the generate .java files.

IDLC_OPTIONS Environment Variables

Any idlc command line option can be specified in the environment by adding the option to the string named IDLC_OPTIONS environment variable. Options specified in the IDLC_OPTIONS variable are treaeted as if they were keyed on the command line before any of the actual command line options. For example, if:

```
IDLC_OPTIONS="-m cpponly -mdllname=mydll"
```

and the command line is:

```
idlc -ehh idlfile
```

the result is the same as if the IDLC_OPTIONS variable was not set and the command line was:

```
idlc -m cpponly -mdllname=mydll -ehh idlfile
```

The IDL-to-Java Compiler

The IDL-to-Java Compiler generates Java bindings for a given IDL file.

Quick Reference

The command to invoke the IDL-to-Java code compiler has the general form:

```
java com.ibm.idl.toJava.Compile [options] source_IDL
```

where *source_IDL* is the name of a file that contains IDL definitions, and [options] is any combination of the options listed in “Compilation Options.” Options may appear in any order, but must precede the IDL file specification.

Compilation Options

Invoke the compiler without any arguments to view the following options:

-bean Emit client-side bindings as Java Beans.

-d This is equivalent to the following line in an IDL file: #define *symbol.*

-emitAll Emit all types, including those found in #include files.

-f*side* Defines what bindings to emit. *side* is one of `client`, `server`, `all`, `serverTie`, and `allTie`. Assumes -fclient if the flag is not specified.

-i *include_ path* By default, the current directory is scanned for included files. This option adds another directory.

-keep If a file to be generated already exists, do not overwrite it. By default it is overwritten.

-m Generate information to be included in a make description file; output goes to .u file.

-pkgPrefix *type package* Wherever *type* is encountered, ensure it resides within *package* in all generated files. *type* is a fully qualified, Java-style name.

-sep *string* Only valid with -m. Replace the file separator character with *string* in the file names listed in the .u file.

-stateful Emit Object by Value bindings.

-td *target_directory* Emit bindings to *target_directory* rather than to the current directory.

-v Verbose mode.

The sections that follow provide complete instructions for using each option along with tips about when to use them.

Emitting Client and Server Bindings

To generate the Java bindings for an IDL file named My.idl, set the current working directory to that containing My.idl and issue the following command:

```
java com.ibm.idl.toJava.Compile My.idl
```

This command generates client-side bindings only and is equivalent to:

```
java com.ibm.idl.toJava.Compile -fclient My.idl
```

Client-side bindings include all generated files except the Skeleton. If you wish to generate server-side bindings for My.idl, issue the command:

```
java com.ibm.idl.toJava.Compile -fserver My.idl
```

This command generates all client-side bindings plus an inheritance-model Skeleton (ImplBase). Currently, server-side bindings include all generated files, even the Stub. Thus, the command above is currently equivalent to each shown below:

```
java com.ibm.idl.toJava.Compile -fclient -fserver My.idl
java com.ibm.idl.toJava.Compile -fall My.idl
```

The compiler generates inheritance-model Skeletons by default. Given an interface My defined in My.idl, the compiler generates Skeleton _MyImplBase.java. You provide the implementation for My, which must extend _MyImplBase.

There is another server-side model called tie. Tie Skeletons delegate method requests to the actual object via reference rather than via extension, as with inheritance-model Skeletons. The following commands generate tie-model bindings for My.idl:

```
java com.ibm.idl.toJava.Compile -fserverTIE My.idl
java com.ibm.idl.toJava.Compile -fallTIE My.idl
```

For the interface My, these commands will generate client-side bindings plus a tie-model Skeleton, _MySkeleton.java. The constructor to _MySkeleton takes a My. You must provide the implementation for My (_MyImpl), which does not have to extend any other class; it must, however, implement the My interface. To use _MyImpl with the ORB, you must wrap it within _MySkeleton. For instance:

```
_MyImpl myImpl = new _MyImpl ();
_MySkeleton skel = new _MySkeleton (myImpl);
orb.connect (skel);
```

The reason you might want to use the tie model over the typical inheritance model is if your implementation must inherit some class other than the Skeleton. Java allows any number of interface inheritance, but there is only one slot for class inheritance. If you use the inheritance model, that slot is occupied by the Skeleton. By using a tie model Skeleton, that slot is freed up for your own use. The drawback is that it introduces a level of indirection: one extra method call occurs when invoking a method.

Specifying an Alternate Location for Emitted Files

By default, the compiler outputs bindings to the directory from which it was invoked (the current directory). To direct the output to another directory, specify the target directory immediately following the -td flag. The target directory may be absolute or relative. For example, to direct the output to directory /my/bindings while compiling My.idl, you would invoke the compiler with the following command:

```
java com.ibm.idl.toJava.Compile -td /my/bindings My.idl
```

Similarly, if /my is the current directory, you could direct the output to /my/bindings by issuing the command:

```
java com.ibm.idl.toJava.Compile -td ./bindings My.idl
```

Specifying Alternate Locations for Include Files

If My.idl included another idl file, MyOther.idl, the compiler assumes that MyOther.idl resides in the local directory. If it resides in directory */includes*, for example, you would invoke the compiler with the following command:

```
java com.ibm.idl.toJava.Compile -i /includes My.idl
```

If My.idl also included Another.idl that resided in /moreIncludes, then you would invoke the compiler as:

```
java com.ibm.idl.toJava.Compile -i /includes -i /moreIncludes My.idl
```

You can begin to see that if you have a number of places where included files may come from, the command will become long and unmanageable. So there is another means of indicating to the compiler where to search for included files. This technique is very similar to the idea of an environment variable. You must create a file called idl.config in a directory that is listed in your CLASSPATH. Inside of idl.config you must provide a line of the following form:

```
includes=/includes;/moreIncludes
```

The compiler take the first version of the file it locates and read in its includes list. Note that in this example, the separator character between the two directories is a semicolon (;). This separator character is platform dependent: On NT it is a semicolon, on AIX it is a colon, and so on.

Note: Some platforms will fail when issuing a long command line. If the command line to invoke the compiler becomes too long, use the idl.config file.

Emitting Bindings for Include Files

By default, only those interfaces, structs, and so on, that are defined in the idl file on the command line have the Java bindings generated for them. The types defined in included files are not generated. For example, assume the following two idl files:

```
My.idl

#include MyOther.idl
interface My
{
};
```

```
MyOther.idl

interface MyOther
{
};
```

The following command will only generate bindings for types within My:

```
java com.ibm.idl.toJava.Compile My.idl
```

To generate bindings for all of the types in My.idl and all of the types in files that My.idl includes (in this example, MyOther.idl), use the following command:

```
java com.ibm.idl.toJava.Compile -emitAll My.idl
```

There is a caveat to the default rule. #include statements which appear at the global scope are treated as described. These #include statements can be thought of as import statements. #include statements which appear within some enclosing scope are treated as true #include statements, meaning that the code within the included file is treated as if it appeared in the original file and, therefore, Java bindings are emitted for it. Here is an example:

```
My.idl

#include MyOther.idl
interface My
{
  #include Embedded.idl
};
```

```
MyOther.idl

interface MyOther
{
};


Embedded.idl

enum E {one, two, three};
```

Running the following command:

```
java com.ibm.idl.toJava.Compile My.idl
```

will generate the following list of Java files:

```
./MyHolder.java
./MyHelper.java
./_MyStub.java
./MyPackage
./MyPackage/EHolder.java
./MyPackage/EHelper.java
./MyPackage/E.java
./My.java
```

Notice that MyOther.java was not generated because it is defined in an import-like #include. But E.java was generated because it was defined in a true #include. Notice also that since Embedded.idl was included within the scope of the interface My it appears within the scope of My (that is, in MyPackage).

If the -emitAll flag were used in the previous example, all types in all included files would be emitted.

Inserting Package Prefixes

Say you work for a company called ABC and that company has constructed the following IDL file:

```
Widgets.idl

module Widgets
{
  interface W1 {...};
  interface W2 {...};
};
```

Running this file through the IDL-to-Java compiler will place the Java bindings for W1 and W2 within the package Widgets. But there is an industry convention that states that a company's packages should reside within a package named com.*company name*. The Widgets package is not good enough. To follow the convention, it should be com.abc.Widgets. To place this package prefix onto the Widgets module, execute the following:

```
java com.ibm.idl.toJava.Compile -pkgPrefix Widgets com.abc Widgets.idl
```

You should be aware that, if you have an IDL file which includes Widgets.idl, the -pkgPrefix flag must appear on that command as well. If it does not, then your IDL file will be looking for a Widgets package rather than a com.abc.Widgets package.

If you have a number of these packages that require prefixes, it might be easier to place them into the idl.config file described above. Each package prefix line should be of the form:

```
PkgPrefix.type=prefix
```

So the line for the above example would be:

```
PkgPrefix.Widgets=com.abc
```

Emitting Client-side Bindings as JavaBeans

You might like the client-side bindings to be JavaBeans:

```
java com.ibm.idl.toJava.Compile -bean My.idl
```

This will generate another file, _MyProxy.java, which can be used as a JavaBean, and _MyProxyBeanInfo.java which is the auxiliary bean file. All Beans generated in this manner have three properties: _initialHost, _initialPort, _name. These properties must be filled in with the server bootstrap host, bootstrap port, and name the server-side object was bound to in the Name Server. When these are filled in, _initProxy() must be called. When that returns, _MyProxy is a usable proxy.

Note that this proxy only works with an object that has already been bound with the Name Server. At the point when this was implemented, the Lifecycle service was not fully defined, so there is no way in which a Bean can create an object on a server (unless the developer writes one). Expect a create method to appear on Bean proxy's in the future.

Emitting Object by Value (Stateful) Bindings

CORBA Objects are passed by reference, never by value. There is a proposal before OMG to add the ability to pass an object by value. This compiler has attempted to guess what this mechanism will be. Since it is only a guess, and the proposal is still changing, pass-by-value objects should not be depended upon unless you are willing to change in the future. To pass an object by value, it must have state. The following is an example of the changes made to the IDL language to support stateful objects:

```
StatefulA.idl

interface A
{
  state
  {
    long x;
    short y;
  };
  void proc1 ();
};
```

Within the curly braces, the syntax for the state block is identical to the syntax for a struct. To direct the compiler to correctly parse this code, specify the -stateful option when invoking the compiler, like so:

```
java com.ibm.idl.toJava.Compile -stateful StatefulA.idl
```

A new interface file will be generated: _AState.java. To correctly interact with the other bindings (Helpers, Holders, Stubs, Skeletons), _AState must be used instead of A. This is because A does not contain the state while _AState does. And the other bindings all depend on the state to marshal and demarshal the stateful data.

Emitting Makefiles and Specifying the Path Separator Character

When the Java bindings will be compiled using a makefile, it can become tedious to build the makefile by hand. There are two arguments to the IDL-to-Java compiler which help in building the makefile.

```
java com.ibm.idl.toJava.Compile -m My.idl
```

This will generate, besides the usual bindings, file My.u which will contain the following lines:

```
MyHelper.java: My.idl
My.java: My.idl
MyHolder.java: My.idl
MyPackage/E.java: Embedded.idl
MyPackage/EHelper.java: Embedded.idl
MyPackage/EHolder.java: Embedded.idl
_MyStub.java: My.idl

MyHelper.java \
My.java \
MyHolder.java \
MyPackage/E.java \
MyPackage/EHelper.java \
MyPackage/EHolder.java \
_MyStub.java
```

If you are building a makefile that will run on multiple platforms, the slash '/' character will not necessarily be the file separator character. Perhaps the build environment has a special variable for the file separator character. If this variable were $(Sep), then the compiler can place this in place of the slash in My.u with the following command:

```
java com.ibm.idl.toJava.Compile -m -sep \$\(Sep\) My.idl
```

So that My.u now contains:

```
MyHelper.java: My.idl
My.java: My.idl
MyHolder.java: My.idl
MyPackage$(Sep)E.java: Embedded.idl
MyPackage$(Sep)EHelper.java: Embedded.idl
MyPackage$(Sep)EHolder.java: Embedded.idl
_MyStub.java: My.idl

MyHelper.java \
My.java \
MyHolder.java \
MyPackage$(Sep)E.java \
MyPackage$(Sep)EHelper.java \
MyPackage$(Sep)EHolder.java \
_MyStub.java
```

Defining Symbols Before Compilation

You may wish to define a symbol for compilation that is not defined within the idl file, perhaps to include debugging code in the bindings. The command

```
java com.ibm.idl.toJava.Compile -d MYDEF My.idl
```

is the equivalent to putting the line '#define MYDEF' inside My.idl itself.

Preserving Pre-existing Bindings

If the Java binding files already exist, this argument will keep the compiler from overwriting them. The default is to generate all files without considering if they already exist. If you've customized those files (which you should not do unless you are very comfortable with their contents), then the -keep option is very useful. The command

```
java com.ibm.idl.toJava.Compile -keep My.idl
```

emit all client-side bindings that do not already exist.

Viewing Progress of Compilation

The IDL-to-Java compiler will generate status messages as it progresses through its phases of execution. Use the -v option to activate this "verbose" mode:

```
java com.ibm.idl.toJava.Compile -v My.idl
```

By default the compiler does not operate in verbose mode.

The idl2com Command

The idl2com Command Creates usage and implementation bindings for interfaces described in IDL files.

The Interface Definition Language Compiler-to-COM (idl2com) command compiles one file containing CORBA 2.0-compliant IDL statements, and produces generated language bindings appropriate to the named input file.

The syntax of the idl2com command is:

```
idl2com [options] <-g GUID_VALUE> <filename>
```

Where:

[options]
: See Options for the idl2com Command for a complete list of the options, their usage, and their restrictions.

<**-g GUID_VALUE>**
: -g is a mandatory parameter. See Options for the idl2com Command for a complete description of this parameter.

<**filename**>
: <filename> is the name of a file containing IDL definitions. It is a mandatory parameter and must appear last. It must be specified with a file name extension. For example, the following are acceptable ways to refer to the file "Policy.idl" in directory "E:\idl\src":

```
E:\idl\src\Policy.idl
Policy.idl (if E:\idl\src is the current directory)
```

If no parameters are specified, idl2com will write its syntax and options to standard output.

When the specified IDL file is compiled, the idl2com command returns a java exception if an error has occurred. Warnings may also be given. For instance, if the IDL contains a constant of type float or double, idl2com issues a warning statement to standard output and continues processing. The result is that the definition for the constant in the .bas file will be lacking, otherwise the bindings will be complete.

For additional information on emitted files, see Emitted File Names.

Options for the idl2com Command

Options to the idl2com command may be specified in any combination. The `-g` option is mandatory, all other options are optional. The options may appear in any order. The IDL filename is required and must appear last. A maximum of 9 parameters can be specified, including the IDL filename. If more than 9 parameters need to be specified, the user can resort to invoking java directly. Examine the contents of idl2com.bat to see how to do this. If no parameters are specified, idl2com will write its syntax and options to standard output.

With the exception of the IDL filename, options to the idl2com command are preceded with a dash (-) character and must be specified individually. For example, `-emitAll -keep -v`. Some options accept an argument. The space between this type of option and its argument is mandatory. For example, `-d DEBUG`. All options are case-sensitive, even on platforms where file names are not case-sensitive.

Table 12 describes each available option:

Table 12. idl2com Command Options

Option	**Description**
-d <symbol>	This is equivalent to the following line in an IDL file: #define <symbol>
-emitAll	Emit all types, including those found in included files. The default is to emit only the types that are part of the IDL being processed.
-g <GUID>	The GUID seed to be used for GUID generation in the registry file format of: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx This value is generated by using the guidgen.exe utility included with Microsoft Visual C++. See Generating Interfaces Using the idl2com Command for more information.
-i <include path>	By default, the current directory is scanned for included files. This option adds another directory. Multiple -i <include path> options may be specified.
-keep	If a file to be generated already exists, do not overwrite it. By default it is overwritten.
-v	Verbose mode. Default is non-verbose mode.

Generating Interfaces Using the idl2com Command

When an IDL file is processed by the idl2com command, a unique GUID seed value (-g parameter) is required. This GUID is used to register within the Windows system registry the various interfaces, etc produced by idl2com. Many interfaces may be contained within the IDL, and idl2com uses the provided GUID parameter as a starting point for the to-be registered interfaces. If multiple items produced from the IDL file need to have GUIDs, idl2com increments using the first 8 digits (AE3E2131 in the example below) as needed. idl2com will use all eight of those digits when incrementing and will fail if it hits “FFFFFFFF.” It will not roll over to 00000000 since this is the start of reserved GUID ranges held by Microsoft. For example, if the IDL file results in three interfaces being registered, the following GUIDs would be used:

- AE3E2131-C6DE-11d0-92AF-08005ACE818D
- AE3E2132-C6DE-11d0-92AF-08005ACE818D
- AE3E2133-C6DE-11d0-92AF-08005ACE818D

This is important to know because all registered items need to be unique within the registry. Use the guidgen.exe program included with Microsoft Visual C++ to provide the -g parameter, but be careful not to conflict with the internally-generated values that idl2com will use based on the -g parameter. To avoid this potential conflict, you need to exit and restart guidgen.exe prior to generating a new GUID for another run of idl2com. Be sure to include a space between the -g and the GUID value on the idl2com command line.

Emitted File Names

The idl2com command processes an IDL file and produces output files that contain language-specific usage and implementation bindings for the IDL interface. The list of generated files is dependent on the contents of the IDL. However, the following is a general list of what can be expected given an IDL file called Policy:

Table 13. idl2com Emitted Files

File Name	Content
Policy.odl	the Object Description file for the IDL
Policy.def	definition file
Policy.bas	Visual Basic file containing any constants that were defined in the IDL
Policy.mak	makefile
Policy.rc	resource definition
*.cpp	a set of implementation files which will vary based on the contents of the IDL
*.h	a set of header files which will vary based on the contents of the IDL

The names of the generated .cpp and .h files vary depending on the contents of the IDL.

Data Type Restrictions

The idl2com command has the following limitations:

- IDL constants of datatype float or double are not supported.
- IDL types long long, unsigned long long, long double and fixed are not supported.

idl2com Generated Makefile

As part of the idl2com processing, a makefile is automatically generated. The makefile is named .mak. In most cases it will be complete. If however, the developer extends the code in a way that is not reflected within the IDL, the makefile may require some additional hand coding. Also, if the IDL makes use of other IDL whose bindings are linked into a different .dll, the developer will have to modify the generated .mak file to add the other .LIB to the list of .LIBs.

Appendix C. C++ CORBA Programming

This appendix includes the following C++ topics:

C++ Bindings

CORBA 2.0 specifies standard forms by which client C++ code can manipulate data whose types are described using IDL. C++ bindings that support these forms are termed *compliant* and client code that uses (only) these forms is termed *conformant*. The bindings in Component Broker are compliant.

C++ Bindings for Constants

Constants can be defined within the IDL either of the following ways:

- Within the module or interface
- Globally

Within the Module

An IDL constant declaration contained with a module or an interface is mapped to a static constant data member of the C++ class to which the module or interface is mapped. For example, consider the following IDL:

```
module M
{
   const string name = "testing";
};
```

After compiling the client bindings, a C++ programmer could use the following expression to denote the previous `name` constant:

```
M::name
```

Globally

Globally-defined constants are mapped to static data local to a single compilation unit. For example, if the following IDL appeared globally, un-nested within a module or interface:

```
const string name = "testing";
```

The code that includes the corresponding .hh file refers to the name constant using the expression `name`.

CORBA Types and Business Objects

Most of the CORBA types are straightforward and can be easily used in business objects. Other CORBA types are more difficult to use but are still useful if care is taken.

Basic Types

The basic C++ types are mapped directly into CORBA types. These include:

- Boolean
- Char
- Double
- Enum (enumerations)
- Float
- Long
- Octet (hexadecimal)
- Short
- Struct
- UShort (unsigned short)
- ULong (unsigned long)
- WChar (wide character)

All types are scoped to the class CORBA and must be declared accordingly. They are used transparently to C++ and are straightforward. For example:

```
CORBA::Short aShortvariable;
...
aShortVariable = 12;
...
```

Types and Object References

Other CORBA types are more complex to use because they return object references to the caller. It is the responsibility of the caller to manage these object references and their associated memory. There are two facilities provided by CORBA to do this:

A_var This is most frequently used by client code because it is a *smart* pointer and automatically releases its object reference when it is deallocated or when assigned a new object reference. This is the most straightforward and safest approach to managing these types.

Note: You should avoid declaring C++ **Static** variables as _var. The _var holds a reference to an object. During process termination, this object could reference another object that was removed before termination processing is completed for this Static type. As a result, the _var could reference an invalid address or null pointer and thereby cause bad termination.

A_ptr This is a pointer type and provides the most basic object reference, which has similar semantics to a standard C++ pointer.

The CORBA types that return object references include:

- Any
- Array
- Sequence
- String
- Union
- WString (wide string)

C++ Bindings for Data Types

The following are C++ bindings for data types:

- “Any Type” on page 301
- “Array Types” on page 306
- “Atomic Data Types” on page 308
- “Enums” on page 308
- “Sequence Types” on page 309
- “Strings” on page 312
- “Struct Types” on page 314
- “Union Types” on page 315
- “Using WStrings” on page 316

Any Type

The purpose of the IDL “any” type is to encapsulate data of some arbitrary IDL type. The C++ bindings provide a C++ class named CORBA::Any that provides this functionality. A CORBA::Any object encapsulates a void* pointer and a CORBA::TypeCode object that describes the thing pointed to by the void*.

The Any type can be used with many of the CORBA types and is useful when different types can be used that are unknown to the receiver of the data or as a common storage mechanism for passing a variety of types. It is used easily with many of the CORBA types but has a unique method of redirection operators for setting and retrieving data.

The following types are handled in this manner:

- Double
- Enumerations
- Float
- Long
- Short
- ULong
- UShort
- Unbounded Strings
- Object References

```
::CORBA::Any anything;
anything <<= (::CORBA::Long) 123456;
::CORBA::Long anythingStart  = 123456;
::CORBA::Long anythingLongResult = 0;
policyVar->anything(anything);

::CORBA::Any_var anythingResult_var(policyVar->anything());
::CORBA::Any anythingResult(anythingResult_var);
anythingResult >>= anythingLongResult;
if ( anythingStart != anythingLongResult)
{
   cout << "Anything not set" << endl;
      return 1;
}
else
{
   cout << "Anything set correctly..." << endl;
}
```

There are also specialized structures provided for the following types for conversion with Any:

- Boolean
- Char
- Octet
- String

The data in an Any object is initialized and accessed using insertion (<<=) and extraction (>>=) operators defined by the C++ bindings. These operators are provided (using overloading) by CORBA::Any for each primitive data type, and are provided by the generated C++ bindings for each user-defined IDL type. As a result, there is usually no need to indicate a typecode when inserting or extracting data from a CORBA::Any (although the CORBA::Any class does provide methods for manipulate the data using an explicit TypeCode).

Types that cannot be distinguished by C++ overloading are inserted into and extracted from Any's using special *wrapper* classes. These wrapper classes are not transparent to the application; the application must explicit create and use them when inserting/extracting ambiguous types into/from Any's. For primitive IDL types that do not map to distinct C++ types (boolean, octet, and char), the wrapper classes are defined within the CORBA::Any scope; they are called from_boolean, to_boolean, from_octet, to_octet, from_char, and to_char. For information on the scope see “IDL Name Scoping” on page 271. Because bounded strings cannot be distinguished in C++ from unbounded strings, CORBA::Any provides the from_string and to_string wrapper classes, for inserting/extracting bounded strings. For extracting object references from Any's as the base CORBA::Object type, CORBA::Any provides a to_object wrapper class.

For application-specific arrays, the bindings provide a special *forany* class, for inserting/extracting the array into/from an Any. For example, given the following IDL array definition:

```
typedef long LongArray[4][5];
```

the emitted bindings define the following:

```
typedef CORBA::Long LongArray[4][5];
typedef CORBA::Long LongArray_slice[5];
typedef LongArray_slice* LongArray_slice_vPtr;
typedef const LongArray_slice* LongArray_slice_cvPtr;
class LongArray_forany
{
   public:
   LongArray_forany();
   LongArray_forany(LongArray_slice*, CORBA::Boolean nocopy=0);
   LongArray_forany(const LongArray_forany&);
   LongArray_forany & operator= (LongArray_slice*);
   LongArray_forany & operator= (const LongArray_forany &);
   ~LongArray_forany();
   const LongArray_slice& operator[] (int) const;
   const LongArray_slice& operator[] (CORBA::ULong) const;
   LongArray_slice & operator[] (int);
   LongArray_slice & operator[] (CORBA::ULong);
   operator LongArray_slice_cvPtr () const;
   operator LongArray_slice_vPtr& ();
};
void operator<<=(Any&, const LongArray_forany &);
CORBA::Boolean operator>>=(Any&, LongArray_forany &);
```

Note that the nocopy optional parameter of the _forany's second constructor indicates whether the _forany makes a copy of the input array or assumes ownership of it. (The default is for the _forany to assume ownership of the input array; that ownership will then be transferred to the Any when the _forany is inserted into the Any.)

To determine what kind of data is in Any, invoke the type method on a CORBA::Any to access a TypeCode that describes the data it holds. Alternatively, you can simply try to extract data of a particular type from the Any; the extraction operator returns a boolean to indicate success. If the extraction operation fails, the Any doesn't hold data of the type you tried to extract.

A CORBA::Any object always owns the data that its void* points to, and deletes (or releases) it when the Any is given a new value or deleted. The only question is whether this data is a copy of the data that was inserted into the Any. When primitives (including strings and enums) are inserted, a copy is made, and a copy is returned when the data is extracted.

For non-primitive (constructed) data, extraction from an Any always updates a pointer (owned by the caller) so that it points to the data owned by the Any. The caller should not, therefore, free this data or reference it after the Any has been given a new value or deleted. For constructed IDL type T, the emitted bindings define the following extraction operator:

```
  CORBA::Boolean operator>>=(Any&, T*&);
```

When a reference to constructed data is inserted into an Any (when the C++ syntax looks as if you are inserting a value instead of a pointer) a copy is made. In this case, the caller retains ownership of the original data. For example, for constructed type T and interface I, the emitted bindings define the following insertion operators, which copy (or, in the case of object references, _duplicate) the inserted value):

```
  void operator<<=(Any&, const T&);
  void operator<<=(Any&, T_ptr);
```

When a pointer to constructed data is inserted into an Any, as when using the following insertion operators emitted for type T and interface I:

```
  void operator<<=(Any&, T*);
  void operator<<=(Any&, T_ptr*);
```

The Any takes ownership of the constructed type's top-level storage only; however, the Any makes no copy of the top-level storage or any embedded storage. All further use of the pointer that was inserted is forbidden; the Any now owns it and is free to delete it at any time. The next time data is inserted into the Any, or when the Any is destroyed, the Any deletes the previously-inserted pointer. However, if the constructed type consists of multiple dynamically-allocated regions of memory, only the top-level storage is deleted. (The Any deletes arrays using a single array delete; other constructed types are deleted using a single, normal delete.) Further, the top-level storage is deleted as a void*, rather than its true type, which means that the constructed type's destructor will not be run. Due to these restrictions, insertion by pointer of constructed types into an Any should be used with caution.

In summary, when extracting data from an Any, the caller does own the data for primitive types, but does not own the data for constructed types. When inserting data into an Any, the caller retains ownership of the data for primitive types, for constructed types inserted by value, and for storage embedded within constructed types inserted by pointer. The caller does not retain ownership of the top-level contiguous storage for a constructed type inserted into an Any by pointer.

The followng is an example that illustrates the previously discussed aspects of CORBA::Any usage. The IDL for this example appears immediately below. It defines a struct and an array that will be inserted into an Any.

```
  Module M
  {
     Struct S
     {
        string str;
        longlng;
     };
```

```
   typedef long long1[2][3];
}
```

A C++ program illustrating Any insertion and extraction appears below:

```
#include <stdio.h>
#include <any_C.cpp>
main()
{
   CORBA::Any a;  // the Any that we'll be using
   // test a long
   long l = 42;
   a <<= l;
   if (a.type()->equal(CORBA::_tc_long))
   {
      long v;
      a >>= v;
      printf("the any holds a long = %d\n", v);
   }
   else
   printf("failure: long insertion\n");
   // test a string
   char *str = "abc";
   a <<= str;
   if (a.type()->equal(CORBA::_tc_string))
   {
      char *ch;
      a >>= ch;
      printf("the any holds the string = %s\n", ch);
      delete ch;
      a >>= ch;
      printf(" the any still holds the string = %s\n", ch);
      delete ch;
   }
   else
   printf("failure: string insertion\n");
   // test a bounded string -- note we don't use a typecode here
   char *bstr = "abcd";
   char *rstr;
   a <<= CORBA::Any::from_string(bstr, 6);
   if (a >>= CORBA::Any::to_string(rstr,6))
   printf("the any holds a bounded string<6> = %s\n", rstr);
   else
   printf("failure: bounded string insertion\n");

   // test a user-defined struct
   M::S *s1 = new M::S;
   char *saveforlater = CORBA::string_dup("abc");
   s1->str = saveforlater;
   s1->lng = 42;
   a <<= s1; // insertion by pointer
   if (a.type()->equal(_tc_M_S))
   {
      M::S *sp;
      a >>= sp;
      printf("the any holds an M::S = {%s, %d}\n", sp->str, sp->lng);
   }
   else
```

```
    printf("failure: struct insertion by pointer\n");
    M::S s2;
    s2.str = CORBA::string_dup("def");
    s2.lng = 23;
    a <<= s2; // note: this deletes *s1, but not saveforlater
    printf("saveforlater still = %s\n", saveforlater);
    CORBA::string_free(saveforlater);
    if (a.type()->equal(_tc_M_S))
    {
       M::S *sp;
       a >>= sp;
       printf("the any holds an M::S = {%s, %d}\n", sp->str, sp->lng);
    }
    else
    printf("failure: struct insertion by value\n");
    M::S_var s3 = new M::S;
    s3->str = CORBA::string_dup("ghi");
    s3->lng = 96;
    a <<= *s3;
    if (a.type()->equal(_tc_M_S))
    {
       M::S *sp;
       a >>= sp;
       printf("the any holds an M::S = {%s, %d}\n", sp->str, sp->lng);
    }
    else
    printf("failure: struct insertion by ref to value\n");
    // test an array
    M::long1_var l1v =  M::long1_alloc();
    for (i=0;i<2;i++)
    for (j=0;j<3;j++)
    l1v[i][j] = (i+1)*(j+1);
    a <<= M::long1_forany(l1v);
    if (a.type()->equal(_tc_M_long1))
    {
       M::long1_forany l1s;
       a >>= l1s;
       printf("the any holds the array: ");
       for (i=0;i<2;i++)
       for (j=0;j<3;j++)
       printf("%d ",l1s[i][j]);
       printf("\n");
    }
    else printf("failure: array insertion\n");
 }
```

Output from the above program is:

```
the any holds a long = 42
the any holds a string = abc
the any still holds a string = abc
the any holds a bounded string<6> = abcd
the any holds an M::S = {abc, 42}
saveforlater still = abc
the any holds an M::S = {def, 23}
the any holds an M::S = {ghi, 96}
the any holds the array: 1 2 3 2 4 6
```

Array Types

An IDL array type is mapped to the corresponding C++ array definition. There is also a corresponding _var type. For example, given the following IDL definition:

```
typedef long LongArray [4][5];
```

The C++ bindings provide the following definitions:

```
typedef CORBA::Long LongArray[4][5];
typedef CORBA::Long LongArray_slice[5];
typedef LongArray_slice* LongArray_slice_vPtr;
typedef const LongArray_slice* LongArray_slice_cvPtr;
class LongArray_var
{
   public:
   LongArray_var();
   LongArray_var(LongArray_slice*);
   LongArray_var(const LongArray_var&);
   LongArray_var & operator= (LongArray_slice*);
   LongArray_var & operator= (const LongArray_var &);
    LongArray_var();
   const LongArray_slice& operator[] (int) const;
   const LongArray_slice& operator[] (CORBA::ULong) const;
   LongArray_slice & operator[] (int);
   LongArray_slice & operator[] (CORBA::ULong);
   operator LongArray_slice_cvPtr () const;
   operator LongArray_slice_vPtr& ();
};
LongArray_slice * LongArray_alloc();
void LongArray_free (LongArray_slice*);
LongArray_slice * LongArray_dup (const LongArray_slice*);
```

As shown above, array mappings provide alloc, dup, and free functions (for allocating, copying, and freeing array storage), as static member functions of the class within which the array type name is scoped. The alloc function dynamically allocates an array, which can be later freed using the free function. The dup function dynamically allocates an array and copies the elements of an existing array into it. A NULL pointer can be passed to the free function. None of these functions throws exceptions.

The type of the pointer returned from LongArray_alloc is LongArray_slice*. The C++ bindings define array “slice” types for all arrays declared in IDL. The reason is that using the name LongArray in a program doesn't denote the array LongArray; rather, it denotes a pointer to the array. For historical reasons (related to the fact that arrays are not an actual data type in C and C++) the type of this pointer has one less array dimension than the array LongArray. Thus, the bindings for LongArray include the following typedef:

```
typedef string LongArray_slice[5];
```

Hence, LongArray_slice* is the correct type for a pointer to an array of IDL type LongArray.

As with structs and sequences, arrays use special auxiliary classes for automatic storage management of String and object reference elements. The auxiliary classes for Strings and object references manage the storage just as the associated _var classes do.

When assigning a value to an array element that is an object reference, the assignment operator will automatically release the previous value (if any). When assigning an object reference pointer to an array element, the array assumes ownership of the pointer (no _duplicate is done), and the application should no longer access the pointer directly. (If this is not the desired behavior, then the caller can explicitly _duplicate the object reference before assigning it.) However, when assigning to an object reference array element from a _var object or from another struct, union, array, or sequence member (rather than from an object reference pointer), a _duplicate is done automatically.

For an array of Strings, when assigning a value to an element or deleting the array, any previously held (non-null) char* is automatically freed. As when assigning to String_vars, assigning a char* to a string element does not make a copy, but assigning a const char * or another struct/union/array/sequence String element does make a copy. One should never assign a string literal (for example, “abc”) to a String array element without an explicit cast to “const char*.” When assigning a char* that occupies static storage (rather than one that was dynamically allocated), the caller can use CORBA::string_dup to duplicate the string before assigning it.

The following is an example that involves multidimensional arrays, and array_vars, from the IDL snippet immediately below:

```
typedef string s2_3[2][3];
typedef string s3_2[3][2];
```

The code that exercises the C++ arrays that correspond to the above IDL is shown below. Notice that in the following example:

- There is no need to explicitly use slice types when working with the array _var types, because the bindings declare the pointer held by an array _var type using the appropriate slice type.
- At the end, the program explicitly frees the storage pointed to by s2_3p (using an array delete operator), but does not do this for s3_2v, because its pointer is deleted when the destructor for s3_2v is executed. (This is the purpose of the _var types.)

```
#include <arr_C.cpp>
#include <stdio.h>
main()
{
   int i,j;
   char id[40];
   // create arrays
   s2_3_slice* s2_3p = s2_3_alloc();
   s3_2_var s3_2v = s3_2_alloc();
   // load the arrays
   for(i=0; i<2; i++)
   {
      for (j=0; j<3; j++)
      {
         sprintf(id, "s2_3 element [%d][%d]",i,j);
         // Use string_dup when assigning a char*
         // if you don't want the array to own the original:
         s2_3p[i][j] = CORBA::string_dup(id);
      }
   }
   for(i=0; i<3; i++)
   {
      for (j=0; j<2; j++)
      {
         sprintf(id, "s3_2_var element [%d][%d]",i,j);
         // Use string_dup when assigning a char*
         // if you don't want the array to own the original:
         s3_2v[i][j] = CORBA::string_dup(id);
      }
   }
   // print the array contents
   for(i=0; i<2; i++)
```

```
    {
       for (j=0; j<3; j++)
       {
          printf("%s\n", s2_3p[i][j]);
       }
    }
    for(i=0; i<3; i++)
    {
       for (j=0; j<2; j++)
       {
          printf("%s\n", s3_2v[i][j]);
       }
    }
    delete [] s2_3; // needed to prevent a storage leak.
    // Nothing is needed for s3_2v, because
    // it is a _var type.
 }
```

Output from the above program is:

```
s2_3 element [0][0]
s2_3 element [0][1]
s2_3 element [0][2]
s2_3 element [1][0]
s2_3 element [1][1]
s2_3 element [1][2]
s3_2_var element [0][0]
s3_2_var element [0][1]
s3_2_var element [1][0]
s3_2_var element [1][1]
s3_2_var element [2][0]
s3_2_var element [2][1]
```

Atomic Data Types

The atomic IDL data types (long, short, unsigned long, unsigned short, float, double, char, boolean, and octet) are mapped into types defined in corba.h, nested within the CORBA scope. See “IDL Name Scoping” on page 271 for more information. The first letter of the mapped type is capitalized. For example, to introduce and initialize a local variable named `Myvar` whose type corresponds to the IDL type named long, a C++ programmer could employ the following expression:

```
CORBA::Long Myvar = 1;
```

The mapping for the IDL boolean type (CORBA::Boolean) defines only the values 0 and 1. The unsigned long and unsigned short IDL types are mapped to CORBA::ULong and CORBA::UShort, respectively.

Enums

An IDL enum is mapped to a corresponding C++ enum. For example, given the following IDL:

```
module M
{
   enum Color
   {
      red, green, blue
   };
};
```

A C++ programmer could introduce a local variable of the corresponding C++ type and initialize it with the following code:

```
{
   M::Color MYCOLOR = M::red;
}
```

The enumeration constant red is not denoted using the expression M::Color::red. For this reason, names of enumeration constants must be carefully chosen.

Sequence Types

An IDL sequence type is mapped to a C++ class that behaves like an array with a current length (how many elements have been stored) and a maximum length (how much storage is currently allocated). The array indexing operator [] is used to read and write sequence elements. (Indexing begins at zero.) It is the programmer's responsibility to check the current sequence length or maximum to prevent accessing the sequence beyond its bounds. The length and maximum of the sequence are not automatically increased to accommodate new elements; the programmer must explicit increase them.

The maximum length of a bounded sequence is implicit in the sequence class's type and cannot be changed. The initial maximum length of an unbounded sequence is set to zero by the default constructor, or can be initialized by the programmer using a non-default constructor. Setting the maximum of an unbounded sequence using the non-default constructor causes storage to be allocated for the specified number of sequence elements.

Sequence classes provide an overloaded member function length that either returns or sets the length of the sequence. Setting the length of an unbounded sequence to a value larger than the current maximum causes the sequence to allocate new storage of the required size, copy any previous sequence elements to the new storage, free the old storage (if any), and reset the maximum to the new length. Sequence classes also provide allocbuf and freebuf member functions for explicitly allocating/freeing the sequence's storage buffer. Decreasing a sequence's length does not cause any storage to be deallocated, but any orphaned sequence elements are no longer accessible, even if the sequence length is subsequently increased.

Sequences may or may not manage (own) the storage that contains their elements, and the elements themselves By default, a sequence manages this storage, but a *release* constructor parameter allows client programmers to request otherwise (when passing in a buffer explicitly allocated using the allocbuf function).

The following IDL:

```
typedef sequence s1; // unbounded sequence
```

is mapped to the following C++ sequence class:

```
class s1
{
   public:
   s1();// default constructor
   s1(CORBA::ULong max);// "max" constructor
   s1(CORBA::ULong max, CORBA::ULong length,
   T* data, CORBA::Boolean release=0);
   // "data" constructor
   s1(const s1&);// copy constructor
   s1 &operator= (const s1&);   // assignment operator
    s1();// destructor
   CORBA::ULong maximum() const;
   CORBA::ULong length() const;
```

```
    void length(CORBA::ULong len);
    T& operator[] (CORBA::ULong index);
    const T& operator[] (CORBA::ULong index) const;
    static T* allocbuf(CORBA::ULong nelems);
    static void freebuf(T* data);
};
```

The default constructor sets the length and maximum to zero. (For a bounded sequence, the default constructor sets the maximum to the sequence bounds and allocates storage for the maximum number of elements, which the sequence owns.)

The “max” constructor sets the initial sequence maximum and allocates a storage buffer for the specified number of sequence elements, which the sequence owns. The length of the sequence is initialized to zero. (This method is not available for bounded sequences.)

The “data” constructor sets the initial length and maximum of the sequence, as well as its initial contents. (For bounded sequences, the maximum cannot be set by the “data” constructor.) The input storage should match the specified sequence maximum. Ownership of the input storage is indicated by the “release” parameter. Passing release=1 specifies that the storage was allocated using s1::allocbuf, that the sequence should delete the storage and the sequence elements when the sequence is deleted or when the storage needs to be reallocated, and that the caller will not directly access the storage after the call (since the sequence is free to delete it at any time). In general, sequences constructed with release=0 should not be passed as inout parameters, because the callee must assume that the sequence owns the sequence elements.

The copy constructor creates a new sequence with the same maximum and length as the input sequence and copies the sequence elements to storage that the sequence owns. The assignment operator performs a deep copy, releasing the previous sequence elements if necessary. It behaves as if the destructor were run, followed by the copy constructor.

The destructor destroys each of the sequence elements (from zero through length-1), if the sequence owns the storage.

The allocbuf function allocates enough storage for the specified number of sequence elements; the return value can then be passed to the “data” constructor. Each sequence element is initialized using its default constructor; string elements are initialized to NULL; object reference elements are initialized to nil object references. NULL is returned if storage cannot be allocated for any reason. If ownership of the allocated buffer is not transferred to a sequence using the “data” constructor with release=1, the buffer should be subsequently freed using the freebuf function. The freebuf function insures that each sequence element's destructor is run (or, for strings, that CORBA::string_free is called, or for object references, that CORBA::release is called) before the buffer is deleted. The freebuf function ignores NULL pointers passed to it. Neither allocbuf nor freebuf throw CORBA exceptions.

As with structs, sequences that manage their elements use special auxiliary classes for automatic storage management of String and object reference sequence elements. These auxiliary classes manage Strings and object references just as the associated _var classes do.

For a storage-managing sequence whose elements are object references, when assigning a value to an element, the assignment operator will automatically release the previous value (if any). When assigning an object reference pointer to such a sequence element, the sequence assumes ownership of the pointer (no _duplicate is done), and the application should no longer access the pointer directly. (If this is not the desired behavior, then the caller can explicitly _duplicate the object reference before assigning it to the sequence element.) However, when assigning to such an object reference sequence element from a _var object or from another struct, union, array, or sequence (rather than from an object reference pointer), a _duplicate is done automatically.

For a storage-managing sequence whose elements are Strings, when assigning a value to such an element or deleting the sequence, any previously held (non-null) char* is automatically freed. As when assigning to String_vars, assigning a char* to a string element doesn't make a copy, but assigning a const char *, a String_var, or another struct/union/array/sequence String member does automatically make a copy. Thus, one should never assign a string literal (such as “abc”) to such an element without an explicit cast to const char*. When assigning a char* that occupies static storage (rather than one that was dynamically allocated), the caller can use CORBA::string_dup to duplicate the string before assigning it.

There is a corresponding _var type defined for every sequence class.The _var type for a sequence provides an overloaded operator[] that forwards the operator the underlying sequence.

Following is an example that illustrates loading and accessing the elements of a sequence. This example illustrates a recursive sequence (whose entries are structs of the same type that contain the sequence). The IDL for the example is shown below:

```
struct S
{
   long sf1;
   sequence sf2;
};
typedef sequence Sseq;
```

The following is an example program that creates and loads a sequence of type Sseq and then prints out its contents.

```
#include <seq_C.cpp>
#include <stdio.h>
main()
{
   int i,j;
   Sseq seq;      // create an Sseq
   seq.length(3); // set length of seq to 3
   for (i=0; i<3; i++) { // index the three S structs in seq
   seq[i].sf1 = i;    // place a number in the i-indexed struct
   seq[i].sf2.length(i+1); // set length of the sequence in
   //   the i-indexed struct
   for (j=0; j<i+1; j++) // index the i+1 S structs in the sequence
   //  in the i-indexed struct
   seq[i].sf2[j].sf1 = (i+1)*10+j; // place a number in
   //   the j-indexed struct
}
// OK. Print out what we have created!
printf("seq = (%d sequence elements)\n", seq.length());
for (i=0; i<3; i++)
{
   printf("   struct[%d] = {\n", i);
   printf("      sf1 = %d\n", seq[i].sf1);
   printf("      sf2 = (%d sequence elements)\n",
   seq[i].sf2[j].length());
   for (j=0; j<i+1; j++)
   {
      printf("         struct[%d] = \n",j);
      printf("            sf1 = %d\n", seq[i].sf2[j].sf1);
      printf("            sf2 = (%d sequence elements)\n",
      seq[i].sf2[j].sf2.length());
      printf("         }\n");
      }
   printf("   }\n");
```

```
      }
  }
```

Note that the above program never explicitly constructs any data of type S, even though the sequences contain structs of this type. The reason is that when a sequence buffer is allocated, default constructors are run for each of the buffer elements. So, when the above program sets the length of a sequence of S structs (either at the top level for the seq variable, or for the sf2 field of an S struct in seq), the resulting buffer is automatically populated with default structs of type S.

The output from the above program is:

```
  seq = (3 sequence elements)
     struct[0] = {
        sf1 = 0
        sf2 = (1 sequence elements)
           struct[0] = {
              sf1 = 10
              sf2 = (0 sequence elements)
           }
     }
     struct[1] = {
        sf1 = 1
        sf2 = (2 sequence elements)
           struct[0] = {
              sf1 = 20
              sf2 = (0 sequence elements)
           }
           struct[1] = {
              sf1 = 21
              sf2 = (0 sequence elements)
           }
     }
     struct[2] = {
        sf1 = 2
        sf2 = (3 sequence elements)
           struct[0] = {
              sf1 = 30
              sf2 = (0 sequence elements)
           }
           struct[1] = {
              sf1 = 31
              sf2 = (0 sequence elements)
           }
           struct[2] = {
              sf1 = 32
              sf2 = (0 sequence elements)
           }
     }
```

Strings

The mapping for strings is provided by corba.h, within the CORBA scope. See "IDL Name Scoping" on page 271 for more information. The user-visible types are CORBA::String and CORBA::String_var. CORBA::String is a typedef name for char*. The CORBA::String_var class performs storage management of a dynamically allocated CORBA::String. The following functions are for dynamic allocation/deallocation of memory to hold a String:

- CORBA::string_alloc

- CORBA::string_free
- CORBA::string_dup

A String_var object behaves as a char*, except that when it is assigned to, or goes out of scope, the memory it points to is automatically freed by CORBA::string_free. When a String_var is constructed or assigned from a char*, the String_var assumes ownership of the string and the caller should no longer access the string directly. (If this is not the desired behavior, as when the char* occupies static storage, the caller can use CORBA::string_dup to copy the char* before assigning it.) When a String_var is constructed or assigned from a const char*, another String_var, or a String element of a struct, union, array, or sequence, an automatic copy of the source string is done. The String_var class provides subscripting operations to access the characters within the embedded string.

C++ compilers don't treat a string literal (such as “abc”) as a const char* upon assignment; given both a const and a non-const assignment operator, the compiler will choose the non-const operator. As a result, when assigning a string literal to a String_var, no copy of the string into dynamically allocated memory is made; the pointer “owned” by the String_var will point to memory that cannot be freed. Thus, string literals should not be assigned to a String_var without an explicit cast to const char*.

Some examples using String_var objects are:

```
// first some supporting functions for the examples
char* f1()
{
   return "abc";
}
char* f2()
{
   char* s=CORBA::string_alloc(4);strcpy(s,"abc");return s;
}
// then the examples
void main()
{
   CORBA::String_var s1;
   if (0) s1 = f1();// Wrong!! The pointer can't be freed and
   // no copy is done.
   if (0) s1 = "abc";  // Also wrong, for the same reason.
   const char* const_string = "abcd"; // *const_string can't be changed
   s1 = const_string; // OK. A copy of the string is made because
   // it is const, and the copy can be freed.
   CORBA::String_var s3 = f2();// OK. no copy is made, but f2
   // returns a string that can be freed
   CORBA::String_var s4 = CORBA::string_alloc(10); // also OK. no copy
   s4 = s3; // s4 will use string_free followed by string_dup
   long l4 = strlen(s4); // l4 will receive 3
   long l1 = strlen(s1); // l1 will receive 4
   if (l4 >= l1)
   strcpy(s4,s1); // OK, but only because of the condition.
   // note that s4's buffer only has size=4.
   s4 = const_string; // OK. s4 will use string_free followed by
   // string_dup. The copy is made because String_vars
   // must reference a buffer that can be modified.
}
// The s1, s3 and s4 destructors run successfully, freeing their buffers
```

Struct Types

An IDL struct type is mapped to a corresponding C++ struct whose field names correspond to those in the IDL declaration, and whose field types support access and storage of the C++ types corresponding to the IDL struct field types. Dynamically allocated storage used to hold such a C++ struct must be allocated and freed using the C++ new and delete operators.

When a new struct is created, the default constructor for each of its fields executes. Object reference fields are initialized to nil references, and String fields are initialized to NULL. When the struct is deleted (or goes out of scope), the destructor for each of its fields executes. The (default) copy constructor performs a deep copy, including duplicating object references; the (default) assignment operator acts as the destructor followed by the copy constructor.

The actual types of the fields in the C++ struct to which an IDL struct is mapped may be auxiliary classes for the purpose of storage management. In particular, String and object reference field types are auxiliary classes that manage Strings and object references in the same way that the associated _var classes do. Although client code should not depend on the names of these auxiliary classes, the client code does need to know that struct fields containing Strings and object references are managed by these auxiliary classes.

When assigning a value to a struct field that is an object reference, the assignment operator for the struct field will automatically release the previous value (if any). When assigning an object reference pointer to a struct member, the struct member assumes ownership of the pointer (no _duplicate is done), and the application should no longer access the pointer directly. (If this is not the desired behavior, then the caller can explicitly _duplicate the object reference before assigning it to the struct member.) However, when assigning to an object reference struct member from a _var object or from another struct, union, array, or sequence member (rather than from an object reference pointer), a _duplicate is done automatically.

When assigning a value to a struct field that is a String, or when the struct is deleted or goes out of scope, any previously held (non-null) String is automatically freed. As when assigning to String_vars, assigning a char* to a String field does not make a copy, but assigning a const char *, a String_var, or another struct/union/array/sequence String member does automatically make a copy. One should never assign a string literal (for example, “abc”) to a String struct member without an explicit cast to “const char*.” When assigning a char* that occupies static storage (rather than one that was dynamically allocated), the caller can use CORBA::string_dup to duplicate the string before assigning it.

As with all constructed types, a _var type is provided for managing an instance of the C++ struct that corresponds to an IDL struct. When assigning one struct's _var to another, the receiving _var deletes its current pointer (thus running all contained destructors), and creates a new struct to hold the assignment result, which is initialized using copy constructors for each of the contained fields. Thus, for example, if the source struct has an object reference field, the struct _var assignment will automatically duplicate this reference.

The IDL that follows is used in the succeeding example, which shows both correct and incorrect ways to to create and manipulate the corresponding C++ struct and the corresponding _var type :

```
Interface A
{
   struct S
   {
      string f1;
      A      f2;
   };
};
```

The following code illustrates both correct and incorrect ways to create and manipulate the corresponding C++ struct and the corresponding _var type.

```
{
   A::S_var sv1 = new A::S;
   A::S_var sv2 = new A::S;
   // sv1->f1 = "abc"; -- Wrong! f1 can't free this pointer later
   sv1->f1 = CORBA::string_alloc(20);
   A_ptr a1 =  // get an A somehow
   A_ptr a2 =  // get an A somehow
   sv1->f2 = a1; // a1 still has ref cnt = 1
   sv2->f1 = CORBA::string_alloc(20);
   sv2->f2 = a2; // a2 still has ref cnt = 1
   sv1 = sv2; // This runs copy ctors, and increments a2's ref cnt.
   // Also, a1's ref count is decremented.
   sv1->f1 = sv2->f1;
}
```

Union Types

Union fields are not directly accessible to C++ programmers. Instead, the C++ mapping for IDL unions defines a class that provides accessor methods for the union discriminator and the corresponding union fields. The union discriminator accessor is named _d. The union field accessors are named using the IDL union field names and are overloaded to allow both reading and writing.

Setting a union's value using a field accessor automatically sets the discriminator, and releases the storage associated with the previous value, if any. It is an error for an application to attempt to access the union's value through an accessor that does not match the current discriminator value. It is also an error for the application to use the discriminator modifier method to implicitly switch between difference union members.

Unions with implicit default members (those that do not have an explicit default case and do not list all possible values of the discriminator as cases) provide a _default method, for setting the discriminator to a legal default value. This method causes the union's value to be composed only of the legal default value, since there is no explicit default member in this case.

A _var type is defined, for managing a pointer to a union in dynamically allocated memory.

To illustrate the C++ bindings for IDL unions, consider the following IDL:

```
module A
{
   interface X
   {
   };
   union U switch (long)
   {
      case 1: long u1;
      case 2: string u2;
      case 3: X u3;
   };
};
```

The following code illustrates usage of the C++ bindings corresponding to the previous IDL:

```
{
   X_ptr x = // get an X somehow
   A::U_var uv = new A::U;
   uv.u2((const char*) "testing");  // sets the discriminator to 2
   // and copies the string
   if (u._d() == 2)// the condition evaluates to true
   u.u1(23); // frees the string, and sets the discriminator to 1
   if (u._d() == 1) // the condition evaluates to true
   u.u3(x);  // duplicates x and sets discriminator to 3
}
```

The default constructor of a union class does not initialize the discriminator or the union members, so the application must initialize the union before accessing it. The copy constructor and assignment operator perform deep copies. The assignment operator and destructor release all storage owned by the union.

With respect to memory management, accessor and modifier methods for union members work similarly to those for struct members. Modifier methods make a deep copy of their input when passed by value (for simple types) or by reference (for constructed types). Accessor methods that return a non-const reference can be used by the application to update a union member's value, but only for struct, union, sequence, and any members.

The modifier method for a string union member makes a copy when given a const char* or a String_var, but not when given a char*. As shown in the example above, a string literal should not be assigned to a union without an explicit “const char*” cast. The accessor method for a string union member returns a const char*, therefore the string union member cannot be modified. (This is done to prevent the string union member from being assigned to a String_var, resulting in memory management errors.)

The modifier method for an object reference union member always duplicates the input object reference and releases the previous object reference value, if any. The accessor method for an object reference union member does not duplicate the returned object reference, because the union retains ownership of it.

The accessor method for an array union member returns a pointer to the array slice. The application can thus read or write the union-member array elements using subscript operators. If the union member is an anonymous array (one without an explicit type name), the union defines (typedefs) the slice type, by cocatenating a leading underscore and appending “_slice” to the union member name.

Using WStrings

The WString type provides support for wide strings. It is fairly comparable to using strings except for type declarations and assignments:

```
#include <wcstr.h> // For WChar and WString support
...
const wchar_t* wcomments = L"This policy looks pretty good...";
wchar_t* wcommentsResult=::CORBA::wstring_alloc(wcslen(wcomments));
::CORBA::WString_var wcommentsResult_var(wcommentsResult);
policyVar->wcomments(wcomments);

if (!wcscmp(wcommentsResult_var, wcomments) )
{
   cout << "Wcomments not set" << endl;
      return 1;
}
else
{
   cout << "Wcomments set correctly..." << endl;
```

```
  }
  wcommentsResult = policyVar->wcomments();
```

Exceptions

The preferred coding practice for handling errors in C++ and Java is by using Exceptions. The programming model and CORBA support this coding practice using the standard try and throw logic of exception handling. Handling exceptions is a critical part of the programming model. The exceptions that are thrown must be understood and handled appropriately by application developers.

Note: See a standard C++ book for further information regarding exceptions and their usage.

No matter how much care an object provider takes in implementing a business object, there are times when things go wrong. In these cases, a business object might need to throw an exception to the client to give the client the opportunity to recover from the error.

Which Exceptions to Use

CORBA exceptions are used to communicate between business objects and client applications. Specific rules must be followed regarding which CORBA Exceptions to use. There are several abstract CORBA exception classes defined:

- CORBA::Exception
- CORBA::UserException
- CORBA::SystemException

CORBA::Exception: This is the abstract class that is the base of all CORBA exceptions. Because this class is abstract, it is never thrown. However, it can be used in catch blocks to process all CORBA exceptions in one block.

CORBA::UserException: This is the abstract class for all CORBA user exceptions and is a subclass of CORBA::Exception. This class should be used as the base class of all user-defined exception classes. The contents of these classes have no special format. Methods that throw these classes must declare their usage in IDL using the raises clause.

CORBA::SystemException: This is the abstract class for all CORBA standard exceptions and is a subclass of CORBA::Exception. These exceptions can be thrown by any method regardless of the interface specification. Standard exceptions cannot be listed in raises expressions, therefore whether or not an interface throws a system exception is unknown. This means you should be prepared to handle standard exceptions on all method calls. Each standard exception includes a minor code to provide more detailed information. This field is used by Component Broker components. The definition of the minor code values is included in the online documentation.

Note: CORBA standard exceptions are a predefined list of exceptions. These can be thrown from any method. CORBA has defined the class that provides this support as CORBA::System Exception. See *The Common Object Request Broker: Architecture and Specification* for further information regarding CORBA exceptions.

Throwing Exceptions

A business object might wrapper existing logic which might not be written in C++ or might not use the exception paradigm. These business objects must convert the existing exceptions or error return codes to CORBA exceptions that can be returned to the client program.

Any non-CORBA exception thrown by the business object is automatically mapped to CORBA::UNKNOWN by the framework. This does not provide specific information to the client and severely limits the error recovery capability of the client program. These C++ exceptions should be mapped to appropriate CORBA exceptions by the business object.

Catching Exceptions

It is a requirement to handle exceptions in client programs. Remember that any method might throw a standard exception, therefore an exception can be thrown by these methods at any time – even if there are no exceptions declared in the raises clause of that method. The default behavior for uncaught exceptions is to terminate that process. If this happens, suspect an uncaught exception first. The exact style of how or what exceptions are caught depends on what the client application does for error recovery but there are some good general rules to follow:

- Perform as specific error recovery as makes sense. By proper structuring of catch clauses specific error recovery can be done.
- Check for most specific exceptions first, most general exceptions last.
- Make use of information that is available in the exception. All CORBA exceptions support the .id() method that returns the exception identifier. System exceptions also provide .minor() and .completed() methods which return the minor code and completion status respectively.

A Simple Client Code Example

```
try
{
   // Some real code goes here
   foo.boo();
}
// Catch any specific User exceptions defined for the method in the
//  raises' clause
catch (IManagedClient::INoObjectWKey &nowk)
{
   // Process the error, more specific recovery could be made here
   // because the specific error is known
}
// Catch and process any other specific User exceptions
...
// Catch any other User exceptions defined for the method in the
//  raises' clause
catch (CORBA::UserException &ue)
{
   // Process any other User exceptions. Use the .id() method to
   // record or display useful information
   cout << "Caught a User Exception: " << ue.id() << endl;
}
// Catch any System exceptions defined for the method in the
//  raises' clause
catch (CORBA::SystemException &se)
{
   // Process any System exceptions. Use the .id(), and .minor()
   // methods to record or display useful information
   cout << "Caught a System Exception: " << ue.id() << ": " << ue.minor() << endl;
}
catch (...)
{
   // Process any other exceptions. This would catch any other C++
   // exceptions and should probably never occur
   cout << "Caught an unknown Exception" << endl;
}
```

Specific standard exceptions cannot be caught individually. If processing individual standard exceptions is required it can be done within the CORBA::SystemException catch block and using the .id() method.

Name Scoping and Modules in the C++ Bindings

IDL scoped names are mapped to C++ scopes as follows.

- In the IBM C++ bindings, IDL modules are, by default, mapped to C++ classes of the same name. If the programmer using the bindings #defines _USE_NAMESPACE before including the bindings, then the bindings map the IDL module to a C++ namespace of the same name. IDL definitions occurring within a module are mapped to corresponding C++ definitions within the C++ module class or namespace.
- IDL interfaces are mapped to C++ classes. All IDL constructs defined within an interface are mapped to corresponding C++ definitions within the C++ interface class.
- Every use in IDL of a C++ keyword (such as “class”) is mapped into the same identifier with a leading underscore.

C++ Bindings for Interfaces

The CORBA 2.0 C++ client bindings define a variety of C++ types corresponding to a single IDL interface. Specifically, an IDL interface I is mapped to C++ types with the following four names: I, I_ptr, IRef, I_var. The types named I and I_var are classes. The types I_ptr and IRef are unspecified by CORBA, but are required to name the same type; these types are the C++ form for an object reference. (The use of the IRef types will be removed by the CORBA 2.0 specification.)

The class I is referred to as the interface class corresponding to IDL interface named I; the C++ mappings of the typedefs, operations, and constants defined within the IDL interface I appear publicly within the C++ interface class I. For example, an IDL operation that accepts an in parameter of interface type I is mapped to a C++ virtual member function of the class named I that has a parameter of type I_ptr. Similarly, an IDL operation in I that returns data of type I is mapped to a C++ member function in the class I that returns data of type I_ptr.

As with other user-defined IDL types, the I_var type is used to assist storage management. Specifically, an I_var type holds an I_ptr and can be used as if it were an I_ptr. When a I_var type is assigned a new value or when it goes out of scope, it releases the I_ptr it is holding at that time.

The CORBA 2.0 specification prohibits CORBA-compliant applications from:

- Explicitly creating an instance of an interface class, as in:

  ```
  I my_instance;   // NOT ALLOWED!
  I_ptr my_instance = new I;  // NOT ALLOWED!
  ```

- Declaring a pointer (I*) or a reference (I&) to an interface class.

Instead, the I_ptr, IRef, and I_var types must be used to hold object references, and object references can only be created (by client applications) by invoking methods that return object references. The interface class (I) is used by client applications only as a name scope.

IDL operations defined in (or inherited by) interface I are invoked in C++ using the arrow (->) operator on either an I_ptr, IRef, or I_var type.

Nil object references of type I_ptr can be obtained using a static member function of I called _nil(). Operations cannot be invoked on nil object references. The CORBA::is_nil function is the only

CORBA-conformant way to determine whether a given object reference is nil. CORBA::release can be invoked on a nil object reference, but is not needed. The _duplicate and _narrow functions defined by the C++ bindings can be given a nil object reference.

In the IBM C++ bindings, the CORBA-prescribed types are implemented as follows:

1. The interface class for I is derived using virtual inheritance from the interface classes for I's IDL parents. When I has no IDL parents, its interface class is derived using virtual inheritance from CORBA::Object. Types, constants, and operations declared within the I interface are mapped to types, constants, and member functions declared within the corresponding interface class.
2. The object reference types I_ptr and IRef are typedef'd to I* (for example, an I_ptr points to an object of type I). However, CORBA 2.0 specifies that treating an I_ptr as a C++ pointer (e.g., using conversion to void*, arithmetic and relational operators, test for equality) is not conformant, although this is not enforced by the bindings.
3. An instance of I addressed is called a proxy, and is created by a proxy factory object of class ProxyFactory. For each interface I, the bindings define an ProxyFactory class, and provide a global instance of this class with the name _ProxyFactory.
4. Nil object references are represented as NULL pointers (but CORBA 2.0 conformant applications should not assume so, and should instead use the _nil() and is_nil() functions to manipulate nil object references).
5. The I_var class introduces an instance variable of type I_ptr. The purpose of an I_var object is to handle release operations on the I_ptr that it holds.
6. An auxiliary class I_SeqElem class is used to return sequence elements, and is similar to the I_var class. It is returned from array access operations on an IDL type sequence. An I_SeqElem is different from an I_var in that it must honor the release setting of the sequence from which it is selected (that is, it only owns the object that its I_ptr references if it was taken from a sequence that owns its buffer storage).

Managing Object References

The mapping for interface I defines a static member function named _duplicate that takes as input an object reference of type I_ptr and returns an object reference of type I_ptr (potentially the same reference, when reference counting is employed, as is the case with IBM C++ bindings). The CORBA::release function indicates that the caller will no longer access the object reference, and the resources associated with the object reference can be deallocated. (In the IBM C++ bindings, an object reference is only deleted when its reference count falls to zero, that occurs only if CORBA::release is called for each _duplicate or _narrow performed on the object reference.)

Duplicating an object reference using _duplicate is analogous to copying a string before transferring ownership of it, and releasing an object reference is analogous to deleting a string that is no longer needed. Unlike strings, object references cannot be directly copied or deleted by the client programmer; object references are managed by the ORB and can only be duplicated or released by the application.

Widening Object References

If interface A is a (direct or indirect) base of interface B, the following assignments do not require an explicit C++ cast.

- B_ptr to an A_var
- B_ptr to A_ptr
- B_ptr to Object_var
- B_ptr to Object_ptr

- B_var to A_ptr
- B_var to Object_ptr

B_var cannot be assigned to A_var or a compile-time error occurs. To assign B_var to A_var:

- Use B::_duplicate on B_var to create B_ptr.
- Assign B_ptr to A_var.

Narrowing Object References

The mapping for an interface I defines a static member function named _narrow that takes as input an object reference of any type (for example, an Object_ptr) and returns an object reference of type I_ptr. If the referenced object (the actual implementation object corresponding to the proxy addressed by the input object reference) does not support the I interface, the result is NULL; otherwise, the I_ptr addresses an object that also supports the I interface. In the case where the proxy addressed by the input argument does not support interface I and the actual implementation object does, the I_ptr returned by I::_narrow addresses a different proxy object than the input argument.

The _narrow static member function does an implicit _duplicate of the input argument. Therefore, the caller is responsible for releasing both the object reference input to _narrow and the return result.

Narrowing to a C++ Implementation

Given an interface pointer to an object, it is sometimes useful to narrow to the implementation pointer of the object. For example, given interface I, the C++ implementation hierarchy for I might look like:

```
           I
           ↑
           |
       I_Skeleton
           ↑
           |
        I_Impl
```

You might want to convert a pointer to I into a pointer to I_Impl.

There is no CORBA-prescribed mechanism for this conversion. Within the confines of the C++ language, dynamic cast can be used. Since Component Broker does not require the C++ compiler to support dynamic cast, a second mechanism is provided, with the virtual _narrow_impl() method defined in CORBA::Object:

```
class Object
{
   ...
   virtual void *_narrow_impl(const char* impl_name = NULL)
   {
      return NULL;
   }
};
```

The default implementation of this method in CORBA::Object returns NULL, so that implementations that do not support _narrow_impl() need not worry about its existence. Implementations that support _narrow_impl() should override it, and provide an alternate implementation. The impl_name parameter is optional, and may be used by the implementation to perform sanity checks.

For the example with interface I, I_Impl might look like:

```
   class I_Impl: virtual public I_Skeleton ...
   {
      ...
      void * _narrow_impl(const char* impl_name)
      {
         if (impl_name != NULL && !strcmp(impl_name, "I_Impl"))
         return this;
         else
         return NULL;
      }
   }
```

A user of this service would do the following to issue the narrow:

```
  I_ptr i = ...;        /* get the IDL ptr somehow */
  I_Impl * ii = (I_Impl *) i->_narrow_impl("I_Impl");
  if ( ii == NULL )
  {
     // problem
  }
  else
  {
     // Use ii
     ...
  }
```

Storage Management and _var Types

The C++ bindings try to make the programmer's storage management responsibility as easy as possible. One aspect of this is the “_var” types. For each user-defined structured IDL type T (struct, union, sequences, and arrays) and for interfaces, the bindings generate both a class T and a class T_var. The classes CORBA::String and CORBA::Any also have corresponding CORBA::String_var and CORBA::Any_var classes.

The essential purpose of a _var object is to hold a pointer to dynamically allocated memory. A _var object can be used as if it were a pointer to the IDL type for which it is named; special constructors, assignment operators, and conversion operators make this work in a way that is invisible to programmers. The memory pointed to by a _var object is always considered to be owned (managed) by the _var object, and when the _var object is deleted, goes out of scope, or is assigned a new value, it deletes (or, in the case of an object reference, releases) the managed memory.

A typical _var object is declared by a programmer as an automatic (stack) variable within a code block, and is then used to receive an operation result or is passed to an operation as an out parameter. Later, when the code block is exited, the _var object destructor runs and its managed memory is deleted (or, for object references, released).

When a pointer (rather than another _var object or struct/union/array/sequence element) is assigned to (or used to construct) a _var object, this pointer should point to dynamically allocated memory, because the _var object does not make a copy; it assumes ownership of the pointer and will later delete it (or, for object references, release it). The single exception is that pointers to const data can be assigned to a _var object. When this occurs, the _var object dynamically allocates new memory and copies the const data into the new memory. A pointer assigned to a _var object must not be “owned” by some other data structure, and the pointer should not be subsequently used by the application except by the _var object.

The default constructor for a _var type loads the contained pointer with NULL. You must assign a value to a _var object created by a default constructor before invoking methods on it, just as you must assign a value to a pointer variable before invoking methods on it.

The copy constructor and _var assignment operator of a _var type perform a deep copy of the source data. The copy is later deallocated (or released, in the case of object references) when the _var is destroyed or when it is assigned a new value.

The following is the typical form for a T_var class, emitted for an IDL—constructed data type named T:

```
class T_var
{
   public:
   T_var ();
   T_var (T*);
   T_var (const T_var&);
   ~T_var ();
   T_var &operator= (T*);
   T_var &operator= (const T_var &);
   T * operator-> const ();
   ...
};
```

Argument Passing Considerations for C++ Bindings

Rules must be observed when passing parameters to a CORBA object implementation. These rules are dependent on a combination of the IDL type of the argument and the argument mode (in, inout, out or return value), and must be followed to:

- Ensure the required access authority.
- Prevent memory leaks.
- Ensure that the allocation and deallocation of memory is performed consistently.

C++ Type Mapping for Argument Passing

The type used to pass the parameters of a method signature is dependent on the IDL type and the directionality of the parameter (in , inout, out, or return value). The rules are dictated by CORBA OMG IDL to C++ mapping. These mapping rules are captured in and enforced by the header files produced when an IDL interface description is compiled. Some rules cannot be enforced by the bindings. For example, parameters that are passed or returned as a pointer type (T*) or reference to pointer(T*&) should not be passed or returned as a null pointer. Memory management responsibilities cannot be enforced by the bindings. Client (caller) and implementation (callee) programmers must understand and implement according to these rules. The argument type mappings are discussed in the following paragraphs and summarized in the table, Table 14 on page 324. Memory management responsibilities are discussed in "Storage Management Responsibilities for Arguments" on page 325.

For primitive types and enumerations, the type mapping is straightforward. For in parameters and return values the type mapping is simply the C++ type representation (abbreviated as "T" in the text that follows) of the IDL specified type. For inout and out parameters the type mapping is a reference to the C++ type representation (abbreviated as "T&" in the text that follows).

For object references, the type mapping uses _ptr for in parameters and return values and _ptr& for inout and out parameters. That is, for a declared interface A, an object reference parameter is passed as type A_ptr or A_ptr&. The conversion functions on the _var type permit the client (caller) the option of using _var type rather than the _ptr for object reference parameters. Using the _var type may have an advantage in that it relieves the client (caller) of the responsibility of deallocating a returned object reference (out parm or return value) between successive calls. This is because the assignment operator of a _ptr to a _var automatically releases the embedded reference.

The type mapping of parameters for aggregate types (also referred to as complex types) are complicated by when and how the parameter memory is allocated and deallocated. Mapping in parameters is straightforward because the parameter storage is caller allocated and read only. For an aggregate IDL type t with a C++ type representation of T the in parameter mapping is const T&. The mapping of out and inout parameters is slightly more complex. To preserve the client capability to stack allocate fixed length types, OMG has defined the mappings for fixed-length and variable-length aggregates differently. The inout and out mapping of an aggregate type represented in C++ as T is T& for fixed-length aggregates and as T*& for variable-length aggregates.

Table 14. Basic Argument and Result Passing

Data	**Type In**	**Inout**	**Out**	**Return**
short	short	short&	short&	short
long	long	long&	long&	long
unsigned short	ushort	ushort&	ushort&	ushort
unsigned long	ulong	ulong&	ulong&	ulong
float	float	float&	float&	float
double	double	double&	double&	double
boolean	boolean	boolean&	boolean&	boolean
char	char	char&	char&	char
wchar	wchar	wchar&	wchar&	wchar
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum
object reference ptr	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct, fixed	const struct&	struct&	struct&	struct
struct, variable	const struct&	struct&	struct*&	struct*
union, fixed	const union&	union&	union&	union
union, variable	const union&	union*&	union*&	union*
string	const char*	char*&	char*&	char*
wstring	const char*	char*&	char*&	char*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, fixed	const array	array	array	array slice*
array, variable	const array	array	array slice*&	array slice*
any	const any&	any&	any*&	any*

For an aggregate type represented by the C++ type T, the T_var type is also defined. The conversion operations on each T_var type allows the client (caller) to use the T_var type directly for any directionality, as opposed to using the required form of the T type (T, T& or T*&) The emitted bindings define the operation signatures in terms of the parameter passing modes shown in Table 15 on page 325, and the T_var types provide the necessary conversion operators to allow them to be passed directly.

Table 15. T_var Argument and Result Passing

Data Type	In	Inout	Out	Return
object referenc var	const object_var&	objref_var&	objref_var&	objref_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
sequence_var	const sequence_var&	sequence_var&	sequence_var&	sequence_var
sequence_var array_var	const array_var&	any_var&	any_var&	any_var

For parameters that are passed or returned as a pointer type (T*) or reference to pointer(T*&) the programmer should not pass or return a null pointer. This cannot be enforced by the bindings.

Storage Management Responsibilities for Arguments

Table 16 summarizes the storage access and allocation responsibilities for argument passing. As an overall requirement when allocating and deallocating argument storage, the storage allocation rules for the specific type must be followed. Specifically, for strings, sequences, and arrays or for aggregate types composed of these types, the associated memory allocation and dealloaction functions must be used. For string types this means the use of string_alloc(), string_dup(), and string_free(), for sequence types this means the use of allocbuf() and freebuf() and for arrays this means the use of T_alloc(), T_dup() and T_free(). The memory deallocation responsibilities of the client can be minimized by stack allocation and the use of the _var types when that is possible. When an argument is passed or returned as a pointer type, a NULL pointer value should never be passed or returned.

Table 16 (Page 1 of 2). Argument Storage Responsibilities

Data Type	Inout	Out	Return
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enum	1	1	1
object reference pointer	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3

Table 16 (Page 2 of 2). Argument Storage Responsibilities

Data Type	Inout	Out	Return
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3

in parameters

The caller (client) must allocate the input parameters. The callee (implementation) is restricted to read access. The caller is responsible for the eventual release of the storage. Primitive types and fixed-length aggregate types may either be heap allocated or stack allocated. By their nature, variable-length aggregates cannot be completely stack allocated.

inout parameters

For inout parameters, the caller provides the initial value and the callee may change that value. For primitive types and fixed-length aggregates this is straight forward. The caller provides the storage and the callee overwrites the storage on return. For variable-length aggregates the size of the contained data provided on input may differ than the size of the contained data provided on output. Therefore, the callee is required to deallocate any input contained data that is being replaced on output with callee allocated data. For object references, the caller provides an initial value: if the callee reassigns the value the callee must first release the original input value. The callee assumes or retains ownership of the returned parameters and must eventually deallocate or release them.

out parameters

For primitive types and fixed-length aggregate types, the caller allocates the storage for the out parameter and the callee sets the value. For variable-length aggregate types, the caller allocates a pointer and passes it by reference and the callee sets the pointer to point to a valid instance of the parameter's type. For object references the caller allocates storage for the _ptr and the callee sets the _ptr to point to a valid instance.

Because a pointer to an array in C++ must actually be represented as a pointer to the array element type, CORBA defines an array_slice type, where a slice is an array with all the dimensions of the original except the first. The output parameter is typed as a reference to an array_slice pointer. The caller allocates the storage for the pointer and the callee updates the pointer to point to a valid instance of an array_slice.

The caller assumes or retains ownership of the output parameter storage and must eventually deallocate it or, in the case of object references, release it.

return values

For primitive types and fixed-length aggregate types, the caller allocates the storage for the return value and the callee returns a value for the type. For variable-length aggregate types, the caller allocates a pointer and the callee returns a pointer to an instance of the type. For object references the caller allocates storage for the _ptr and the callee returns a _ptr that points to a valid object instance.

As a pointer to an array in C++ must actually be represented as a pointer to the array element type, the array_slice type is used for returning array values. The caller allocates storage for a pointer to the array_slice and the callee returns a pointer to a valid instance of an array_slice.

The caller assumes or retains ownership of the storage associated with returned values and must eventually deallocate it or, in the case of object references, release it.

For definitions of the numerical values in Table 16 on page 325, refer to Table 17 on page 327.

Table 17. Argument Passing Cases

Case	Description
1	Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.
2	Caller allocates storage for the object reference. For inout parameters, the caller provides an initial value; if the callee wants to reassign the inout parameter, it will first call **CORBA:release** on the original input value. To continue to use an object reference passed in as an inout, the caller must first duplicate the reference. The caller is responsible for the release of all out and return object references. Release of all object references embedded in other structures is performed automatically by the structures themselves.
3	The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following the completion of a request, the caller is not allowed to modify any values in the returned storage: in order to do so, the caller must first copy the returned instance into a new instance, then modify the new instance.
4	For inout strings, the caller provides storage for both the input string and the char* pointing to it. Since the callee may deallocate the input strings and reassign the char* to point to new storage to hold the output value, the caller should allocate the input string using **string_alloc()**. The size of the out string is therefore not limited by the size of the in string. The caller is responsible for deleting the storage for the out using **string_free()**. The callee is not allowed to return a null pointer for an inout, out or return value.
5	For inout sequences and **any's**, assignment or modification of the sequence or any may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the Boolean release parameter with which the sequence or **any** was constructed.
6	For out parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following completion of a request, the caller is not allowed to modify any values in the returned storage: in order to do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance.

C++ Client Bindings

In these bindings, the client usage picture for the IDL types declared in the file T.idl appears as follows. Bold lines enclose files that are generated from IDL. Double lines enclose files that would normally be produced by a programmer or development tool.

The corba.h header file defines the C++ mappings for primitive IDL data types and other types required by the bindings, within a scope called CORBA. For more information about the scope see “IDL Name Scoping” on page 271. These types are implemented in a shared library that can be linked with a client application. A client application is created by compiling/linking emitted bindings and client code to produce an executable file.

The C++ bindings for the IDL types defined in the file T.idl are represented by a set of declarations in the emitted T.hh header file. The classes declared in T.hh that support client code are implemented by the

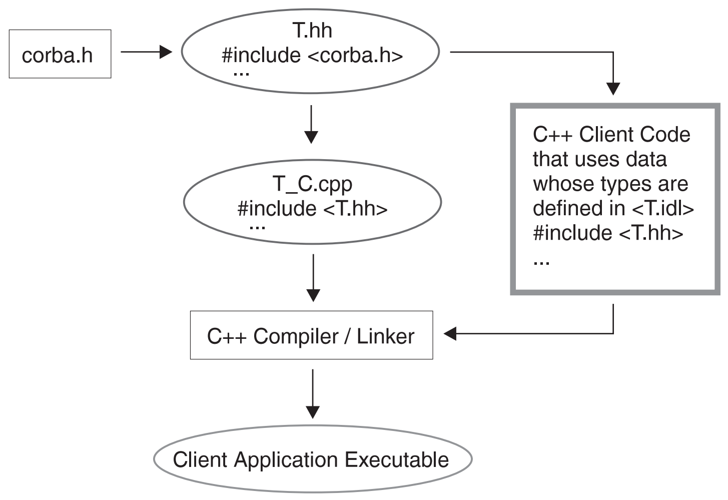


Figure 66. Client Usage Picture for IDL Types

code emitted into a corresponding T_C.cpp implementation file. The pair of files T.hh and T_C.cpp thus collectively provide the client bindings for T. (To minimize the number of generated files, some types used by servers are also declared in T.hh).

In general, the C++ bindings map non-primitive IDL types to C++ classes that implement constructors, destructors, assignment operators, and other functionality. Auxiliary classes are also sometimes defined, such as classes to automate storage management for array elements, sequence elements, and structure and union fields. The names of these auxiliary classes are not specified by CORBA, because specially-designed conversion operators and copy constructors hide their existence from client code. These classes are not of interest to programmers that use the bindings.

C++ Server Bindings

To allow IDL interfaces to be implemented in C++, server-side bindings are emitted. The resulting classes work with the client bindings. The following figure illustrates the module structure of the server-side bindings, assuming that interface T is declared in the file T.idl.

The differences between this figure and the client-side figure presented in Client C++ Bindings are:

- An emitted T_S.cpp file that provides server-side implementation bindings is compiled.
- Server-side C++ code (written by a programmer) defines the implementation for the operations introduced and inherited by the T interface.

The T_S.cpp file provides an implementation for the class T_Dispatcher. This class inherits from the dispatcher classes corresponding to T's parents. An instance of this class contains a T_ptr that addresses the T_Impl object upon which it will dispatch operation invocations. Each target object (for example, each T_Impl instance) exported by a server must have a corresponding dispatcher object, whose purpose is to receive a CORBA::Request object, determine what method is being invoked, stream the method arguments out into local variables, invoke the method on the target object, then stream the results back into the request so these can be returned to the caller.

The target object for a T_Dispatcher is an instance of the T_Impl class, which subclasses from (at least) the T_Skeleton class defined by the implementation bindings (in the file T.hh). The T_skeleton class inherits from the T interface class and the skeleton classes corresponding to T's parents. As a result, T_skeleton inherits all the methods that T_Impl must support. Furthermore, this is done in a way that forces T_Impl to indeed provide implementations for all of these methods.

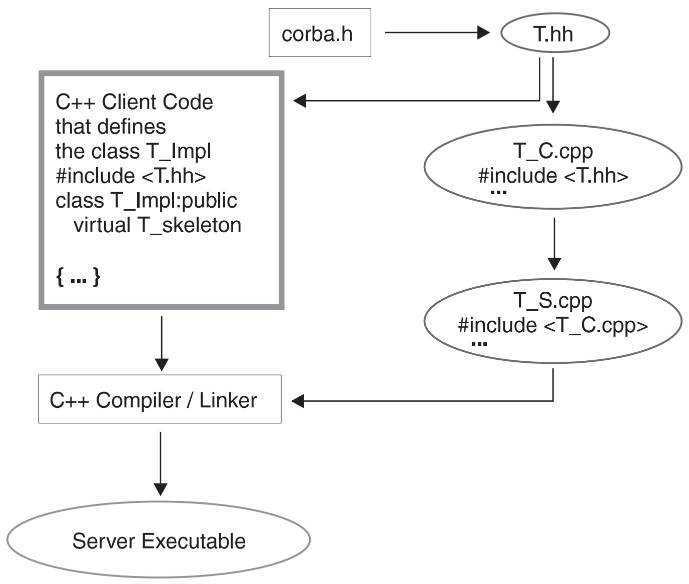


Figure 67. Module Structure of Server-Side Bindings

Take notice of the fact that the class name T_Impl is entirely arbitrary. The implementation class may have any name. Also note that the implementation class is not nested within any of the C++ classes that might be used to provide nesting scopes corresponding to IDL modules within which the interface T is defined. Thus, naming conflicts are a concern. A simple solution is to use underscores to concatenate module names with the name of the implemented interface. For example, if the interface T is defined within module M, then the implementation class name M_T_Impl can be used.

If the programmer responsible for T_Impl desires, the implementations (and supporting instance data) for any or all of T's parents can be inherited from their implementation classes, using C++ inheritance. Alternatively, T_Impl can provide its own implementation for the operations inherited into T. The image below graphically illustrates these options from an IDL snippet, using a dotted inheritance line to show optional C++ inheritance.

```
interface A
{
   ...
};
interface B : A
{
   ...
};
```

Figure 68 on page 330 focuses on C++ classes. The term *pure* is based on the use of this word in C++ to describe virtual functions that have no implementation (denoted in C++ by assigning a 0 to the name of the virtual function). Classes with pure virtual functions cannot be instantiated. Therefore, the skeleton classes require a subclass to provide complete implementations for all virtual functions in an interface. The dotted line in the previous figure indicates one way that B_Impl can provide implementations for the pure virtual functions inherited from A_Skeleton (using B_Skeleton). One way of viewing the skeleton classes is that they “turn off” proxy behavior and require subclasses to explicitly provide an alternative (non-proxy) implementation.

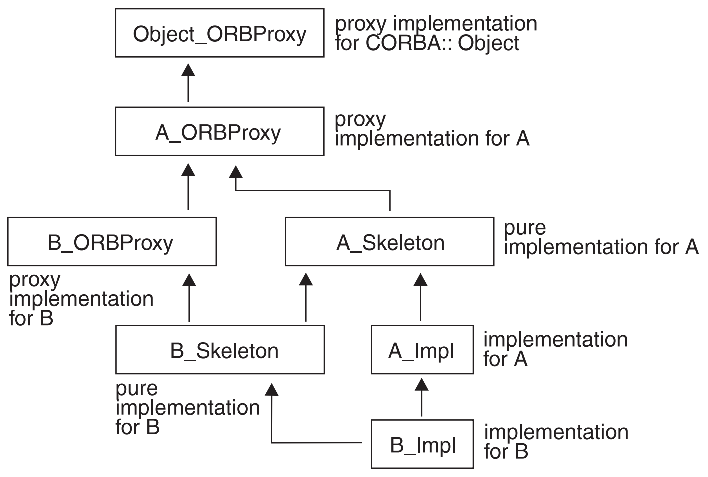

Object_ORBProxy
proxy implementation for CORBA:: Object
A_ORBProxy
proxy implementation for A
B_ORBProxy
proxy implementation for B
A_Skeleton
pure implementation for A
B_Skeleton
pure implementation for B
A_Impl
implementation for A
B_Impl
implementation for B

Figure 68. Inherited Implementation

C++ Binding Restrictions

When a forward reference to an interface appears within an IDL module, the IDL compiler issues an error message if the referenced interface is not defined within the module. When a similar unresolved forward reference appears at global (file) scope, a warning is issued that indicates the bindings being emitted won't include a mapping for the undefined interface. For information on the scope see “IDL Name Scoping” on page 271. The assumption is that the interface will be defined by other bindings than those being currently generated. This approach supports IDL files with mutually-referential interfaces (as long as they appear at global scope). The following example that illustrates how to organize the IDL files for such cases:

```
// file foo.idl
#ifndef foo_idl
#define foo_idl
interface Foo; // declare Foo so bar.idl can refer to it
#include <bar.idl>
interface Foo
{
   Bar foo1(); // notice the use of Bar
};
#endif // foo_idl
// file bar.idl
#ifndef bar_idl
#define bar_idl
interface Bar; // declare Bar so foo.idl can refer to it
#include <foo.idl>
interface Bar
{
   Foo bar1(); // notice the use of Foo
};
#endif // bar_idl
```

Due to problems inherent to the CORBA 2.0 mapping for C++, there are currently two known limitations with respect to handling legal CORBA 2.0 IDL. The compiler provides informative error messages in these two cases, and indicates that C++ bindings cannot be generated. The cases are:

- The C++ bindings map most IDL data types to C++ classes contained within a nesting scope provided by another C++ class. However, it is not legal to define a nested C++ class (or any other type) that has the same name as a containing C++ class. Thus, for example, the following IDL cannot be mapped to useful C++ bindings:

```
module X
{
   interface X ...;
   // or struct X ... ;
   // or union X ...;
   // or typedef sequence < > X;
   // ...
};
```

- The C++ bindings map attributes into overloaded C++ accessor functions whose name is the attribute name. As a result, for example, the following IDL will not map to useful C++ bindings (because Y's l method interferes with the inherited mapping for X's attribute). If Y's method took any arguments, there would not be a problem, because of C++ overloading. The compiler indicates an error only when C++ overloading won't distinguish inherited accessors from newly introduced methods (or vice versa).

```
interface X
{
   attribute long l;
}
interface Y : X
{
   long l();
};
```

Appendix D. Unit Test Environment

Note: The Unit Test Environment is only available on AIX and Windows NT platforms and does NOT apply to OS/390 Component Broker. Since the standard Component Broker development process has matured over the past releases, the usefulness of the Unit Test Environment has diminished. Thus, no further enhancements are planned for the Unit Test Environment, and the support of this Environment will be removed in a future release of Component Broker.

The purpose of the unit test process is to test business objects (BOs) without having to install them in a Component Broker server. This lets you, the object provider test the following Component Broker components:

- Business logic
- CosExternalization::Streamable implementation
- Managed object subclass implementation; for example, IManagedObjectWithCachedDataObject
- Data object interface
- Primary key
- Copy helper

The Unit Test environment is limited to a simple scenarios and does NOT support the full Component Broker development environment, including the following but is not limited to:

Persistent DOs

Transactions

PAA

Query

Security

Full Reference Collections interfaces

Full Iteration interfaces

Naming interfaces

If the Unit Test environment does not fill your needs, then it is better to follow the standard development process without using it.

The unit test process described here is not for testing the installation of a business object in a Component Broker server. There is no need to build a managed object (MO) and integrate it with a mixin. Building a managed object implies writing code that implements some interfaces, delegates to a mixin for others, and delegates to a business object for others. There is no need to customize the data object (DO) to a particular backing store.

Environment

The goal of the unit test environment (UTE) is to make unit testing of business objects as quick and easy as possible. As such, the unit test environment is designed to be extremely light-weight. The only requirement for unit testing of business objects is a run-time version of the unit test environment. The run-time version of the unit test environment is a single DLL which is dynamically linked to a unit test program. The unit test process does not require the presence of a Component Broker server and can even be performed on a Component Broker client.

Figure 69 shows all of the elements involved in unit testing a business object. The business object is just one part of the overall unit test equation. It makes little sense to unit test a business object without also unit testing the corresponding helper classes: Primary Key, and Copy Helper. The business object, Primary Key, and Copy Helper are all considered to be implemented by the object provider as input to the unit test process.

In addition to the previous implementations which are input to the unit test process, the process requires the implementation of a unit test program and a data object. A data object must be implemented at this stage because of the programming model adherence to the principle of separation of concerns.

The separation of an object into a business object and a data object allows a programmer with business application domain expertise to develop the business logic in the business object without having to become an expert in the persistent storage mechanisms that are used in the data object implementation. The business object is thus dependent on the data object interface, but not its implementation. Of course, the data object interface must eventually be implemented for the business object to be of any use. “Implementing the Data Object” on page 337 describes how to implement a transient data object for use in the unit test environment.

Finally, there is also some run-time support required by the unit test program and the various types of objects being tested.

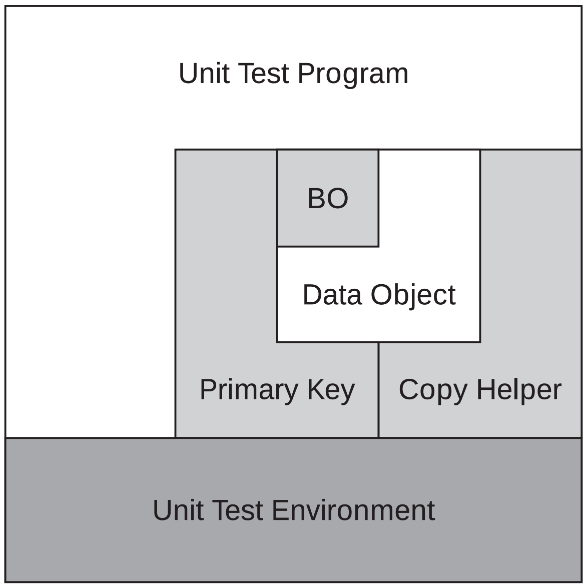


Figure 69. Unit Test Environment

Interfaces

The unit test environment provides support for many of the interfaces that a business object client expects to use when a business object has been installed on a Component Broker server. Expected interfaces include:

- IManagedClient::IHome
- IManagedClient::IManageable
- IManagedLocal::INonManageable

IManagedClient::IHome

This interface gives clients a way to create business objects and locate previously-created business objects. This interface includes the following methods:

- createFromPrimaryKeyString()
- findByPrimaryKeyString()
- createFromCopyString()
- getPrimaryKeyClass()
- getManagedObjectClass()

These methods are described in Chapter 5, “MOFW Server Programming Model” on page 57. These methods are supported in the unit test environment. The IManagedClient::IHome interface also supports the CosLifeCycle::GenericFactory interface, which introduces two additional methods that the unit test environment supports:

- supports()
- create_object()

Note: Create_object() is not supported by Component Broker for OS/390.

IManagedClient::IManageable

This interface introduces some common function which every managed object is expected to be able to do. For example, it is expected that every managed object is capable of telling its clients the Home in which it can be found. The interface to do this includes the following methods:

- getPrimaryKeyString()
- getHome()
- getHandleString()

The unit test environment provides support for the getHandleString() method which wrappers a stringified object reference. The unit test environment does not provide support for the getHome() method; instead an exception is thrown if this method is invoked in the unit test environment. The getPrimaryKeyString() must be implemented by the business object provider; neither the unit test environment nor the managed object Framework can provide a default implementation of this method.

The IManagedClient::IManageable interface also inherits three other interfaces:

CosStream::Streamable

This interface includes the following methods:

- externalize_to_stream()
- internalize_from_stream()

Both methods must be implemented by the business object provider.

CosLifeCycle::LifeCycleObject

This interface includes the following methods:

- copy()
- move()
- remove()

The unit test environment does not support these methods; an exception is thrown if these methods are invoked in the unit test environment.

CosObjectIdentity::IdentifiableObject

This interface includes the following methods:

- constant_random_id()
- is_identical()

The unit test environment supports these methods. The unit test environment implementation of constant_random_id() returns the address of the object; is_identical() performs a pointer comparison.

IManagedLocal::INonManageable

This interface is introduced because some objects (for example, Primary Keys and Copy Helpers) do not need all of the support that a managed object requires. They do not need to be accessed remotely, but they do need to be passed from a client to a server. Because the ORB does not support pass-by-value of objects, the IManagedLocal::INonManageable interface allows these objects to be converted to a string format which the ORB is capable of passing by value. The methods that are introduced for this purpose are toString() and fromString().

For an example of how these methods are used, refer to “Implementing the Unit Test Program” on page 344.

Unit Test Process

The unit test process starts where the business object development process (described in Chapter 5, “MOFW Server Programming Model” on page 57) ends. The following list is the input to the unit test process:

IDL files

- Interface for clients (Policy.idl)
- Interface for implementation (PolicyBO.idl)
- Interface for essential state (PolicyDO.idl)
- Interface for primary key class (PolicyKey.idl)
- Interface for copy helper class (PolicyCopy.idl)

Usage header files (generated by IDL compiler)

- Policy.hh
- PolicyBO.hh
- PolicyDO.hh
- PolicyKey.hh
- PolicyCopy.hh

Client and server binding files (generated by IDL compiler)

- Policy_C.cpp

- Policy_S.cpp
- PolicyBO_C.cpp
- PolicyBO_S.cpp
- PolicyDO_C.cpp
- PolicyKey_C.cpp
- PolicyCopy_C.cpp

Implementation files

- PolicyBO.ih
- PolicyBO_I.cpp
- PolicyKey.ih
- PolicyKey_I.cpp
- PolicyCopy.ih
- PolicyCopy_I.cpp

Note: The server binding files Policy_S.cpp and PolicyBO_S.cpp are not really needed for unit test because the unit test environment makes no client or server distinction.

Given the previous input, the only remaining pieces to be implemented are the data object and the unit test program itself.

Implementing the Data Object

The data object (PolicyDO) that is part of the input to the unit test process is just an interface, but before the data object can be used during unit test, it must have an implementation. For the unit test environment, the data object implementation is transient; that is, the essential state of the business object is not saved across executions of the unit test program. The following are the steps for implementing a data object for the unit test environment:

1. Create an IDL file that introduces an interface that inherits from the input data object interface (PolicyDO), and IManagedServer::IDataObject. IManagedServer::IDataObject introduces two methods that need to be implemented in order for the data object to work in the unit test environment.
2. Using the IDL compiler, generate a usage header file, client bindings, and implementation files from the IDL file. Server bindings are not necessary for data objects because data objects are local-only on the server.
3. Implement the two methods introduced by IManagedServer::IDataObject, as well as those introduced by the input data object (PolicyDO).

The following sections use the Policy example to illustrate these steps.

Creating IDL for the Unit Test Data Object

One of the inputs to the unit test process is the interface of a data object. This data object encapsulates all of the essential state of its corresponding business object. For the Policy object in the example, the data object interface provided by the object provider looks like the following:

```
interface PolicyDO
{
   attribute long policyNo;
   attribute float amount;
   attribute string comment;
};
```

For a data object to work in the unit test environment, it must implement the IManagedServer::IDataObject interface as well as the one required by its corresponding business object. The following IDL file introduces an interface (PolicyUnitTestDO) that combines both of these interfaces:

```
#include "PolicyDO.idl"
#include <IManagedServer.idl>

interface PolicyUnitTestDO : PolicyDO, IManagedServer ::IDataObject
{
   #pragma meta <Name>UnitTestDO localonly, abstract
};
```

Compiling IDL for the Unit Test Data Object

Compiling the IDL produces the following C++ implementation header file:

```
#ifndef _PolicyUnitTestDO_ih_included
#define _PolicyUnitTestDO_ih_included

// Generated from PolicyUnitTestDO.idl
// on Sun Feb 23 18:54:41 CST 1997
// by IBM CORBA 2.0 C++ (ih) header emitter version 2.00

#include <PolicyDO.ih>
#include <IManagedServer.ih>
#include "PolicyUnitTestDO.hh"

class PolicyUnitTestDO_Impl :
   public virtual ::PolicyUnitTestDO_Skeleton,
   public virtual PolicyDO_Impl,
   public virtual IManagedServer_IDataObject_Impl
{

   public:
};

#endif /* _PolicyUnitTestDO_ih_included */
```

However, the emitted implementation header file is not correct for the example because of the following assumptions made by the idlc emitter:

- Every interface is implemented where it is introduced.
- All inheritance is implementation inheritance.

In the example, both PolicyDO and IManagedServer::IDataObject are interfaces without implementations. As such, there is no implementation from which PolicyUnitTestDO can inherit. Fix the emitter's incorrect assumptions by removing the parent classes and their corresponding implementation header files:

```
...
// Remove these implementation header files
#include <PolicyDO.ih>
#include <IManagedServer.ih>
...
   // Remove these parent classes
   public virtual PolicyDO_Impl,
   public virtual IManagedServer_IDataObject_Impl
...
```

Compiling the IDL also resulted in the following C++ implementation file:

```
// Generated from PolicyUnitTestDO.idl
// on Sun Feb 23 18:54:41 CST 1997
// by IBM CORBA 2.0 C++ (ic) header emitter version 2.00

#include "PolicyUnitTestDO.ih"
```

This is an empty file because the PolicyUnitTestDO interface does not introduce any methods, it merges together two other interfaces and the emitter assumes that all interfaces are implemented where they are introduced. Because PolicyDO and IManagedServer::IDataObject do not have implementations, it is necessary to implement these interfaces in PolicyUnitTestDO.

Implementing the Unit Test Data Object

The unit test data object interface described previously merges together two interfaces:

- An interface that represents the essential state of a business object (PolicyDO in the example)
- IManagedServer::IDataObject

The step of implementing the unit test data object can be summarized as implementing the two interfaces which it merges together. The first interface (PolicyDO) is a set of attributes, one for each data element which makes up a business object's essential state. These IDL attributes result in get and set methods that must be implemented. The PolicyDO.hh file declares the following methods:

```
virtual ::CORBA::Long policyNo()=0;
virtual ::CORBA::Void policyNo(::CORBA::Long policyNo)=0;

virtual ::CORBA::Float amount()=0;
virtual ::CORBA::Void  amount(::CORBA::Float amount)=0;

virtual char* comment()=0;
virtual ::CORBA::Void  comment(char* comment)=0;
```

Because these methods must be implemented in the unit test data object, they must be added to the PolicyUnitTestDO.ih file. The second interface that must be implemented, IManagedServer::IDataObject, introduces two methods: internalizeFromPrimaryKey and internalizeFromCopyHelper.

The IManagedServer.hh files declare them as follows:

```
virtual ::CORBA::Void internalizeFromPrimaryKey
              (::IManagedLocal::IPrimaryKey_ptr inKey) = 0;

virtual ::CORBA::Void internalizeFromCopyHelper
              (::IManagedLocal::INonManageable_ptr inCopy) = 0;
```

These methods must also be added to the PolicyUnitTestDO.ih file, resulting in the following:

```
#ifndef _PolicyUnitTestDO_ih_included
#define _PolicyUnitTestDO_ih_included

// Generated from PolicyUnitTestDO.idl
// on Sun Feb 23 18:54:41 CST 1997
// by IBM CORBA 2.0 C++ (ih) header emitter version 2.00

#include "PolicyUnitTestDO.hh"

 class PolicyUnitTestDO_Impl :
              public virtual ::PolicyUnitTestDO_Skeleton
{
   public:

   // Implement PolicyDO interface

   virtual ::CORBA::Long policyNo();
   virtual ::CORBA::Void policyNo(::CORBA::Long policyNo);

   virtual ::CORBA::Float amount();
   virtual ::CORBA::Void  amount(::CORBA::Float amount);

   virtual char* comment();
   virtual ::CORBA::Void  comment(char* comment);

   // Implement IManagedServer::IDataObject interface

   virtual ::CORBA::Void internalizeFromPrimaryKey
              (::IManagedLocal::IPrimaryKey_ptr inKey);

   virtual ::CORBA::Void internalizeFromCopyHelper
              (::IManagedLocal::INonManageable_ptr inCopy);

};

#endif /* _PolicyUnitTestDO_ih_included */
```

The PolicyUnitTestDO.ih file is starting to take shape, but is not complete. The policy business object (PolicyBO) counts on the data object to manage its essential state. Although the data object is not persistent in the unit test environment, it requires a transient location to store the business object's essential state. For this purpose, for each IDL attribute in PolicyDO.idl, one C++ data member is introduced to the PolicyUnitTestDO.ih resulting in the following implementation header file which is ready to be implemented:

```
#ifndef _PolicyUnitTestDO_ih_included
#define _PolicyUnitTestDO_ih_included

// Generated from PolicyUnitTestDO.idl
// on Sun Feb 23 18:54:41 CST 1997
// by IBM CORBA 2.0 C++ (ih) header emitter version 2.00

#include "PolicyUnitTestDO.hh"

class PolicyUnitTestDO_Impl :
          public virtual ::PolicyUnitTestDO_Skeleton
{

   public:

   // Implement PolicyDO interface

   virtual ::CORBA::Long policyNo();
   virtual ::CORBA::Void policyNo(::CORBA::Long policyNo);

   virtual ::CORBA::Float amount();
   virtual ::CORBA::Void  amount(::CORBA::Float amount);

   virtual char* comment();
   virtual ::CORBA::Void comment(char* comment);
   // Implement IManagedServer::IDataObject interface

   virtual ::CORBA::Void internalizeFromPrimaryKey
                (::IManagedLocal::IPrimaryKey_ptr inKey);

   virtual ::CORBA::Void internalizeFromCopyHelper
                (::IManagedLocal::INonManageable_ptr inCopy);

   private:
      // Store the values of the above public attributes transiently.
      ::CORBA::Long  tPolicyNo;    // Store value of "policyNo" attribute
      ::CORBA::Float tAmount;      // Store value of "amount" attribute
      ::CORBA::String_var tComment;// Store value of "comment" attribute

};

#endif /* _PolicyUnitTestDO_ih_included */
```

Implementing PolicyDO Methods

The policy business object (PolicyBO) uses the methods on the PolicyDO interface to access its essential state. The transient implementation of the PolicyDO interface in PolicyUnitTestDO can be described by the following rules:

- For each get method in the C++ implementation header file PolicyUnitTestDO.ih resulting from the IDL attributes in PolicyDO.idl, return the corresponding C++ data attribute.
- For each set method in the C++ implementation header file PolicyUnitTestDO.ih resulting from the IDL attributes in PolicyDO.idl, assign the value of the parameter to the corresponding C++ data attribute.

Following is the code that these two rules produce for the first of the IDL attributes in PolicyDO.idl (amount). For the get method:

```
::CORBA::Float PolicyUnitTestDO_Impl::amount()
{
   return tAmount;
}
```

For the set method:

```
::CORBA::Void PolicyUnitTestDO_Impl::amount(::CORBA::Float amount)
{
   tAmount = amount;
}
```

The *policyNo* attribute follows the same pattern. The *comment* attribute is a little different, because its IDL data type is string. The implementation of the *comment* attribute follows:

```
char* PolicyUnitTestDO_Impl::comment()
{
   return CORBA::string_dup(tComment); // return a copy
}

::CORBA::Void PolicyUnitTestDO_Impl::comment(char* comment)
{
   tComment = comment;  // ::CORBA::String_var will copy parameter
}
```

Implementing IManagedServer::IDataObject Methods

The unit test implementation of IHome requires that the internalizeFromPrimaryKey() and internalizeFromCopyHelper() methods of the IManagedServer::IDataObject interface be implemented. To understand why these methods are required, look at the Primary Key case.

The Home receives a Primary Key helper object (a stringified version of a Primary Key helper object) on a createFromPrimaryKeyString() request. The Home first creates (or recreates) the original Primary Key helper object from its stringified version. Then, it creates a data object and passes the Primary Key helper object to the data object on an internalizeFromPrimaryKey() call. When the data object is created, it has no state associated with it. It is this internalizeFromPrimaryKey() method that associates an initial state with the data object.

The implementation of internalizeFromPrimaryKey() must do the following: for each attribute that makes up the Primary Key, get the value from the Primary Key and set the corresponding value in the data object. An example of the implementation of internalizeFromPrimaryKey() method for the unit test policy data object follows:

```
::CORBA::Void PolicyUnitTestDO_Impl::internalizeFromPrimaryKey
                    (::IManagedLocal::IPrimaryKey_ptr inKey)
{
   // Convert the input parameter Primary Key reference
   //   to a policy Primary Key reference

   PolicyKey_var pk = PolicyKey::_narrow(inKey);

   // Store the key attributes in the DO data attributes.
   // Get value from the Primary Key and set it in the DO private state
   //   variable.  Repeat as needed for additional Primary Key attributes

   tPolicyNo = pk->policyNo();  // Store value of "policyNo" attribute
}
```

A Copy Helper can be thought of as a super-set of a Primary Key. As such, a Copy Helper would have the same attributes as a Primary Key plus some additional ones (possibly up to and including every attribute on a data object). Given that, the implementation of internalizeFromCopyHelper() would look similar to that of internalizeFromPrimaryKey(). Here is an example of the unit test policy data object:

```
::CORBA::Void PolicyUnitTestDO_Impl::internalizeFromCopyHelper
          (::IManagedLocal::INonManageable_ptr inCopy)
{
   PolicyCopy_var pc = PolicyCopy::_narrow(inCopy);

   // Store the copy attributes in the DO data attributes

   tPolicyNo = pc->policyNo();  // Store value of "policyNo" attribute
   tAmount   = pc->amount();   // Store value of "amount" attribute
   tComment  = pc->comment();   // Store value of "comment" attribute
}
```

Having implemented both the PolicyDO and IManagedServer::IDataObject interfaces, the PolicyUnitTestDO_I.cpp file looks like this:

```
// Generated from PolicyUnitTestDO.idl
// on Sun Feb 23 18:54:41 CST 1997
// by IBM CORBA 2.0 C++ (ic) header emitter version 2.00

#include "PolicyUnitTestDO.ih"
```

```
/*
 * Method from the IDL attribute statement:
 * "attribute Long policyNo"
 */
::CORBA::Long PolicyUnitTestDO_Impl::policyNo()
{
   return tPolicyNo;
}

/*
 * Method from the IDL attribute statement:
 * "attribute Long policyNo"
 */
::CORBA::Void PolicyUnitTestDO_Impl::policyNo(::CORBA::Long policyNo)
{
   tPolicyNo = policyNo;
}

/*
 * Method from the IDL attribute statement:
 * "attribute Float amount"
 */
::CORBA::Float PolicyUnitTestDO_Impl::amount()
{
   return tAmount;
}

/*
 * Method from the IDL attribute statement:
 * "attribute Float amount"
 */
::CORBA::Void PolicyUnitTestDO_Impl::amount(::CORBA::Float amount)
{
   tAmount = amount;
}

/*
 * Method from the IDL attribute statement:
 * "attribute string comment"
 */
char* PolicyUnitTestDO_Impl::comment()
{
   return CORBA::string_dup(tComment); // return a copy
}

/*
 * Method from the IDL attribute statement:
 * "attribute string comment"
 */
::CORBA::Void PolicyUnitTestDO_Impl::comment(char* comment)
{
   tComment = comment;  // ::CORBA::String_var will copy parameter
}

::CORBA::Void PolicyUnitTestDO_Impl::internalizeFromPrimaryKey
        (::IManagedLocal::IPrimaryKey_ptr inKey)
{
   PolicyKey_var pk = PolicyKey::_narrow(inKey);

   // Store the key attributes in the DO data attributes

   tPolicyNo = pk->policyNo();  // Store value of "policyNo" attribute
}

::CORBA::Void PolicyUnitTestDO_Impl::internalizeFromCopyHelper
        (::IManagedLocal::INonManageable_ptr inCopy)
{
   PolicyCopy_var pc = PolicyCopy::_narrow(inCopy);

   // Store the copy attributes in the DO data attributes

   tPolicyNo = pc->policyNo();  // Store value of "policyNo" attribute
   tAmount   = pc->amount();    // Store value of "amount" attribute
   tComment  = pc->comment();   // Store value of "comment" attribute
}
```

Implementing the Unit Test Program

Now that there is a fully implemented transient data object for the business object, you are ready to write a unit test program. As discussed previously, the purpose of this unit test process is to test business objects. To do this, the unit test environment enables a subset of the client programming model. This subset is described in “Environment” on page 334; the client programming model is described in Chapter 4, “MOFW Client Programming Model” on page 33.

However, the unit test environment has requirements that must be filled by a unit test program:

1. For each business object that a unit test program tests, the unit test program must define and create a factory object that takes no parameters for each of the following types of objects:

   - Business object
   - Data object
   - Primary Key
   - Copy Helper

   These factory objects all have a single create() method that takes no parameters, and returns the appropriate object from the application domain. The factory objects return the objects as the more general type from which they inherit. For instance, the factory object for a policy business object would look like this:

   ```
   class PolicyBOFactory: public
         IUnitTestClient:IBusinessObjectFactory_Skeleton
   {
      ::IManagedClient::IManageable_ptr create()
      {
         return (IManagedClient::IManageable*) new PolicyBO_Impl();
      };
   };
   ```

2. For each business object class that a unit test program tests, the unit test program is responsible for creating and initializing a Home that can be used to create and locate business objects of that class. Before a Home can be used in a unit test program, it must be initialized with references to the factory objects described previously, plus the class name of the business objects it creates (and finds). The following code segment shows how this would be done:

```
    IUnitTestClient::IHome_var myHome =
         IUnitTestClient::IHome::_create();

    PolicyBOFactory*   f1 = new PolicyBOFactory();
    PolicyDOFactory*   f2 = new PolicyDOFactory();
    PolicyKeyFactory*  f3 = new PolicyKeyFactory();
    PolicyCopyFactory* f3 = new PolicyCopyFactory();
    myHome->initForTesting("Policy", f1, f2, f3, f4);
```

Having fulfilled the requirements, a unit test program would proceed to use the Home to create and locate business objects, exercising all of the functions provided by the business object. Following is an example of a simple unit test program that tests the policy business object:

```
#include <iostream.h>

#include <IUnitTestClient.ih>

#include "PolicyBO.ih"
#include "PolicyDO.ih"
#include "PolicyKey.hh"
#include "PolicyCopy.hh"

class PolicyBOFactory : public
       IUnitTestClient::IBusinessObjectFactory_Skeleton
{
   ::IManagedClient::IManageable_ptr create()
   {
      return (IManagedClient::IManageable*) new PolicyBO_Impl();
   };
};
class PolicyDOFactory : public
       IUnitTestClient::IDataObjectFactory_Skeleton
{
   ::IManagedServer::IDataObject_ptr create()
   {
      return (IManagedServer::IDataObject*) new
              PolicyUnitTestDO_Impl();
   };
};

class PolicyKeyFactory : public
        IUnitTestClient::IPrimaryKeyFactory_Skeleton
{
   ::IManagedLocal::IPrimaryKey_ptr create()
   {
      return (IManagedLocal::IPrimaryKey*)PolicyKey::_create();
   };
};

class PolicyCopyFactory : public
         IUnitTestClient::ICopyHelperFactory_Skeleton
{
   ::IManagedLocal::INonManageable_ptr create()
   {
      return (IManagedLocal::INonManageable*)PolicyCopy::_create();
   };
};
```

```
main()
{
   IUnitTestClient::IHome_var myHome =
        IUnitTestClient::IHome::_create();

   PolicyBOFactory*   f1 = new PolicyBOFactory();
   PolicyDOFactory*   f2 = new PolicyDOFactory();
   PolicyKeyFactory*  f3 = new PolicyKeyFactory();
   PolicyCopyFactory* f3 = new PolicyCopyFactory();
   myHome->initForTesting("Policy", f1, f2, f3, f4);

   cout << "Creating a Policy primary key object...";
   PolicyKey_var pk = PolicyKey::_create();
   pk->policyNo(12345);

   cout << "Creating Policy BO using home and primary key...";
   IManagedClient::IManageable_ptr temp;
   ByteString * tempBS = pk->toString();
   temp = myHome->createFromPrimaryKeyString( *tempBS );
   PolicyBO_var myPolicy = PolicyBO::_narrow(temp);
   CORBA::release( temp );
   delete tempBS;

   cout << "************ Object Created Successfully ************";

   ::CORBA::Long pNo = myPolicy->policyNo();
   cout << "Policy number - should be 12345.  It is " << pNo << endl;

}
```

Compiling, Linking, and Executing the Unit Test Program

Having implemented both a unit test data object and a unit test program, you must compile and link all of the pieces and execute the resulting .EXE file. The files that need to be compiled are:

Input to Unit Test Process

- Policy_C.cpp
- PolicyBO_C.cpp
- PolicyBO_I.cpp
- PolicyDO_C.cpp
- PolicyKey_C.cpp
- PolicyKey_I.cpp
- PolicyCopy_C.cpp
- PolicyCopy_I.cpp

Output of Unit Test Process

- PolicyUnitTestDO_C.cpp
- PolicyUnitTestDO_I.cpp
- UnitTest.cpp

The objects that need to be linked to build the executable unit test program are:

Input to Unit Test Process

- Policy_C.obj
- PolicyBO_C.obj
- PolicyBO_I.obj
- PolicyDO_C.obj
- PolicyKey_C.obj
- PolicyKey_I.obj
- PolicyCopy_C.obj
- PolicyCopy_I.obj

Output of Unit Test Process

- PolicyUnitTestDO_C.obj
- PolicyUnitTestDO_I.obj
- UnitTest.obj

Unit Test Environment

- sompmuli.lib

Note: These lists represent only one possible packaging scheme. Other packaging schemes include:

1. All of the .cpp files that were input to the unit test process are compiled and linked into a single LIB/DLL which is then linked in like the unit test environment LIB/DLL.
2. All of the client usage binding (_C.cpp) files that were input to the unit test process are compiled and linked into one LIB/DLL, and all of the implementation (_I.cpp) files that were input to the unit test process are compiled and linked into a second LIB/DLL. Both of these LIB/DLLs are then linked in like the unit test environment LIB/DLL.
3. PolicyUnitTestDO_C.cpp and PolicyUnitTestDO_I.cpp are compiled and linked together into a LIB/DLL which is then linked in like the unit test environment LIB/DLL.

Regardless of the packaging decision, at this point, you are ready to execute the unit test program.

Unit Test for Java Business Objects

After successfully writing and compiling a Java business object and some related classes, unit testing is the next step. Unit testing exercises the business logic in a scaffolded, pure Java environment outside of the Component Broker server, and can accomodate any Java development toolset.

Files

The previous set of development steps resulted in a number of IDL files and some Java files that were written (probably using the Object Builder). Since the unit test environment (UTE) is a pure Java environment, you need to run `java com.ibm.idl.toJava.Compile`to generate the bindings for the following IDL files:

IDL Files

- Policy.idl
- PolicyBO.idl
- PolicyDO.idl
- PolicyKey.idl
- PolicyCopy.idl

Written Java Packages

- XYZCompany/_PolicyBOBase.java
- XYZCompany/_PolicyKeyImpl.java
- XYZCompany/_PolicyCopyImpl.java

Generated Java needed for Unit Test

- XYZCompany/Policy.java (from Policy.idl)
- XYZCompany/PolicyHelper.java
- XYZCompany/PolicyBO.java (from PolicyBO.idl)
- XYZCompany/PolicyBOHelper.java
- XYZCompany/PolicyDO.java (from PolicyDO.idl)
- XYZCompany/PolicyDOHelper.java
- XYZCompany/PolicyKey.java (from PolicyKey.idl)
- XYZCompany/PolicyKeyHelper.java
- XYZCompany/PolicyCopy.java (from PolicyCopy.idl)
- XYZCompany/PolicyCopyHelper.java

Generated Java saved for later

- XYZCompany/_PolicyStub.java
- XYZCompany/_PolicyBOStub.java
- XYZCompany/_PolicyDOStub.java

Component Broker supplies the Java scaffolding that simulates the server execution environment. In addition to the implementations required by the unit test process, the object provider must supply the unit test data object and the unit test program iteslf.

Implementing a Java Unit Test Data Object

Unit testing performs locally (outside the Component Broker server), and does not require a persistent data store. The unit test data object, therefore, is a transient object; that is, the essential state of the business object is not saved across executions of the unit test program. The unit test data object for the Policy business object must implement the PolicyDO interface as well as the IManagedServer::IDataObject interface.

```
interface PolicyDO

{
   attribute long policyNo;
   attribute float amount;
   attribute float premium;
   attribute string comment;
   attribute com.ibm.IManagedCollections.IReferenceCollection beneficiaries;
};
```

You can combine this data object interface with the IManagedServer::IDataObject interface to construct the implementation of the transient data object as follows:

```
package XYZCompany;

public class _PolicyTransientDOImpl
   extends com.ibm.IManagedClient._IManageableBase
   implements PolicyDO, com.ibm.IManagedServer.IDataObject
{
   private int policyNo;
   private float amount;
   private String comment;
```

```
    private float premium;

    public int policyNo()           { return policyNo; }
    public void policyNo(int no)    { policyNo = no; }

    public float amount()           { return amount; }
    public void amount(float amt)   { amount = amt; }

    public float premium()          { return premium; }
    public void premium(float amt)  { premium = amt; }

    public String comment()         { return comment; }
    public void comment(String com) { comment = com; }

    public void internalizeFromPrimaryKey(com.ibm.IManagedLocal.IPrimaryKey key)
    {
       PolicyKey pk = (PolicyKey)key;                                // See note 1
       policyNo = pk.policyNo();
    }

    public void internalizeFromCopyHelper(com.ibm.INonManageable copy)   // See note 2
    {
       PolicyCopy pc = (PolicyCopy)copy;
       policyNo = pc.policyNo();
       amount = pc.getAmount();
       comment = pc.comment();
    }

    public byte[] getPrimaryKeyString () { return null; }
    public void externalize_to_stream (org.omg.CosStream.StreamIO s) {}
    public void internalize_from_stream (org.omg.CosStream.StreamIO f1,
                             org.omg.CosLifeCycle.FactoryFinder f2 {}
 }
```

This is a simple data object that expects to receive all of its state data from a unit test program using a PrimaryKey or CopyHelper object or by explicit calls to set the data object attributes. If the test scenarios you plan to use require considerable amounts of data, you may want to consider other methods of initializing unit test data objects. For example, when a data object key has been set, the data object could then scan a text file that has, on each line, a key value followed by initial values for the other data object attributes. You can provide test data in any of several ways; you can choose one with which you are comfortable.

Notes about the data object:

1. Remember that you must have implemented a pure Java version of the PrimaryKey class in order to use the Java unit test process.
2. Copy Helper classes are optional; if you do not have one you can leave the body of this method empty.

Implementing the Unit Test Factories

The Java unit test environment requires that you provide factory objects to create instances of the business object, unit test data object, Primary Key, and optional Copy Helper classes. Each factory has a single create() method that takes no parameters and implements an interface in the IUnitTestClient package.

Following are examples of factory classes suitable for the Policy business object class:

```
public class PolicyBOFactory
      implements com.ibm.IUnitTestClient.IBusinessObjectFactory
{
   public com.ibm.IManagedClient.IManageable create()
   {
      return new _PolicyBOBase();
   }
}

public class PolicyDOFactory()
   implements com.ibm.IUnitTestClient.IDataObjectFactory
{
   public com.ibm.IManagedServer.IDataObject create()
   {
      return new _PolicyTransientDOImpl();
   }
}

public class PolicyKeyFactory()
   implements com.ibm.IUnitTestClient.IPrimaryKeyFactory
{
   public com.ibm.IManagedLocal.IPrimaryKey create()
   {
      return PolicyKeyHelper._create();
   }
}

public class PolicyCopyFactory()
   implements com.ibm.IUnitTestClient.ICopyHelperFactory
{
   public com.ibm.IManagedLocal.INonManageable create()
   {
      return PolicyCopyHelper._create();
   }
}
```

Because these classes are only used in the controlled unit test environment, they are not constrained to use only the *XXX*Helper._create() pattern to create instances of these classes. In cases where the name of the implementation class is known, it can be used directly in unit test, although that is not recommended for code that executes on the server.

These factory objects are used in the unit test program to configure the unit test Home, as shown in the next section.

Implementing the Unit Test Program

When the preparation is complete a unit test program can now be written to exercise the business logic of the Policy object. This test program is written as a Java application and, like any Java application, is a class containing a public static main() method.

The unit test program uses an instance of IUnitTestClient::IHome interface configured to act as a mockup Home for the business object. If the test scenario involves more than one type of business object, one unit test Home instance needs to be configured for each. Because IUnitTestClient::IHome extends the normal Home client programming interface IManagedClient::IHome, the test Home can be used within the business object. However, the test Home is not an IterableHome nor is it a QueryableHome.

Each unit test program performs the following steps:

1. Create Homes
2. Configure Homes with unit test factory objects
3. Create business objects
4. Initialize business objects, if necessary
5. Call business objects business methods and validate results

An example of a unit test program for the Policy implementation follows:

```
import XYZCompany.*;

public class PolicyUnitTest
{
   public static void main(String args[])
   {
      com.ibm.IUnitTestClient.IHome home =
            com.ibm.IUnitTestClient.IHomeHelper._create();

      home.initForTesting("Policy",
                  new PolicyBOFactory(),
                  new PolicyDOFactory(),
                  new PolicyKeyFactory(),
                  new PolicyCopyFactory());

      System.out.println("Creating Policy Key");
      PolicyKey pk = PolicyKeyHelper._create();
      pk.policyNo(12345);

      System.out.println("Creating Policy using key");
      Policy p = PolicyHelper.narrow(
            home.createFromPrimaryKeyString(pk._toString()));

      System.out.println("Policy object created");

      System.out.print("Checking key == 12345: ");
      if(p.policyNo() == 12345)
            System.out.println("Pass");
      else
            System.out.println("Fail");
   }
}
```

Running the Test Program

To run the unit test program, the Java client must be installed on the system. You can run the Java unit test program using the `java` command or from within your favorite Java development environment. However you run it, the proper Java classes must be available. In some cases Component Broker includes several Java implementations of the same class for use in different run time environments. It is important to ensure that the version that is appropriate for the unit test environment is used when you run your unit test program.

The order that the `java` command searches directories and ZIP archives for Java classes is established by the CLASSPATH environment variable. If you are running your unit test program in a Java development environment you should consult the documentation for that product to determine how to set the search path. The path should contain, in order:

1. The directory containing the class files for the unit test program, business object, unit test data object, and other related classes.
2. The path to the installed SOMOJUT.ZIP file (the unit test class archive) containing special unit test simulation classes such as the IUnitTestClient package.
3. The path to the installed SOMOJOR.ZIP file (the pure Java ORB and bindings archive) providing access to remote objects and services using the Java client ORB.
4. The path to the installed IBMCBJS.ZIP file (the Server Java bindings archive) providing Java interface definitions but no classes that are usable in the unit test.
5. The path to the install SOMOJIJ.ZIP file providing the idl-to-Java compiler.
6. Any other ZIP files or directories required by your application or your Java run time environment.

Unit Test Supported Function

The unit test environment is basically a simulation of a server environment, and as such, has limitations. Certain facilities and code patterns that can be used in server code are not supported by the unit test simulation. Therefore, business object code that attempts to use them could suffer failures at unit test execution time, or could fail to load correctly. If your business object uses these unsupported features, you may still be able to unit test other business methods or you may need to perform all of your debugging and testing activity in a true server environment.

Setting the CLASSPATH as described earlier provides access to the following facilities in the unit test environment, subject to the following limitations:

- Your business object, running in simulated mode
- Objects running in simulated mode that support the following interfaces:
  - IManagedClient::IHome
  - CosObjectIdentity::IdentifiableObject
  - CosLifeCycle::GenericFactory
- Objects and services running in client mode:
  - CBSeriesGlobal
  - Other object services
  - Remote business objects running in servers

Restrictions on the use of these objects are implied by their modes.

Simulated mode means that a reference to the object cannot be passed on a remote call and cannot be used with an Object Service unless that service is also simulated. The operations introduced by the CORBA::Object interface are not supported in the UTE.

Client mode means that the objects referenced must execute in a configured, separately running server. The objects are accessed remotely through the ORB using a proxy.

Summary

This chapter described the unit test process and environment. Unit test was defined as testing of business objects. More specifically, it is the testing of all the code that is described in Chapter 5, “MOFW Server Programming Model” on page 57 and in Chapter 10, “Java Server Programming Model” on page 171. As in those chapters, this chapter explains the steps involved if you were to do everything manually. Tools support should make both development and unit test of business objects much easier: the Object Builder tool should be able to generate a unit test data object, and it should be able to assist in the writing of a unit test program.

Regardless of whether unit test was completed manually, or with the help of tools, when a business object has been tested, it is ready to be installed into a Component Broker server. For more information about installing a business object, see Chapter 11, “Assembling and Installing Business Objects on AIX and Windows NT” on page 197.

Appendix E. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

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 LZKS
11400 Burnet Road
Austin, TX 78758
U.S.A.

Trademarks

OS/2
OS/390
PowerPC
VisualAge

AFS and DFS are trademarks of Transarc Corporation in the United States, or other countries, or both.

Java and HotJava are trademarks of Sun Microsystems, Inc.

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

Oracle and Oracle8 are registered trademarks of the Oracle Corporation.

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

Other company, product, and service names may be trademarks or service marks of others.

Index

Part Number: 04L2376

Printed in the United States of America

04L2376

G04L-2376-04