MICRO FOCUS

Orbix Mainframe 6.3.1

CORBA Concepts Guide

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
https://www.microfocus.com

2021-03-18

Contents

List of Figures

List of Tables

Preface

Orbix is a full implementation of the Common Object Request Broker Architecture (CORBA), as specified by the Object Management Group. Orbix complies with the following specifications:

- CORBA 2.6
- GIOP 1.2 (default), 1.1, and 1.0

Orbix Mainframe is an implementation of Orbix for the z/OS platform.

Orbix Mainframe documentation is periodically updated. The latest versions are available from:

https://www.microfocus.com/documentation/orbix/

Audience

This guide is intended for COBOL and PL/I application programmers with no object-oriented programming knowledge who want to develop Orbix Mainframe applications in a native z/OS environment.

A working knowledge of the COBOL or PL/I programming language is essential. Experience in Client/Server architecture and distributed systems is helpful, but not essential. Also, the chapter on IDL design assumes that the reader is familiar with the concepts of IDL, as presented in the *COBOL Programmer's Guide and Reference* and *PL/I Programmer's Guide and Reference*.

Organization of this guide

This guide is divided as follows:

Chapter 1, "Introduction to Object-Oriented Programming"

This chapter discusses the development of the client-server technology model from a two-tier architecture to a disturbed system architecture. It also introduces the fundamental concepts of object-oriented computing relevant to CORBA development.

Chapter 2, "Introduction to CORBA"

This chapter introduces CORBA which is a specification for a specific type of distributed system. It also introduces Interface Definition Language (IDL) and the Object Request Broker (ORB) which are the two foundation stones of CORBA.

Chapter 3, "Introduction to Orbix"

This chapter introduces Orbix, which is an implementation of the CORBA 2.4 specification. It provides an overview of the Orbix architecture relative to Orbix Mainframe, which comprises the Adaptive Runtime Technology (ART) framework and an open-ended set of plug-ins that provide the required functionality.

Chapter 4, "Introduction to Orbix Mainframe"

This chapter introduces Orbix Mainframe, which is an implementation of Orbix for the z/OS environment. It illustrates how Orbix Mainframe can be used to develop a COBOL or PL/I server which can be part of an Orbix distributed system.

Chapter 5, "IDL Design"

This chapter introduces considerations for designing IDL interfaces. It is assumed that for this chapter the reader is familiar with the concepts of IDL as presented in either the COBOL or PL/I programmer's guide and reference.

Chapter 6, "Locating CORBA Objects"

This chapter introduces the fundamental tenants of CORBA object location, including how clients in a CORBA distributed system obtain object references. It also introduces the CORBA Naming Service. Several approaches to publishing and locating objects are discussed, and their strengths and weaknesses are explored.

“Glossary”
This glossary contains definitions for object-oriented, CORBA, and Orbix terminology used in this document.

Additional resources

The Knowledge Base contains helpful articles, written by experts, about Orbix Mainframe, and other products:

https://community.microfocus.com/t5/Orbix/ct-p/Orbix

If you need help with Orbix Mainframe or any other products, contact technical support:

https://www.microfocus.com/en-us/support/

Typographical conventions

This guide uses the following typographical conventions:

`Constant width` Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

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

Italic Italic words in normal text represent *emphasis* and *new terms*.

Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:

`% cd /users/`*your_name*

Note: Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
`%`	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
`#`	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, Windows 95, or Windows 98 command prompt.

`...`
 `.`
 `.`
 `.`
	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Part 1

Object-Oriented Programming, CORBA, and Orbix Mainframe

In this part

This part contains the following chapters:

CHAPTER 1

Introduction to Object-Oriented Programming

This chapter provides a brief overview of how Object-Oriented programming and client-server computing evolved. It also describes the concepts of object-oriented programming, and illustrates these concepts with real world examples.

In this chapter

This chapter discusses the following topics:

Object-Oriented Programming

Overview

This section summarizes the fundamental concepts of object-oriented (OO) programming relative to non-OO programming. It also lists the languages that interoperate with Common Object Request Broker Architecture (CORBA).

This section discusses the following topics:

- Non-OO programming
- OO programming
- OO languages
- Languages that interoperate with CORBA

Non-OO programming

Non-OO programming languages, otherwise known as procedural languages and in existence long before OO programming languages, are still very much in use today. Applications are still developed using only procedural languages. However, the range of problems that programming languages have to solve in today's enterprise environment is much wider than was the case when procedural languages were initially being developed. Solving these problems has lead to the development of a fundamentally different approach to programming.

OO programming

There are two basic concepts behind the development of OO programming: interoperability and reusability. OO programming is a fundamentally different way of writing applications than procedural programming, because in OO programming, objects (refer to "What are objects?" on page 16 for a definition) are (usually) not designed from scratch each time they are required, but adapted from previously tried and tested objects. The real difference in the two approaches becomes apparent at the design stage of an application. An OO approach examines how objects, already developed and publicly available, can be adapted to meet the requirements of a specific application.

OO languages	The two most widely used OO languages for commercial applications are C++ and Java, which draw on many concepts that are closely related. There are many other OO languages, but those of interest to us are those that interoperate with CORBA.
Languages that interoperate with CORBA	Currently languages that interoperate with CORBA include Ada, C, COBOL, C++, LISP, Java, PL/I, Python, Smalltalk, and XML.

Client-Server Computing

Overview

This section provides a brief historical overview of client-server architecture and how it evolved into distributed systems architecture. It also outlines the relationship between distributed systems and OO programming.

In this section

This section discusses the following topics:

Two-Tier Client-Server Architecture

Overview

This section summarizes the development of two-tier client-server architecture, and provides a simple example of this model. It also summarizes the main problems of the two-tier model.

This subsection discusses the following topics:

- Basic client-server architecture
- History of client-server architecture
- A Simple Example
- Problems with two-tier architecture

Basic client-server architecture

The basic paradigm for client-server architecture is as follows: when an application executes, a client makes a request to a server, which processes that request and then notifies clients that it has completed the process. If requested, the server also returns information to the client.

The whole point of client-server architecture is to distribute components of an application between a client and a server so that, for example, a database can reside on a server machine (for example a UNIX box or mainframe), a user interface can reside on a client machine (a desktop PC), and the business logic can reside in either or both components. The client-server architecture, outlined here, is known as two-tier client-server architecture.

History of client-server architecture

Client-server architecture only became a reality with the advent of personal computers (PCs). Before PCs, programmes were written for mainframes. Software written for mainframes was often coded in a monolithic format—that is, the user interface, business logic, and data access functionality are all contained in a single application. Because the entire application ran within the mainframe this was not a problem.

With the advent of PCs, it became possible to off load some application processing onto a PC user's desktop. Because of this possibility and because it was economically expedient for many applications to use this paradigm, client-server architecture developed. The proliferation of UNIX-based servers about the same time as the advent of PCs helped to boost the development of client-server architecture.

A Simple Example

The UNIX print spooler is an example of a two-tier client-server architecture. The client (the UNIX lp command) reads a file to be printed and passes the file's contents to the server. The server performs a service by printing the file. All the basic characteristics of client-server computing are present in this example.

Problems with two-tier architecture

Two-tier client-server architecture has a number of problems. For example, if database access functionality (such as embedded database queries) and business logic are contained in the client component, any changes to the business logic, database access, or even the database itself often requires a new client component to be deployed for all users of the application. Usually, the effects of such changes would break earlier versions of the client component, resulting in a fragile system. The problems of two-tier client-server architecture led to the development of three-tier client-server architecture.

Three-Tier Client-Server Architecture

Overview

This section summarizes the development of three-tier client-server architecture from the two-tier model. It also describes the characteristics and advantages of the three-tier model.

This subsection discusses the following topics:

- The three-tier model
- Characteristics of Three-Tier Applications
- Advantages of three-tier architecture

The three-tier model

The most popular type of n-tier client-server architecture to evolve from two-tier architecture was three-tier architecture, which separated application components into three logical tiers: the user interface tier, the business logic tier, and the database access tier. In this type of system, the user interface tier communicates only with the business logic tier, never directly with the database access tier. The business logic tier communicates both with the user interface tier and the database access tier. For this model:

- The user interface tier is a client only, in that it only makes requests to the business logic tier.
- The database access tier is a server only, in that it only responds to requests from the business logic tier.
- The business logic tier acts as both a client and a server: a server relative to the user interface tier, because it process its request, and a client to the database access tier, because it sends a request to it.

Characteristics of Three-Tier Applications

Three-tier client-server architecture has the following characteristics:

- Three-tier applications partition an application logically into three components.
- Communication between each logical tier can be tightly controlled.

Note: Although the business logic tier and the database access tier are logically separate entities, there is no reason why they can't reside on the same server machine.

Advantages of three-tier architecture

Three-tier client-server architecture has the following advantages over a typical mainframe architecture:

- The ability to separate logical components of an application ensures that applications are easy to manage.
- Because communication can be controlled between each logical tier of an application, changes in one tier, for example, the database access tier, do not have to affect the client component tier, which would have to be redistributed if any changes are made to it.

Quiet often in COBOL and PL/I applications, changes to one part of the application can cause unforeseen changes to other parts of the application.

Distributed Systems

Overview

This section summarizes the development of distributed systems from the multi-tier client-server model. It also describes the characteristics, advantages, and disadvantages of distributed systems.

This subsection discusses the following topics:

- Distributed system model
- The nature of distributed systems
- Characteristics of distributed systems
- Advantages of distributed systems
- Disadvantages of distributed systems

Distributed system model

Distributed systems are a logical extension of multi-tier client-server architecture. Distributed systems invariably tend to be heterogeneous—that is composed of several operating systems, running applications developed in several languages, running on various operating systems. Instead of separating the business logic tier and the data access tier, distributed systems simply exposes all functionality of the application as objects, each of which can use any of the services provided by any of the other objects in the system, or even objects in another system. This architecture blurs the distinction between client and server, because clients can create objects that behave like servers, and servers can make requests (and thus act as clients) to other servers. There is no logical limit to the number of objects a system can have.

Note: Not all distributed systems are object oriented. Some distributed systems use remote procedure calls and are therefore not object oriented. All references to distributed systems referred in this document however are CORBA based and therefore object oriented.

The nature of distributed systems

Most organizations find themselves with a variety of hardware platforms, operating systems, programming languages, and networking technologies. IT infrastructure grows by adding new components, based on the technology available. The technology changes as business requirements change, and as the available technology options evolve.

It is unlikely that homogeneous systems, where all components use the same hardware, operating system, networking technology, and programming language will ever be realized. All of these different tools have strengths and weaknesses that make them more or less suitable for different tasks. To remain competitive, organizations must be able to use the right tools for each task, but still have all this different software working together.

Characteristics of distributed systems

In a distributed system:

- Clients actively issue requests to objects in servers.
- Servers passively provide access to objects that respond to client requests.
- Clients and servers are usually in different address spaces.
- Clients and servers usually execute on several machines.

As outlined in "What are objects?" on page 16, an object is defined by its data and the operations that can be performed on that data. In a distributed system, this information must be contained in the object's interface so that other objects in the system can communicate with it. Separating an object's interface from its implementation means that other objects in the system that communicate with it do not need to know how and where the object is implemented.

The effect of this separation is that any changes to an object's implementation do not affect its interface. This means that one object can be implemented in many different ways. This allows previously unthinkable flexibility.

Note: It is important to realize that clients and servers are just roles that programs can play, and a single program can play both roles during execution.

Advantages of distributed systems

Distributed systems offer a number of advantages:

- Users can be geographically separate.

  This is important for large corporations, where business decisions must be made by people in different locations, but those decisions must be based on company-wide data.

- Multiple machines can improve performance and scalability.

  Because a client-server system is distributed over several machines, you can improve the performance and scalability in several ways. There might be multiple replicas of a server running on separate machines, so each handles only a fraction of the total number of clients. Redundant servers on separate machines can provide fail-over capability, to ensure service in the event of a problem on one machine.
- Heterogeneous systems can use the best tools for each task.

  Different components of an application can run on hardware that is optimized for a specific task. For example, an application might need to retrieve large amounts of statistical or experimental data from a database, perform complex computations on that data (such as computing a weather model), and display the results of that computation in the form of maps. By running the database, the computational engine, and the graphics rendering engine on hardware that is optimized for each task, performance can improve dramatically.
- Distributed systems can reduce maintenance costs.

  For example, by upgrading an application image on a single server, it is possible to upgrade thousands of clients.

Disadvantages of distributed systems

Often, organizations that attempt to create a client-server architecture themselves quickly run into serious problems, especially in heterogeneous environments. Almost every aspect of client-server computing presents very serious technical challenges. Organizations must take care of issues relating to the integration of different networking technologies, hardware architectures, operating systems, communication protocols, as well as error recovery, support for multiple languages, and a whole host of other issues. Attempting to accomplish these tasks internally leads to massive expenditure on infrastructure, and with little chance of success: the technology is complex, few organizations have the skills to solve all the problems or, if they do, might not be able to solve the problems on time. Moreover, any home-grown solution is likely to be proprietary and therefore not open to integration with products from other vendors.

Distributed Systems and Middleware

Overview

This section summarizes how distributed systems brought about the need for middleware. It also notes how the internet has highlighted the importance of middleware.

This subsection discusses the following topics:

- The need for middleware
- Middleware and the internet

The need for middleware

Middleware is a generic term used to describe software that mediates with other software and allows for communication between disparate applications in a heterogeneous system. The need for middleware arises when distributed systems become too complex to manage efficiently without a common interface. The need to make heterogeneous systems work efficiently across a network, and be flexible enough to incorporate frequent upgrades and additions, led to the development of middleware, which hides the underlying complexity of distributed systems.

Middleware is particularly appropriate in situations where:

- A company does not wish to replace legacy systems completely, but opts instead for a mixture of old and new technologies.
- Hardware platforms and software packages, used because of their special features, deviate from the corporate norm.
- Deregulatory legislation requires that computer systems be more flexible and open.
- There is the technical challenge of successfully integrating respective computer networks of companies as a result of a merger or acquisition.

Middleware and the internet

Middleware has come of age with the importance that organizations have given to the Internet as a means of exposing integrated applications, both inside the firewall and externally with business partners.

Object-Oriented Concepts

Overview

This sections presents the most fundamental concepts of OO programming.

In this section

This section discusses the following topics:

Classes

Overview

This subsection defines what a class in an OO application is. It explains the relationship between classes and objects, and how instances of objects are created from classes.

This subsection discusses the following topics:

- What are classes?
- Class instances
- Abstract classes
- Benefits of abstract classes

What are classes?

A class is a template for creating objects. It embodies all the information needed to create objects of a particular type. For example, an account class, which is used to create account objects, defines what type of data an account object can have and all the operations that can be carried out on that data.

Class instances

Objects created from a particular class are said to be instances of that class. For example, account objects are instances of an account class.

Abstract classes

An abstract class is a class that contains abstract methods and, therefore, an instance of it cannot be created. Abstract classes are defined so that other non-abstract classes (often called derived classes) can extend them and can be instantiated, by implementing the abstract methods.

Benefits of abstract classes

Abstract classes are used because they enable application design to be more efficient. For example I want to design an bank account application where a client requests that different operations be performed on various types of bank account objects, savings account, checking account and so on. If I can design a generic bank account object that has characteristics common to all bank account objects and possible new ones that I might want to add at some point in the future, I can use it as a template to create bank account objects that I actually want to use in the application by creating derived classes and adding (or extending them) with the extra characteristics I want.

The reason I want the generic bank account class to remain abstract is that if there are any changes or additions to the derived classes, this does not mean changes to the client, because the abstract class remains unchanged. In a procedural application program however, quiet often changes in one part of the program means changes in many other parts of the program and thus makes an application subject to changes very difficult to maintain.

Objects

Overview

This subsection defines what an object is, how it is created, and how it is uniquely identified. It also provides some real-world examples of objects, and describes how to access them.

This subsection discusses the following topics:

- What are objects?
- Creating objects
- Object identifiers
- Real-world examples
- Accessing an object

What are objects?

An object is an encapsulation of an item of data and the methods (or operations) that can be used to operate on the data. The object's attributes (the type or types of data that an object has) and methods are defined in an interface (that is separated from its implementation) through which it exposes its behavior. For an example of an object interface in Interface Definition Language (IDL) see "IDL Example" on page 94. All objects have a state. An object's state changes when an object's methods manipulate its data.

Creating objects

All OO languages must provide a means of creating objects from classes. In the case of Java a constructor method is used, and for C++ a constructor function is used. For both these languages the constructor must have the same name as the class. Also constructors never return data. Constructors are implemented in different ways depending on the language, refer to the appropriate reference material for more information on constructors.

Object identifiers

Each object in an OO application has a unique identifier, which is allocated when the object is created and is retained for the lifetime of the object. Objects that exist for only a single process (transient objects) are destroyed (along with their identifier) when the process that contains the object ends. Objects that exist after a process terminates (persistent objects), must retain their unique identifier so that they can be re-used. Often it is a database key that is used to retrieve the state of the object from the database. Object identifiers should not be confused with object references. For more information about object references see "CORBA Object References" on page 53.

Real-world examples

Objects in an OO application often correspond to real-world examples. A banking system, for example, could include objects that represent customers, accounts, ledgers, and so on. A `BankAccount` object could include a Balance attribute and `Credit`, `Debit`, and `GetBalance` methods, as shown in Figure 1:

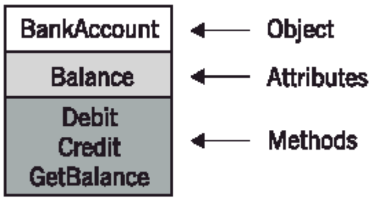

Figure 1: *An OO BankAccount Object*

Accessing an object

You can only find out about or change an object's attributes by making a request to (sending a message to) the object, specifying which method you want to invoke. An object's interface describes each message that the object responds to. A message consists of an instance name, a method name, and if required, a method's arguments.

Methods

Overview

This subsection defines what methods are, and the types of data they can access. It also tells you how a method's arguments are used.

This subsection discusses the following topics:

- What are methods?
- Instance methods and class methods
- Arguments

What are methods?

A method is the code that implements the behavior (or a particular behaviors) of an object. For example, the Credit method is the code that is executed when the credit message is sent to an BankAccount object. An object's methods are the only way to access an object's data. An object's data is encapsulated when only its methods can access it. In OO programming there are different categories of methods depending on how the method is defined. Abstract methods are methods that can not be implemented, class methods are defined within a class and can only manipulate class data not class instance data, and instance methods manipulate a class instance (object's) data.

Instance methods and class methods

The fundamental difference between an instance method and a class method is that a class method can not access an object's (class instance's) data and an instance method can not access class data. Class methods are generally used where efficient design calls for access to class data rather than class instance data.

Arguments

One or more arguments can be supplied as part of a method. For example, the `Credit` method contains an argument that specifies the amount to be credited to the balance of the account. A method's arguments are contained in brackets, for example; `Credit(float amount)` tells you the amount of money in the account defined as a float type, for instance `3445.67`.

Note: The name of a method and the arguments it takes is usually referred to as the method signature.

Inheritance

Overview

This subsection defines the principle of inheritance with respect to classes. It also outlines the main benefit of inheritance and illustrates the principle with an example.

This subsection discusses the following topics:

- Inheriting from a class
- Benefits of inheritance
- Example of inheritance

Inheriting from a class

A subclass (or a derived class) can be created from any existing class. The subclass *inherits* the methods (and therefore the behavior) of the class it is derived from (its parent class, or superclass).

Benefits of inheritance

The main benefit of inheritance is that, for any given application, classes do not have to be designed from scratch. Instead new classes can be written merely by stating how they are different from another, already existing class. Generally, the more general the characteristics of a class, the more re-usable it is. For example if I design an abstract `BankAccount` class, I can re-use it by simply inheriting its methods and attributes to write a new `SavingsAccount` class, a `CheckingAccount` class, and so on. In this way the `SavingsAccount` and `CurrentAccount` classes are extensions of the `BankAccount` class, but they do not modify it.

The beauty of inheriting from a class in this way is that creating a new derived class does not in any way affect derived classes already created, and of course new classes can be created more quickly and easily using inheritance than creating them from scratch.

Example of inheritance

At the planning stage of writing an OO program, you can establish which classes share a common set of attributes and operations and define an economical class inheritance hierarchy accordingly. For example, you can create an abstract class called `BankAccount` and two subclasses called `SavingsAccount`, and `CheckingAccount`. Each of the subclasses inherit all the `BankAccount` class's attributes (`balance`) and operations (`Credit`, `Debit`, and `GetBalance`), but can have their own individual attributes and operations (`InterestAllowed`, `OverdraftLimit`, and so on), as shown in Figure 2:

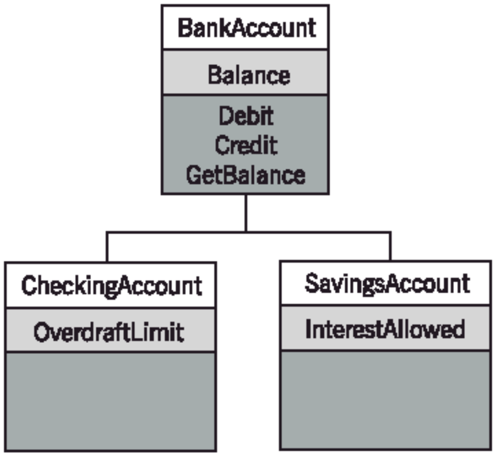

Figure 2: *BankAccount Object Class Inheritance Diagram*

Polymorphism

Overview

This subsection explains what polymorphism means and provides a simple example of how it is used in OO programming.

This subsection discusses the following topics:

- What is polymorphism?
- Illustration of polymorphism

What is polymorphism?

In general terms, polymorphism refers to the quality of being able to assume different forms. In OO programming terms, polymorphism means that objects from different classes but that are related to each other in an inheritance hierarchy behave in a different way to each other when the same method signature manipulates the object's data. The client does not need to know or understand these different behaviors, it just knows that the method signatures are identical.

Illustration of polymorphism

Consider the abstract `BankAccount` class in Figure 3 on page 22. Both `CheckingAccount` and `SavingsAccount` inherit, extend, and implement `BankAccount`. The class itself is not modified. When the `GetInterest()` method is called on the `CheckingAccount` and `SavingsAccount` objects they both return a different value, but a value of the same data type (for example a float type).

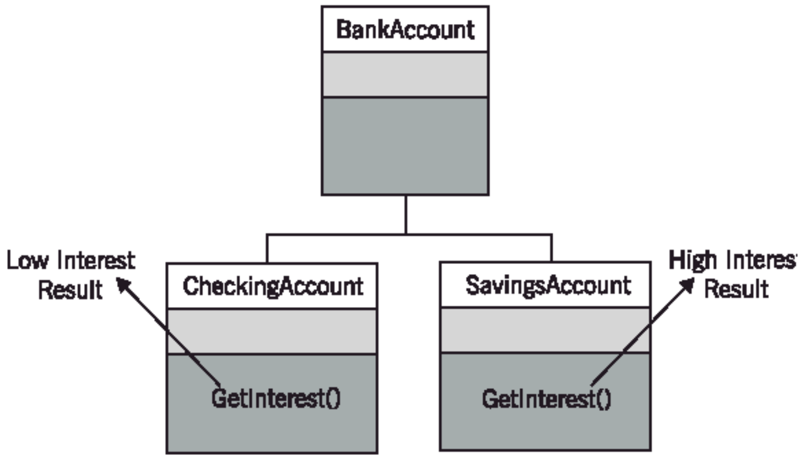

Figure 3: *BankAccount Object Polymorphism Diagram*

Banking Application Example

Overview

The purpose of the banking application outlined here is to illustrate the contrast between a procedural model and an OO model. It also highlights some benefits of an OO model. No sample code is provided.

Note: The banking application outlined here is not a completely comprehensive example of a real world banking application.

In this section

This section discusses the following topics:

Outline of the Banking Application Example

Overview

This subsection introduces the banking application example.

This subsection discusses the following topics:

- Application program description
- Data input

Application program description

The theoretical application program reads a sorted file of transaction records as input and iterates over the file until no more records are left. Each record contains an indicator of the transaction type and details of each transaction, which for this application is a sum of money to be credited or debited. Transactions are processed according to transaction type (that is, credit or debit) causing updates to be written to a database.

Also, a report is written at runtime detailing for auditing purposes date, time, all transactions processed, and the number of failed transactions including subtotals for each bank. For every failed transaction an error record should be written to a file for later examination and eventual reprocessing. A report of all failed transactions should be generated noting the key details of each transaction, that is, bank sort code, and account number.

Data input

The data input for this application is a sorted file of transaction records. Each record contains key details for each transaction, these are the bank sort code, and the account number. The file is sorted according to account number within the bank sort code. Each record contains an indicator of the transaction type and details of each transaction, which for this example is a sum of money to be credited or debited.

A Procedural Perspective

Overview

This subsection presents the banking application from a procedural programming perspective.

This subsection discusses the following topics:

- Modelling the application
- Jackson structure diagram concepts
- JSD actions for the banking application
- The JSD
- Explanation of the JSD
- COBOL and PL/I (procedural) approach

Modelling the application

Jackson Structure Diagrams (JSDs) are an effective technique for analyzing the logical structure of a procedural program. They are especially useful for logical flows that depend on a constant input source as often is the case in batch processing.

Jackson structure diagram concepts

The concepts defined here are only those used in "The JSD" on page 27.

Entity

is an object (either physical or abstract) that causes activity in a system or is affected by the system activity, or both.

Action

An action is an event that happens to an entity or that is carried out by an entity. Each action can be decomposed into smaller actions.

Sequence

A sequence is a decomposition of an entity or action into one or more actions, in which these actions are to be executed in a certain order. In the JSDs the actions in a sequence should always be executed from left to right.

Selection

A selection is a decomposition of an JSD-component into two or more actions in which only one of the containing actions is executed. Which action is executed is based upon a certain condition. A selection can best be compared with an If Then Else-statement, where the Else-clause must be present.

Iteration

An iteration is a decomposition of an JSD-component which contains only one action. Based upon a condition this action occurs (iterates) zero or more times. Thus the parent is formed of zero or more occurrences of the child.

JSD actions for the banking application

The banking application procedural program can be modelled by decomposing the program into the following JSD actions:

1. open input transaction file.
2. open output files.
3. write report header - date, time and title.
4. write error report header - date, time and title.
5. read first transaction record
6. process transaction
7. build and write error record if error
8. accumulate subtotal
9. accumulate master total
10. read next transaction record
11. write bank subtotal
12. write report footer(s) with master total.
13. close output files

The JSD

The JSD should be read from left to right and top to bottom.

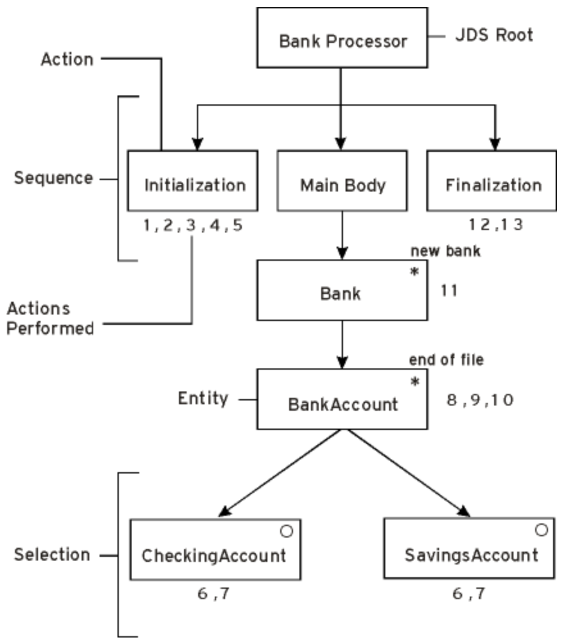

Figure 4: *Jackson Structure Diagram for the Banking Application*

Explanation of the JSD

This diagram effectively describes the program logic flow for the banking application, albeit at a very high level. The symbols have the following meaning:

The * represents an iteration, accompanying it is normally text which defines the condition on which the iteration ends. This implies that all actions defined on the substructure below this box are performed for each iteration.

In the banking application you are performing actions for each bank. In the complete structure above you also iterate over all account transactions for each bank, this allows us to perform bank specific processing at the end of each iteration, for example, print totals for each bank.

The **O** symbol represents a branch of a logical condition. In the banking application you test if you are dealing with a current or savings account and define the actions for each separately.

COBOL and PL/I (procedural) approach

A program written in PL/I or COBOL based on this model would typically have an initialization and finalization section. The main body would be represented by one main loop comprising of an if branch for recording and dealing with bank specific actions. Importantly there might be one section or subprogram for executing the actual transaction regardless of whether it is a debit or credit on a checking or savings account. There could be separate modules for lodgment and withdrawal. Within these modules, database sources are freely accessed. If during a maintenance cycle, processing for current account debit changes, then the resulting regression testing must include savings account as well. This is because the implementation for both is physically in the same module and can be to some extent shared. There can also be an online version of this program, that is, running in IMS or CICS and serving a network of ATMs. There is no guarantee that the same modules for credit and debit have been used in this program. It could be the case that the credit and debit code in the online case is completely different to the batch case and that it makes updates to the underlying database in a different way. Generally this leads to a maintenance headache. This lack of separation of concerns is the major drawback of this model.

An OO Perspective

Overview

This subsection presents the banking application from an OO programming perspective.

This subsection discusses the following topics:

An OO approach of the banking application

The corner stone concepts of object oriented design and programming are polymorphism, inheritance, and encapsulation. In most object oriented programming languages an object is implemented as a class definition. Instances of this class are the objects themselves. Use of inheritance hierarchies enables you to specify the interface of a family of objects. In this case the `BankAccount` object specifies the BankAccount interface in its most general form. You then derive concrete class from this in order to represent `CheckingAccount` and `SavingsAccount`. There are many other types of accounts in a real banking environment and they can be further derived from CheckingAccount or SavingsAccount or also directly from `BankAccount`. Because derived classes inherit methods from the super classes you can then overwrite methods such as Credit() and Debit() (latter case above) or inherit ready implementations of them as in the former case above. In other words this structure also allows code reuse.

An OO model of the banking application

Based on the “Application Program Description” on page 27 a typical OO model for the banking application might have objects that are instances of the following classes:

- An abstract `BankAccount` class which has abstract `Credit()` and `Debit()` methods.
    - A `CheckingAccount` class that inherits from, extends, and implements the `BankAccount` class.
    - `SavingsAccount` class which inherits from, extends, and implements the `BankAccount` class.

Indeed, the `BankAccount` class can be extended any number of times without affecting `CheckingAccount` or `SavingsAccount`.

- An abstract `Report` class that has abstract methods `WriteHeader()` and `WriteFooter()` and `Date`, `Time`, and `Title` attributes.
    - A `TransactionSuccessfulReport` class that inherits from and implements the `Report` class, and also extends it with a `Write_report()` method.
    - A `TransactionErrorReport` class that inherits from and implements the `Report` class, and also extends it with a `Write_error()` method.
- A Transaction class with the methods `openfile()`, `readrecord()`, `process_transaction()`, `accumulate_subtotal()`, `accumulate_master total()`, and `closefile()`.

There is a clear distinction between reports and accounts. Reports are also represented by an inheritance hierarchy. There is an abstract base class called Report - this means there are no direct instances of this class. This serves to define a common interface to all reports. Status and error reports are derived from report.

Application maintenance and implementation

Object orientation gives us a very clear separation of concerns. The implementation of `BankAccount` is separated from the implementation of `Report`. This makes maintenance much easier. Furthermore the implementation of `credit()` and `Debit()` for `CheckingAccount` and `SavingsAccount` is also separate.

Initialization and finalization

In object oriented programming initialization and finalization are performed by class constructors and destructors. Constructors provide a uniform means to create an instance of a class and perform those actions necessary to create an instance in a consistent manner. A class destructor performs those actions necessary to destroy it again. The class destructor is called automatically at various times depending on programming language. For example for C++ automatic variables when the variable goes out of scope. For Java, free store variables when garbage collection runs. Constructors and destructors are very often used to acquire and free up resources needed by an object. For example, an error report constructor would typically open an output file for the report whereas the destructor would close it again.

Encapsulation

It is important to understand that `BankAccount` data can only be accessed through the methods defined in the `BankAccount` interface, that is encapsulation. This ensures that data is accessed in a consistent manner from every point in the system.

Polymorphism and inheritance

If you call `Credit()` with an instance of `CheckingAccount` then you invoke the corresponding method in the `CheckingAccount` class implementation. If you call `credit()` with an instance of the `SavingsAccount` class then you invoke the `credit()` method in the `SavingsAccount` class implementation. Furthermore you can pass instances of `CheckingAccount` and `SavingsAccount` where instances of `BankAccount` are expected.

For example you can safely pass instances of `CheckingAccount` or `SavingsAccount` to have the following method.

```
Boolean Transfer_funds(in account from, in account to)
```

In the implementation you can call `credit()` or `Debit()` without having to test what kind of an account this is. This is because `CheckingAccount` and `SavingsAccount` are derived from `BankAccount` and have methods `credit()` and `Debit()`, and object oriented programming languages are able to determine which class instance is passed. This ability to pass a derived class instance where a base is expected is called polymorphism.

Procedural versus OO approach

In procedural programming you can typically have a module that performs an action such as “place an order” or “make a lodgment”. In this module, the first task is to further characterize the input parameters and follow a certain specific logical path through the module. Also underlying data can be accessed in any way. In object oriented programming this catch all module is replaced by classes representing the different characteristics of the input parameters above. The classes typically form an inheritance hierarchy and you can therefore ensure:

- A uniform interface to data sources underneath.
- You can reuse code through direct implementation inheritance.
- You can define implementations specific to the derived class for the methods defined in the class.

Introduction to CORBA

This chapter outlines the historical development of Common Object Request Broker Architecture (CORBA). It also outlines the structure and function of its various elements, and gives a generic overview of the development of a CORBA system.

In this chapter

This chapter discusses the following topics:

CORBA Object Management

Overview

This section discusses at a high level how CORBA object management came about, the basic architecture for CORBA object management, characteristics of CORBA objects, the shortcomings and advantages of CORBA objects.

In This Section

This section discusses the following topics:

The Object Management Group (OMG)

Background

The OMG was founded in 1989 by a group of eleven companies, and now includes several hundred members worldwide. Its main aim is to provide technically sound, commercially viable, vendor- independent specifications for the software industry, which combine two strands of technology: remote procedure calls and object orientation.

This subsection discusses the following:

- Achievement of the OMG
- Operation of the OMG

Achievement of the OMG

The OMG produced a complete infrastructure for distributed computing, (see "The Object Management Architecture (OMA)" on page 36). A core part of the OMA which describes the basic software infrastructure needed to support distributed objects, is the Request Broker Architecture (ORB) standard.

Operation of the OMG

OMG produces specifications - that is, documents that precisely describe what something should do, and how it should act, perhaps in response to various inputs. The OMG CORBA specification describe software; OMG IDL describes language. Implementations of the OMG CORBA specifications — such as ORBs, and IDL compilers — are not produced by OMG. They are, instead, produced by software vendors.

Specifications are available free of charge and are downloadable[1]. These free, downloadable specifications, can be implemented by anyone. Specifications are frequently added to and updated in response to the changing needs of CORBA users and enterprise requirements.

1. http://www.omg.org

The Object Management Architecture (OMA)

Overview

The OMA is a set of standard interfaces for standard objects that support CORBA applications. It includes the CORBAservices, the CORBAfacilities, and a set of Domain Specifications.

This section discusses the following:

- The OMA and CORBA
- OMA's central component
- OMA diagram
- OMA Components

The OMA and CORBA

When the basic model was established, the OMG set about defining the framework for CORBA, CORBAservices, and CORBAfacilities, the three components that constitute the OMG's implementation of the OMA.

OMA's central component

The central component of the architecture is the Object Request Broker (ORB) which acts as the communication backbone, or object bus that objects use to communicate. CORBA specifies the architectural framework for ORBs, but the ORB design is vendor specific

OMA diagram

Figure 5 illustrates the OMA.

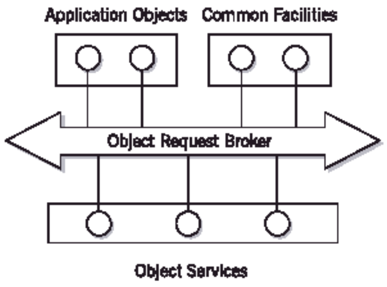

Figure 5: *OMA Diagram*

OMA Components

The architecture identifies three different kinds of object:

Application Objects

These are objects created specifically for individual applications by the developer of the application.

Object Services

These are objects that provide horizontal services, such as object naming, event delivery, and transaction coordination. These services are useful to a wide variety of applications and are standardized by the OMG.

Common Facilities

These are objects that provide vertical, or application-specific services and are also specified by the OMG. Examples are a workflow facility, telecom logging, or a printing service. None of the services and facilities objects are privileged in any way, anyone can implement them. The objects that provide these services and facilities are not distinguished in any way as far as the ORB is concerned. They are like any application object that plugs into the bus. (It just so happens that they are standardized by the OMG; that is, as CORBAservices, and CORBAfacilities.

Object Orientation and CORBA Objects

Overview

This section introduces CORBA objects in the context of a distributed system with OMA. It also discusses how they are accessed, created and destroyed, and their states.

This section discusses the following:

- CORBA objects
- CORBA object messages
- CORBA object states
- Transient CORBA objects
- Persistent CORBA objects
- Creating a persistent CORBA object
- Destroying a CORBA object

CORBA objects

In C++ or Java an object's interface and implementation can be described in the same language. For CORBA objects, the object's interface is defined in Interface Definition Language (IDL) which is a description language not an implementation language. It can only be implemented in one of the languages ratified by the OMG, including COBOL, PL/I, C++ and Java. The IDL compiler compiles CORBA object interfaces and produces implementation code for example in COBOL and PL/I.

CORBA object messages

In CORBA terminology, a client invokes an operation on a CORBA object located on a server. To make a successful invocation on a CORBA object, the client must obtain the object reference, for that particular object. The object reference is a unique identifier that encapsulates the location and all other necessary information needed to access and manipulate the object. Each object reference is unique to that object for the object's lifetime and can be used to access the object any number of times by invoking an operation (or set or get an attribute) on the object via its reference.

CORBA object states

A CORBA object is an abstraction that exists independently of servers, servants, or any other computing artifact. A servant is a programming language object that provides the implementation for one or more methods defined for a CORBA object. In effect it is simply an instance of an implementation of an IDL interface for a CORBA object. At runtime, COBOL and PL/I developers perceive CORBA objects as implementations in either PL/I or COBOL.

Servers can be shut down and restarted without affecting the existence of a CORBA object that lives in that server. The minimum requirement for a CORBA object to exist is that it must have an `ObjectId`. The way ObjectIds are created depends on the nature of the CORBA object they are associated with. So, at this point it necessary to make a fundamental distinction between two different types of CORBA object:

- Transient CORBA object.
- Persistent CORBA object.

Transient CORBA objects

Transient CORBA objects are CORBA objects that last only as long as the server process that uses them.

Persistent CORBA objects

Persistent CORBA objects, are CORBA objects that exist beyond a particular instance of a server process. They are therefore stored (usually in a database), when they are not actively being used.

Creating a persistent CORBA object

The necessary prerequisite for creating a persistent CORBA object is that an ObjectId has already been created, and is stored in some database. When the server process begins, it associates a servant with the CORBA object's ObjectId. The object reference is then created and can be passed to the client, enabling the client to invoke operations on the CORBA object. When the server process ceases, the CORBA object becomes dormant, and is only reactivated when the server process is restarted.

Destroying a CORBA object

Destroying a CORBA object not only means that is it not active; it also means that it cannot be activated to respond to a client request. Clients calling on a reference to an object that has been destroyed receive the following error message (commonly called an exception)—`OBJECT_NOT_EXIST`. The OMG have ratified a CORBAservice which describes a paradigm for creating and destroying CORBA objects.

Shortcomings of CORBA

Summary of CORBA shortcomings

The shortcomings of CORBA are discussed as follows:

- OMG and Source Code
- CORBA backward compatibility
- CORBA and application design

OMG and Source Code

There is no definitive reference implementation for CORBA. This places a larger burden on the specification, because it must define semantics sufficiently well to eliminate portability and interoperability problems.

CORBA backward compatibility

Occasionally, specifications are found to have defects that must be addressed by publishing a revised specification. Revisions sometimes (but fortunately only rarely) are not backward-compatible with existing implementations, forcing source-code changes in application code.

CORBA and application design

CORBA provides you with considerable freedom in how to implement an application. This is both a boon and a bane. While CORBA can be used in a wide variety of applications and deployment scenarios, like any other powerful tool, it can be used in detrimental ways. For example, no amount of CORBA magic can compensate for a poor design; the large number of choices offered by CORBA means that, if you do not understand the consequences of design decisions, you could run into serious problems.

Advantages of CORBA

Summary of CORBA advantages

The advantages of CORBA are discussed as follows:

- Object location transparency
- Server Transparency
- Language Transparency
- Implementation Transparency
- Architecture Transparency
- Operating System Transparency
- Protocol Transparency

Object location transparency

The client does not need to know where an object is physically located. An object can either be linked into the client, run in a different process on the same machine, or run in a server on the other side of the planet. A request invocation looks the same regardless, and the location of an object can change over time without, breaking applications.

Server Transparency

The client is, as far as the programming model is concerned, ignorant of the existence of servers. The client does not know (and cannot find out) which server hosts a particular object, and does not care whether the server is running at the time the client invokes a request.

Language Transparency

Client and server can be written in different languages. This fact encapsulates the whole point of CORBA; that is, the strengths of different languages can be utilized to develop different aspects of a system, which can interoperate through IDL. A server can be implemented in a different language without clients being aware of this.

Implementation Transparency

The client is unaware of how objects are implemented. A server can use ordinary flat files as its persistent store today and use an OO database tomorrow, without clients ever noticing a difference (other than performance).

Architecture Transparency	The idiosyncrasies of CPU architectures are hidden from both clients and servers. A little-endian client can communicate with a big-endian server with different alignment restrictions.
Operating System Transparency	Client and server are unaffected by each other's operating system. In addition, source code does not change if you need to port the source from one operating system to another[2].
Protocol Transparency	Clients and servers do not care about the data link and transport layer. They can communicate via token ring, Ethernet, wireless links, ATM (Asynchronous Transfer Mode), or any number of other networking technologies.

2. Of course, this is true only for CORBA-related code. If you use features that are specific to a particular operating system in your application code, you must still port the application.

Components of a CORBA Distributed System

Overview

This section discusses the core components of a CORBA distributed system. There is an overview of IDL and how its basic types are mapped to COBOL and PL/I, an introduction to object references, an introduction to the generic functionality and structure of ORBs, an introduction to the most common transport protocol used to communicate between distributed ORBs and finally CORBAservices and CORBAfacilities are summarized.

In This Section

This section discusses the following topics:

Interface Definition Language

Overview

The first key concept in CORBA is (IDL). IDL is a neutral language for defining abstract interfaces. IDL interfaces have no executable statements. You cannot code with IDL, only type definitions. IDL only defines the operations that are available, and the data types of their parameters. These interface definitions can be implemented in the application source code. It does not say anything about the implementation of the interface: in particular it does not give any clues about what language the source code the interface might be implemented in, or what networking protocols might be used to reach an object with that interface.

This subsection discusses:

- IDL interfaces
- Implementing source code
- Interface components
- Operation definitions
- Attribute definitions
- Exception definitions
- Data type definitions
- Constant Types

IDL interfaces

Interfaces are the fundamental abstraction mechanism of CORBA. An interface defines a type of object, including the operations that object supports in a distributed enterprise application. Every CORBA object has exactly one interface. However, the same interface can be shared by many CORBA objects in a system. CORBA object references specify CORBA objects (that is, interface instances). Each reference denotes exactly one object, which provides the only means by which that object can be accessed for operation invocations. Because an interface does not expose an object's implementation, all members are public. A client can access variables in an object's implementation only through an interface's operations and attributes.

Implementing source code

IDL can be implemented by languages for which the OMG specify a language mapping. Implementation is achieved by the IDL compiler, which produces client stub and server skeleton code for the implementation language you choose when you compile you IDL.

Interface components

An IDL interface definition typically has the following elements:

- Operation definitions.
- Attribute definitions.
- Exception definitions.
- Data type definitions.
- Constant definitions.

Operation definitions

IDL operations define the signatures of an object's function, which client invocations on that object must use. The signature of an IDL operation is generally composed of three parts:

- Return value data type.
- Parameters and their direction.
- Exception clause.

An operation's return value and parameters can use any data types that IDL supports.

Attribute definitions

An interface's attributes correspond to the variables that an object implements. Attributes indicate which variables in an object are accessible to clients. Unqualified attributes map to a pair of `get` and `set` functions in the implementation language, which let client applications read and write attribute values. An attribute that is qualified with the keyword read-only maps only to a get function.

Exception definitions

IDL operations can raise one or more CORBA-defined system exceptions. System exceptions are not, and should not be defined in IDL. You can also define you own exceptions (called user exceptions) by means of a raises clause, and explicitly specify these in an IDL operation. An IDL exception is a data structure that can contain one or more member fields.

Data type definitions

IDL defines its own data-type definitions, which map to various languages that the OMG supply mapping rules for.

Constant Types

IDL lets you define constants of all IDL supported basic-data types except the any type, refer to the *COBOL Programmer's Guide and Reference* or *PL/I Programer's Guide and Reference* for more details of the any type. To define a constant's value, you can either use another constant (or constant expression) or a literal. You can use a constant wherever a literal is permitted.

Mapping IDL to Implementation Languages

Overview

IDL is mapped onto various programming languages by standard mapping rules, defined by the OMG, that relate IDL data types, operations, and interfaces to their corresponding form in a particular programming language. The OMG currently supports language mappings for the following languages: Ada, C, COBOL, C++, COM, LISP, Java, PL/I, Python, Smalltalk, and XML.

This subsection discusses:

- Language mapping for OO languages
- Language mappings for COBOL and PL/I
- IDL basic types mapped to COBOL and PL/I

Language mapping for OO languages

Java and C++ are the two most commonly used OO programming languages for commercial applications. CORBA IDL data types are very similar to C++ and Java data types. IDL interfaces map easily to C++ and Java classes, and IDL operations map to C++ member functions and Java methods.

If C++ programmers are implementing CORBA objects, they implement them as C++ objects. They only need to understand the mapping rules that relate IDL data types, interfaces, and operations to C++ data types, classes, and member functions. If Java programmers are implementing CORBA clients for the same objects, they only need to understand the Java mapping that relates IDL data types, interfaces, and operations to Java data types, interfaces, classes, and methods.

Language mappings for COBOL and PL/I

Because COBOL and PL/I are not OO languages, CORBA objects are not implemented in COBOL and PL/I in the same way they are in OO languages such as C++ and Java.

IDL basic types mapped to COBOL and PL/I

Table 1 shows the mapping rules for basic IDL types (types not currently supported by Orbix COBOL are denoted by italic text).

Table 1: *Mapping Rules for Basic IDL Types for COBOL and PL/I*

IDL Type	COBOL Representation	PL/I Representation
short	PIC S9(05) BINARY	FIXED BIN(15)
long	PIC S9(10) BINARY	FIXED BIN(31)
unsigned short	PIC 9(05) BINARY	FIXED BIN (15)[a]
unsigned long	PIC 9(10) BINARY	FIXED BIN (31)[a]
float	COMP-1	FLOAT DEC (6)
double	COMP-2	FLOAT DEC (16)
char	PIC X	CHAR (1)
boolean	PIC 9(01) BINARY	CHAR (1)
octet	PIC X	CHAR (1)
enum	PIC S9(10) BINARY	FIXED BIN (31)[a,b]
fixed<d,s>	PIC S9(d-s)v(s) PACKED-DECIMAL	FIXED DEC (d,s)
fixed<d,-s>	PIC S9(d)P(s) PACKED-DECIMAL	FIXED DEC (d,s)
any	See the *COBOL Programmer's Guide and Reference* for details.	See the *PL/I Programmer's Guide and Reference* for details.
long long	*PIC S9(18) BINARY*[c]	FIXED BIN (31)[d]
unsigned long long	*PIC 9(18) BINARY*[c]	FIXED BIN (31)[d]
wchar	*PIC G*	GRAPHIC

a. `UNSIGNED FIXED BIN` is not supported by earlier versions of the PL/I compiler. Therefore, the maximum length for a PL/I unsigned short is half that of the CORBA-defined equivalent. The same applies for a PL/I unsigned long CORBA type.

b. The maximum number of digits allowed for the PL/I representation of an enum is 31 bits.

c. Due to restrictions in earlier versions of the COBOL compiler, only up to 18 significant digits were originally supported. Because more recent COBOL compilers provide an `ARITH(EXTEND)` option that provides support for 31 digits, the IDL compiler also supports 31 digits when the `-E` IDL compiler argument is used.

d. The maximum number of digits allowed in a `FIXED BIN` is 31 bits if you are using earlier versions of the PL/I compiler, or 63 bits if you specify the `-E` option with the Orbix IDL compiler, and are using a version of the IBM Enterprise PL/I for z/OS compiler.

Mapping IDL operations to COBOL and PL/I

For COBOL each IDL interface maps to a group of data definitions. There is one definition for each IDL operation. A definition contains each of the parameters for the relevant IDL operation in their corresponding COBOL representation.

For PL/I each IDL interface maps to a group of data definitions. There is one structure defined for each IDL operation. A structure contains each of the parameters for the relevant IDL operation in their corresponding PL/I representation.

Mapping IDL attributes to COBOL and PL/I

For both COBOL and PL/I write attributes map to two operations (get and set), and read-only attributes map to a single get operation. Storage of Mapped Data For COBOL, the mapped data is contained in copybooks that are generated by the IDL compiler. For a summary of the COBOL generated files see, “Generated COBOL Members” on page 95.

For PL/I, the mapped data is contained in include files that are generated by the IDL is compiler. For a summary of the generated PL/I files see, “Generated PL/I Members” on page 110.

CORBA Object References

Overview

This section introduces the essential components of an object reference.

This subsection discusses:

- What's in a reference?
- Protocol-specific information
- Object key

What's in a reference?

All object references carry three essential items of information:

- Repository ID
- Protocol-specific information
- Object key

Repository ID

The repository ID encodes the IDL information of the object as a unique identifier string in the Interface Repository (IFR)F at the time the object reference is created. Every named IDL type is associated with a repository ID.

Protocol-specific information

This part of the reference carries one or more addresses that identify a communication end point. For IIOP ("General Inter-ORB Protocol and Internet Inter-ORB Protocol" on page 56), each address is an Internet domain name or IP address, and a TCP port number. The address (or addresses) can either identify the server, or identify a location broker that can return the address of the server. Logically, the protocol information serves to identify the process that should handle requests made via the object reference.

Object key

The object key contains the ObjectId and ORB-specific information. It is in a format that is not specified by the OMG. The proprietary information does not get in the way of interoperability, because it is never looked at, except by the ORB that created that information.

ORB Functionality

Overview

This section summarizes ORB functionality.

This subsection discusses the following topics:

- Basic functions of an ORB
- ORBs and clients
- Orb components

Basic functions of an ORB

The following are the most fundamental functions of an ORB. An ORB must:

- Bridge the separation of an object's interface from its implementation.
- Provide, to the client, an interface to access objects.
- Locate the correct object for each client request.
- Transmit messages from the client to the object.
- Start the object's server, if it is not already started.
- Invoke the application code in the object's server.
- Return results or errors to the client.
- Deactivate the server, if required.

ORBs and clients

The ORB must transparently dispatch requests sent by a client to the correct object and return the results of requests back to the client. The client need not know an object's implementation details (such as what language it is implemented in, where it is physically located, and how it can be reached). In other words, all clients care about is the interface of an object, its behavior, and its identity. The details of how to communicate with the object are handled by the ORB.

Orb components

Figure 6 shows the components of an ORB.

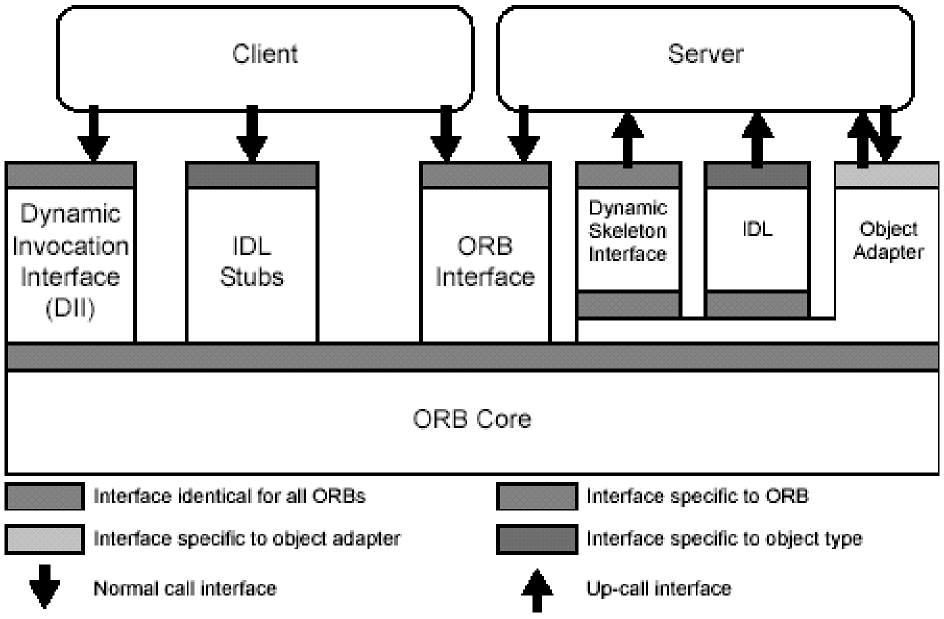
Client
Server
Dynamic Invocation Interface (DII)
IDL Stubs
ORB Interface
Dynamic Skeleton Interface
IDL
Object Adapter
ORB Core
Interface identical for all ORBs
Interface specific to ORB
Interface specific to object adapter
Interface specific to object type
Normal call interface
Up-call interface

Figure 6: *The Components of an ORB*

ORB Structure

Overview

The ORB core, which is proprietary to each implementation, handles the basic communication functions of the platform. The interface to the core is vendor-specific. However, application components never access this interface directly.

This subsection discusses the following topics:

- ORB interface
- IDL stubs
- IDL skeletons
- Dynamic invocation interface
- Dynamic skeleton interface
- Object adapter
- Portable object adapter

ORB interface

The ORB interface is identical for both clients and servers, and standardized. It is concerned with tasks such as initialization and finalization.

IDL stubs

IDL stubs are files produced by a compiler that converts IDL definitions into an Application Programming Interface (API) for a specific programming language. Clients use the generated stubs to access objects. The precise API that is generated depends on the interface definition, and so varies for each type of object. However, the rules for converting IDL into stubs are specified by the OMG. The code that is generated for the stubs takes care of marshaling parameters that are sent from the client to an object, and unmarshaling of values that are returned from an operation invocation to the client.

IDL skeletons

An IDL skeleton—which is a superset of the stub code—is the server-side equivalent of a stub. It provides an interface from the ORB into the application code. Like stubs, IDL skeletons are compiled and therefore provide a type-specific API between the ORB and the server application code. The skeletons contain the code that marshals and unmarshals values on the server-side.

Dynamic invocation interface

IDL stubs are compiled from IDL, and therefore require compile-time knowledge of the interfaces a client wants to use. With DII, invocations can be constructed at runtime by specifying the target object reference, the operation or attribute name, and the parameters to pass. A server that receives a dynamically constructed invocation request does not differentiate between it and static requests. The DII is used mostly by generic applications, such as debuggers or protocol bridges.

Dynamic skeleton interface

The Dynamic Skeleton Interface (DSI) is the server-side equivalent of the DII. It permits a server to implement objects whose interface definition was not known at compile time. Because the type information is not known at compile time the ORB must be able to retrieve the necessary type information it needs at runtime from the IFR. Therefore the IDL needs to be registered in the IFR.

You can use DSI and DII together to construct a bidirectional gateway. This gateway receives messages from the non-CORBA system and uses the DII to make CORBA client calls. It uses DSI to receive requests from clients on a CORBA system and translate these into messages in the non-CORBA system.

Object adapter

Object adapters mediate between the ORB core and programming-language objects. Object adapters are responsible for a variety of tasks, such as keeping track of which objects exist in memory, allowing the server to create and destroy objects. In addition, object adapters are responsible for mapping an incoming request onto whatever language construct is used to represent an object.

By necessity, object adapters are language-specific, because it is the object adapter that bridges the gap between the abstract concept of an object and its specific implementation in a particular programming language.

Note: Neither DII nor DSI are available with current version of Orbix Mainframe.

Portable object adapter

The OMG defines the Portable Object Adapter (POA), but, the characteristics of the POA are defined by POA policies. POA policies determine how the POA implements and manages objects, and processes client requests.

The language-specific interfaces of the POA are defined separately for each language for which CORBA standardizes a mapping. COBOL, and PL/I, specific interfaces of the POA support only one set of POA policies. For a list of these policies, refer to "Orbix Mainframe POA Policy" on page 81.

General Inter-ORB Protocol and Internet Inter-ORB Protocol

Overview

A protocol is a set of formal rules describing how to transmit data, especially across a network. High level protocols deal with the data formatting, including the syntax of messages, the terminal to computer dialogue, character sets, sequencing of messages. In order to bridge the gap between different programming languages, platforms, and so-on, CORBA needs a common protocol that can carry requests to objects regardless of where they are located on a network, or how they are implemented. From a programmer's perspective, the protocol details are irrelevant. Programmers deal with making calls on objects; the ORB automatically and transparently turns those calls into network messages in the appropriate protocol.

This subsection discusses the following topics:

- GIOP
- Other transport protocols

GIOP

The General Inter-ORB Protocol (GIOP) is an abstract meta-protocol. It specifies a standard transfer syntax (how data is represented as bits and bytes) and a set of message formats for object requests. The GIOP is designed to work over many different transport protocols. IIOP The Internet Inter-ORB Protocol (IIOP) (see) specifies how GIOP messages are exchanged, using TCP/IP connections. IIOP provides interoperability between different vendors ORBs. TCP/IP, which underpins the Internet, is the most popular product-neutral and vendor-neutral transport layer used.

Other transport protocols

CORBA requires that all ORBs support IIOP, but leaves the field open for other transports that are suitable to special situations. For example, Orbix supports shared memory and SSL-based variations of the GIOP. Other transports (including multi-cast) will be available in the future.

CORBAservices and CORBAfacilities

Overview

CORBAservices and CORBAfacilities are a group of basic services and facilities that are commonly needed by a wide variety of applications. CORBAservices and CORBAfacilities are specified as CORBA objects with IDL interfaces, and can therefore, be accessed and manipulated like any other object in a CORBA distributed system.

Most up-to-date services and facilities

For the most complete and up to date information on CORBAservices and CORBAfacilities see `http://www.omg.org`.

In this section

This sections discusses the following topics:

CORBAservices

Overview

CORBAservices are basic services that can be used in a wide variety of applications and are ratified by the OMG. Table 2 gives the current complete list of services. This list is subject to change.

Table 2: *CORBAservices (Sheet 1 of 2)*

Service	Description
Collection	Manages and moves groups of CORBA objects.
Concurrency	Enables the performance of distributed locking used by the Object Transaction Service (OTS).
Event	Basic messaging service that controls clients or servers sending messages (events) to receivers.
Externalization	Enables CORBA data types to be stored externally.
Interoperable Naming	Basic directory service which stores name to object reference bindings in a central location.
Licensing	Manages the licensing of a distributed application.
Life Cycle	Describes a paradigm for remotely creating and destroying CORBA objects.
Notification	A messaging service that allows configuration for quality-of-service and message filtering.
Persistent State	CORBA-type persistence mechanism for defining how objects stored in databases are reused.
Property	Associates names with properties.
Query	Manipulates groups of CORBA objects.
Relationship	Defines relationships between pairs of CORBA objects.

Table 2: *CORBAservices (Sheet 2 of 2)*

Service	Description
Security	Describes an ORB’s provision of secure communications and the various security levels obtainable.
Time	Provides current time and various other time-relevant services.
Trading Object	Facilitates object discovery by querying an object’s properties.
Transactions	Provides support for distributed transactions, thus allowing secure simultaneous updates of databases.

CORBAfacilities

Overview

CORBAfacilities are services that many applications can share, but are not as fundamental as the CORBAservices.

This subsection discusses the following topics:

- CORBAfacilities
- Internationalization and time
- Mobile Agent Facility

CORBAfacilities

To date, the OMG have ratified two CORBAfacilities:

- Internationalization and time.
- Mobile Agent Facility.

Internationalization and time

This CORBA Facility is concerned with culturally specific representations of data, such as representations of time, monetary formatting, collation order, and other culturally specific ways of representing data.

Mobile Agent Facility

This CORBAfacility is concerned with interoperability between various mobile agent facilities. An agent is a computer program that acts autonomously on behalf of a person or an organization, and has its own thread of execution, allowing tasks to be performed on the program’s own initiative.

A mobile agent is one that is not confined to the system where it was executed, in that it can move itself from one system in a network to another.

Vertical CORBAfacilities

Vertical CORBAfacilities are domain-specific facilities that are developed by OMG working groups for particular industries. Currently there are specifications for the following domains:

- CORBA Business
- CORBA E-Commerce
- CORBA Finance
- CORBA Life Sciences
- CORBA Manufacturing
- CORBA Medicine
- CORBA Telecoms
- CORBA Transportation

The CORBA Development Process

Overview

This section outlines how the elements of a CORBA system are used to develop distributed applications.

This section discusses the following topics:

- CORBA development process
- Key points of the development process

CORBA development process

Figure 7 illustrates the CORBA development process.

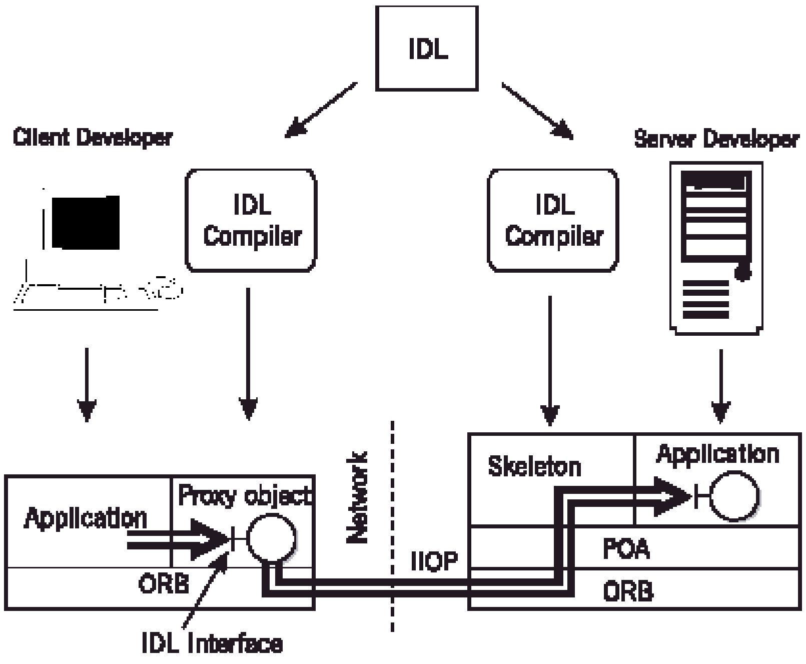

Figure 7: *CORBA Development Process*

Key points of the development process

The key points are:

IDL allows separation of client and server development: The only thing that client and server developers need to agree on is the IDL interface. They can use different programming languages, hardware platforms, even different vendors ORBs.

IDL compiler maps the IDL to programming languages: developers use their ORB's IDL compiler to map the IDL interface onto their chosen programming language. For example, the client developer might be implementing a Java client, whereas the server developer might be implementing a COBOL server.

Note: The Orbix IDL compiler cannot generate COBOL and PL/I client stub code.

Proxy objects provide client-side transparency: The IDL compiler creates stub code that contains a class defining proxy objects for a particular interface. The ORB creates a proxy object based on this class when an object reference (for the object that implements that interface) enters the clients address space. This gives the illusion that the remote object is in the local address space. The ORB then forwards the client calls on the proxy object to the corresponding target object across the network.

Skeletons provide server-side transparency: The IDL compiler creates a server-side skeleton in your chosen server implementation language (for example COBOL). This server skeleton code allows the ORB to transfer client request through to target server objects, and allows the server to service these client requests on the target object.

The ORB and POA mediate the request: The client-side ORB sends a CORBA request across the network to the server-side ORB, which locates the appropriate POA for the object. The POA turns the CORBA request into a function call on the correct programming language object.

CHAPTER 3

Introduction to Orbix

This chapter describes the basic components of Orbix that are relevant to Orbix Mainframe. Orbix consists of the ORB core, which contains the Adaptive Runtime Technology framework and an open-ended set of plug-ins that provide the required functionality for specific systems.

In this chapter

This chapter discusses the following topics:

The Orbix ORB Core

Overview

The ORB core handles the most basic CORBA abstractions: CORBA objects, requests, interceptors, and plug-ins. The ORB core presents a uniform programming interface to developers. In the ORB core, everything is a CORBA object. These CORBA objects' have IDL interfaces. The ORB core passes requests between these objects via their IDL interfaces hiding the implementation details. This means the ORB can allow many different ways of implementing key objects transparently to the rest of the framework.

This section discusses the following topics:

- Adaptive runtime technology
- The ORB core and CORBA objects
- The ORB core and interceptors

Adaptive runtime technology

The Adaptive Runtime Technology (ART) framework encapsulates the ORB core and plug-ins and insulates them from each other. The ORB core handles invocations (requests) on remote objects. To the developer, invocations on remote objects appear to be exactly the same as invocations on local objects. This location transparency is a key feature of the ORB.

The ORB core does not contain any networking code; support for networking protocols, for example, the standard IIOP protocol is provided by a separate plug-in. Different features (such as alternative transport protocols) are provided by alternative implementations of key interfaces in this abstract framework.

The ORB core and CORBA objects

The ORB uses memory-management and memory-sharing techniques to minimize memory use of CORBA objects on both the client-side and server-side. It is designed to handle large numbers of objects, and to make fine-grained object designs practical.

The ORB core and interceptors

Plug-ins that manipulate requests (for example the transport plug-ins) provide an interceptor to the ORB core. The ORB core does not care how any particular transport works, it simply passes a request to its interceptor. Supporting new or multiple transports, does not require any changes to the ORB core.

Orbix Plug-ins

Overview

This section discusses the following topics:

- Plug-ins
- ART plug-in architecture
- Plug-in examples
- The Orbix IDL compiler
- WTO announce plug-in
- SAF plug-in
- Log-stream plug-in example
- Transport protocol plug-ins

Plug-ins

A plug-in is a dynamically loaded shared object library that registers with the ORB plug-in framework. Most ORB functionality is implemented by plug-ins. The set of plug-ins is determined by configuration settings, so the ORB core also knows how to load configuration data. Plug-ins are loaded at run-time, so the features you want can be selected using run-time configuration settings. You can change the plug-in feature set by re-configuring your application—you don't have to re-compile it.

ART plug-in architecture

The plug-in architecture enables new capabilities to be added to Orbix in the form of plug-ins, without modifying the ORB core. This allows new features to be supported more quickly and easily, and with fewer version-incompatibility problems than a monolithic ORB architecture. Figure 8 illustrates the Orbix plug-in architecture:

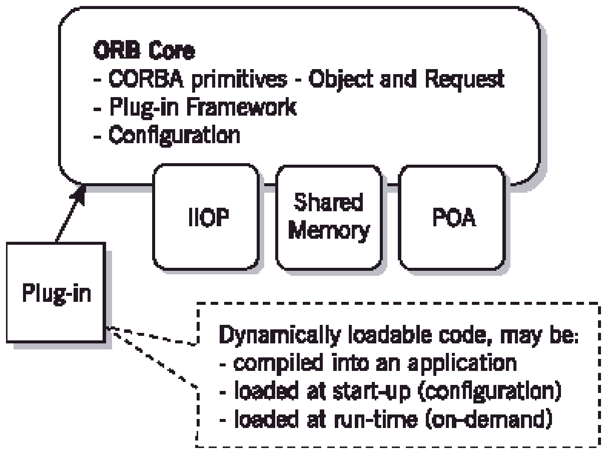
ORB Core
- CORBA primitives - Object and Request
- Plug-in Framework
- Configuration
IIOP
Shared Memory
POA
Plug-in
Dynamically loadable code, may be:
- compiled into an application
- loaded at start-up (configuration)
- loaded at run-time (on-demand)

Figure 8: *Orbix Plug-In Architecture*

Plug-in examples

The standard IIOP protocol, the optimized shared-memory protocol, and the POA server-side support are all packaged as plug-ins. Plug-ins allow a degree of flexibility, both in choice of plug-in and the time a plug-in is activated. For example, suppose an application is sometimes deployed on a single host, and sometimes spread across multiple hosts. You can deploy identical executables, but configure a single-host deployment to load just the shared-memory transport, whereas a multi-host deployment loads the IIOP transport.

The Orbix IDL compiler

In Orbix, the IDL compiler follows the same plug-in architecture as IIOP, the POA, and so on. You must specify the language you wish to use for the client and server. For example, if the client is to be implemented in Java and the server in COBOL, the IDL compiler produces client stub code in Java and server skeleton code in COBOL. For more detailed information on IDL compiler output for COBOL see "IDL to COBOL Mapping" on page 99, and for PL/I see "IDL to PL/I Mapping" on page 113.

WTO announce plug-in

For external monitoring and automation purposes, the following messages can be written when an Orbix server starts up and later ends on z/OS:

```
+ORX2001I ORB orbname STARTED (app-id)
+ORX2002I ORB orbname ENDED (app-id)
```

These messages can be enabled in any server without code changes, by configuring the `orb_plugins` list for the server to include the name `wto_announce`.

SAF plug-in

This Orbix plug-in provides optional Principal-based access control. A server can accept or reject incoming requests based on a `CORBA::Principal` value in the request header. The value is treated as an z/OS user ID and access is checked against an operation-specific SAF profile name. Access can therefore be controlled on a per-operation basis, or (using generic profiles) on a per-server basis

Log-stream plug-in example

Orbix logs diagnostic messages to a log-stream object. Several log-stream implementations are provided as plug-ins. When developing an application, you would most likely use the local log stream, which logs to standard output or to a file. However, in a deployed system you might want to use the system log stream that logs to the operating system's logging service (UNIX syslogd or Windows NT logging service). You can configure the same executable to load either (or both) of these plug-ins, depending on the context in which you run the application.

Transport protocol plug-ins

Most applications use the IIOP transport plug-in, but other protocols are possible. For optimized communication on a single host, Orbix provides a shared-memory transport. No particular transport is special to the ORB core, so you can load whatever transport set you need for your application. This architecture makes it easy to add support for additional transports in the future; for example, multi-cast or support for special-purpose network protocols.

Orbix Interceptors

Overview

This section discusses the following topics:

- Interceptors
- Plug-ins and interceptors
- Interceptor set-up
- Bindings

Interceptors

An interceptor is an implementation of an interface that the ORB uses to process requests. These are abstract request handlers that can implement transport protocols (like IIOP or shared memory), or manipulate requests on behalf of a service (for example, adding transaction identity). Plug-ins that manipulate requests (for example, the transport plug-ins) provide an interceptor to the ORB core. These plug-ins can adapt the ORB to new network protocols. The ORB core does not care how any particular transport works, it simply passes a request to its interceptor. Supporting a new transport, or multiple transports, does not require any changes to the ORB core.

Plug-ins and interceptors

Figure 9 shows an example of plug-ins that contain interceptors.

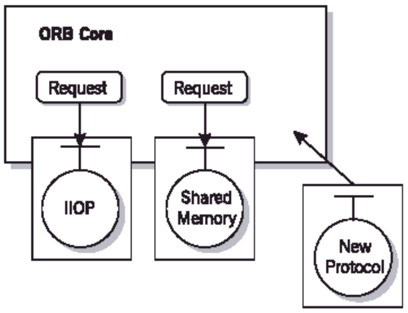

Figure 9: *Orbix Plug-Ins that Contain Interceptors*

Interceptor set-up

Interceptors can be set up in chains. The ORB passes a request to the first interceptor in the chain, which passes it to the next, and so on. For example, an implementation of the transaction service would provide an interceptor to add transaction context to a request before it is passed to the transport interceptor.

Bindings

Choosing an interceptor chain to use for an object reference is called binding. Binding is controlled by:

- Configuration settings, which specify the default bindings that the ORB should try.
- Information in an object reference, which specifies which protocols the object understands.

The same application can be re-configured to use different protocols, by simply changing the binding configuration, or passing an object reference that uses a new protocol. There is no need to change or re-compile application code to change protocols.

Orbix Location Domain

Overview

A location domain is a collection of servers under the control of a single server—the locator daemon. The locator daemon acts as a forwarding agent, forwarding clients to the appropriate server.

This section discusses the following topics:

The locator daemon

A single locator daemon can:

- Locate servers on many hosts.
- Start servers remotely, using a stateless node daemon running on the host where the server is started.
- Maintain the Implementation Repository (IMR), which is a database of available servers.
- Provide transparent load balancing across a group of objects, without any special action by clients.
- Correctly re-direct clients, regardless of where the server is running.
- Transparently redirect connected clients to a new location

Locator daemon benefits

Using the locator daemon provides two benefits:

- By interposing the locator daemon between client and server, a location domain isolates the client from changes in the server address. If the server changes location - for example, it restarts on a different host, or moves to another port - the IORs for persistent objects remain valid. The locator daemon supplies the server's new address to clients.

- Because clients contact the locator daemon first when they initially invoke on an object, the locator daemon can launch the server on behalf of the client. Thus, servers can remain dormant until needed, thereby optimizing use of system resources.

Components of an Orbix location domain

An Orbix location domain consists of two components: a locator daemon and a node daemon:

locator daemon: A CORBA service that acts as the control center for the entire location domain. The locator daemon has two roles:

- Manage the configuration information used to find, validate, and activate servers running in the location domain.
- Act as the contact point for clients trying to invoke on servers in the domain.

node daemon: Acts as the control point for a single host machine in the system. Every machine that runs an application server must run a node daemon. The node daemon starts, monitors, and manages application servers on its machine. The locator daemon relies on node daemons to start processes and tell it when new processes are available.

Invoking on persistent objects

When a client invokes on a persistent object, Application Server Platform locates the object as follows:

1. When a client initially invokes on the object, the client ORB sends the invocation to the locator daemon.
2. The locator daemon searches the implementation repository for the actual address of a server that runs this object in the implementation repository. The locator daemon returns this address to the client.
3. The client connects to the returned server address and directs this and all subsequent requests for this object to that address.

All of this work is transparent to the client. The client never needs to contact the locator daemon explicitly to obtain the server's location.

Object references

There is no change to object references held by clients or stored in a repository such as the Naming or Trader service. A server[3] can be moved to a new host without invalidating references to an object in the server. Moving a server does not require updates to the Naming Service, Trader Service, or any other repository of object references.

Note: The IORs of persistent objects are exported from their server with the address of the domain's locator daemon. This daemon is associated with a database, or implementation repository, which dynamically maps persistent objects to their server's actual address.

Load balancing

In Orbix, the locator daemon is extended to balance client load over a group of objects that have the same object reference. This is completely transparent to the client, and does not require that the object reference come from any particular source (for example the Naming Service). If the client connection is broken, the client transparently reconnects—it does not need to go back to the source of the reference to locate a new instance of the object.

The server process

The server process is just a container for a collection of POAs (each POA implements some collection of CORBA objects). It is actually the POAs that are registered, not the servers, so individual POAs can move between servers or hosts.

3. Most applications migrate entire servers, but actually the unit of migration is a POA, which can be a subset of the objects in a server.

Orbix Configuration Domain

Overview

The configuration domain is a collection of applications under common administrative control.

This section discusses the following topics:

- Orbix configuration
- System size and configuration
- Configuration options
- Default settings

Orbix configuration

Almost everything about Orbix can be set by configuration. The plug-in architecture lets you configure what ORB capabilities you need, and each plug-in has its own configuration settings to fine tune its behavior (for example, network time-outs and error logging). Configuration is organized into domains and scopes, so you can control the configuration of specific applications or sets of applications independently.

System size and configuration

Since configuration is so fundamental, Orbix provides a flexible configuration mechanism:

- Configuration can be loaded from a simple text file during development, or for small-scale deployment.
- For larger deployments, Orbix provides a distributed configuration server that enables centralized configuration for all applications spread across a network from a single point of control.

The configuration system is open to extension, so ORB configuration could come from any kind of configuration repository.

Configuration options

Within a configuration domain, the configuration database has a hierarchical structure that lets you fine tune your system. You can:

- Specify global configuration settings for all the clients and servers in your system.
- Override specific settings for particular groups of applications.
- Specify detailed configuration for specific clients and servers.

Depending on your needs, you can make the configuration as fine-grained or as coarse-grained as you wish

Default settings

Sensible defaults are provided for all configuration values. This means that you do not need to worry about specific settings until you need them.

The Orbix Portable Object Adapter

Overview

The Portable Object Adapter (POA) is a runtime library of routines that are built into the server application executable instance. The POA maps objects to their actual implementations. It converts requests which have come from network clients via the ORB into appropriate calls to server application code. The POA is the only OMG-ratified object adapter.

Note: The POA functionality outlined in this section is not relevant to COBOL or PL/I development. It is therefore included as background information only, to present a complete view of Orbix.

This section discusses the following topics:

- POA states
- POA functions
- Servants
- ObjectId
- Object activation
- Object deactivation
- Root POA
- POA policies

POA states

POAs can be transient or persistent. Their state depends on the state of the CORBA objects that are implemented by the POA. Orbix Mainframe does not support transient POAs.

POA functions

All POAs perform the following functions:

- Create object references to all objects used by application.
- Manage object states for all objects used by an application.
- Map object references to servants when clients make requests.
- Support the portability of object implementations between different ORB products.

- Create servants and use them to activate objects on demand, as requests for these objects arrive.

Servants

Each POA has a number (possibly zero) of associated servants, each of which incarnates one or more CORBA objects. Often, a POA has many servants, but, depending on how you want to structure your application, you can also have POAs that only have a single servant. As a rule, each servant belongs to exactly one POA at a time[4].

ObjectId

The POA uses the ObjectId, see "Persistent CORBA objects" on page 39, to identify the object and choose the correct servant. An ObjectId must be unique within a POA, but different POAs can use the same ObjectId to refer to different objects.

The associations between ObjectIds and servants are very flexible and, by setting POA policies, you can control how associations are established and for how long they are remembered. This fine-grained control is one of the major scalability features of the POA, because it enables you decide on the most appropriate memory consumption versus performance trade-off.

Object activation

When you activate an object, you inform the POA of the association between an `ObjectId` and the programming language servant for that ID. In other words, you tell the POA which servant should handle requests for this particular object.

Object deactivation

When you deactivate an object, you break the association between the `ObjectId` and the servant. A deactivated object cannot accept requests, because it has no associated servant. Deactivating an object does not imply destruction of the servant or the object; it merely means that, for a certain time, no servant is associated with the CORBA object.

Note: An incoming request can be processed only if the server-side runtime knows which servant can handle the request.

4. In principle it is possible to use a single servant with more than one POA, but there are few reasons to do so and it significantly complicates memory management in your server.

Root POA

A server always has at least one POA, namely the root POA. The root POA has a fixed set of standard policies that cannot be changed. The root POA has default values for all policies except `ImplicitActivationPolicy`, and allows implicit activation.

POA policies

Every POA has an associated set of policies. Policies govern aspects of how the POA associates requests with servants. For example, there are policies to control whether the POA uses multiple threads to dispatch requests. All servants for a particular POA share the same set of policies. You assign policies when you create the POA; they remain in effect until the POA is destroyed.

Orbix Mainframe POA Policy

Overview

The Orbix COBOL and PL/I runtimes can use only one set of POA policies. The arguments for these policies are set and cannot be changed for COBOL and PL/I development, unlike C++ and Java development where POA policies play a very important role in application development. They are outlined here merely to illustrate an implementation detail.

Note: The POA policies described in this chapter are the only POA policies that the Orbix COBOL and PL/I runtimes support. Orbix COBOL and PL/I programmers have no control over these POA policies. They are outlined here simply for the purposes of illustration and the sake of completeness.

Summary

Table 3 describes the policies that are supported by the Orbix COBOL and Orbix PL/I runtimes, and the argument used with each policy.

Table 3: *POA Policies Supported by the COBOL and PL/I Runtimes (Sheet 1 of 2)*

Policy	Argument Used	Description
Id Assignment	`USER_ID`	This policy determines whether ObjectIds are generated by the POA or the application. The `USER_ID` argument specifies that only the application can assign ObjectIds to objects in this POA. The application must ensure that all user-assigned IDs are unique across all instances of the same POA. `USER_ID` is usually assigned to a POA that has an object lifespan policy of `PERSISTENT` (that is, it generates object references whose validity can span multiple instances of a POA or server process, so the application requires explicit control over ObjectIds).
Id Uniqueness	`MULTIPLE_ID`	This policy determines whether a servant can be associated with multiple objects in this POA. The `MULTIPLE_ID` argument specifies that any servant in the POA can be associated with multiple ObjectIds.

Table 3: *POA Policies Supported by the COBOL and PL/I Runtimes (Sheet 2 of 2)*

Policy	Argument Used	Description
Implicit Activation	`NO_IMPLICIT_ACTIVATION`	This policy determines the POA's activation policy. The `NO_IMPLICIT_ACTIVATION` argument specifies that the POA only supports explicit activation of servants.
Lifespan	`PERSISTENT`	This policy determines whether object references outlive the process in which they were created. The `PERSISTENT` argument specifies that the IOR contains the address of the location domain's implementation repository, which maps all servers and their POAs to their current locations. Given a request for a persistent object, the Orbix daemon uses the object's virtual address first, and looks up the actual location of the server process via the implementation repository.
Request Processing	`USE_ACTIVE_OBJECT_MAP_ONLY`	This policy determines how the POA finds servants to implement requests. The `USE_ACTIVE_OBJECT_MAP_ONLY` argument assumes that all ObjectIds are mapped to a servant in the active object map. The active object map maintains an object-servant mapping until the object is explicitly deactivated via `deactivate_object()`. This policy is typically used for a POA that processes requests for a small number of objects. If the ObjectId is not found in the active object map, an `OBJECT_NOT_EXIST` exception is raised to the client. This policy requires that the POA has a servant retention policy of `RETAIN`.
Servant Retention	`RETAIN`	The `RETAIN` argument with this policy specifies that the POA retains active servants in its active object map.
Thread	`SINGLE_THREAD_MODEL`	The `SINGLE_THREAD_MODEL` argument with this policy specifies that requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all calls by a single-threaded POA to implementation code (that is, servants and servant managers) are made in a manner that is safe for code that does not account for multi-threading. This policy determines whether the POA works in a single-threaded or a multi-threaded environment.

CHAPTER 4

Introduction to Orbix Mainframe

This chapter provides an overview of the CORBA development process applied to Orbix Mainframe.

In this chapter

This chapter discusses the following topics:

Orbix Applications Model

Overview

This section presents a model for an Orbix banking application called First Northern Bank (FNB). It discusses the following topics:

- Orbix banking model
- FNB business architecture
- Mainframe back-end
- Bank teller applications
- ATM network
- Internet banking
- System administration
- Overseas banking network

Orbix banking model

Figure 10 shows how Orbix can be used to integrate various software and hardware environments in a sample banking application.

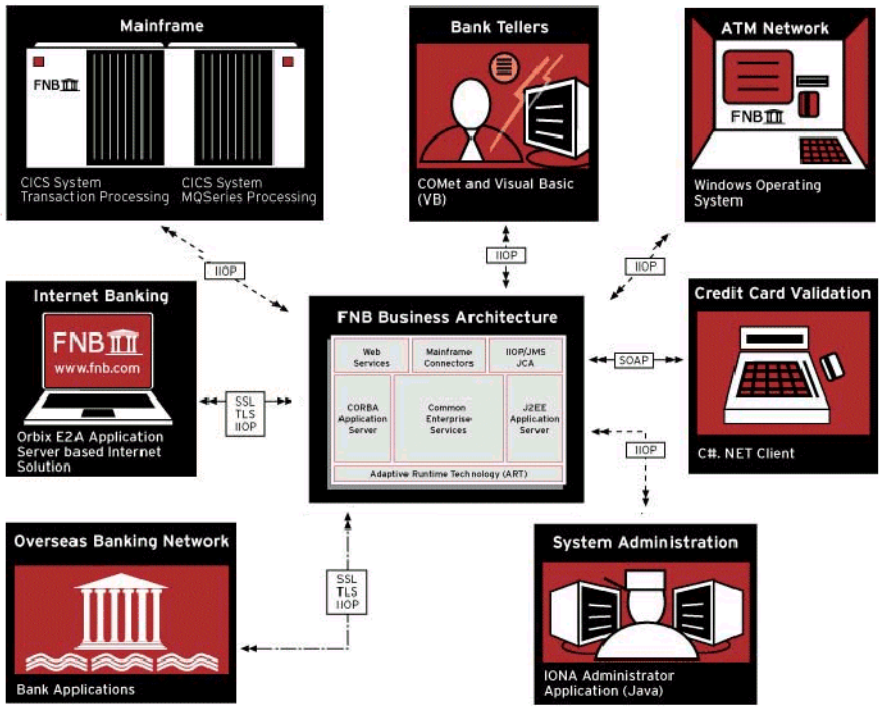

Figure 10: *An Orbix Banking Application Example*

FNB business architecture

This is the middle tier in the bank's new architecture. It is based on Orbix and all clients (such as the ATMs and tellers) use this system to gain access to the bank's resources. Since it is based on Orbix, features such as fault tolerance, load balancing, and security are available.

Mainframe back-end	Because a lot of the bank's data and functionality are stored on a mainframe, migrating from it at this point would be too costly and fraught with risk. Using the Orbix Mainframe, a CORBA wrapper has been placed around the mainframe, enabling access to it from any CORBA-compliant client. This represents the bank-end of the bank's three-tier architecture.
Bank teller applications	These Windows-based applications can be written in a language such as Visual Basic (VB) and use COMet to access the FNB Business Architecture. A new GUI front-end is used to display information to the bank teller in a more useful way.
ATM network	The ATMs use Orbix to communicate with the FNB Business Architecture.
Internet banking	The Internet banking site has been upgraded to use the Orbix Application Server to provide a Java-based Internet banking site. Security is provided using Secure Sockets Layer (SSL) and Transport Layer Security (TLS).
System administration	Key to the success of this network is the ability to examine and re-configure elements of the middle-tier system. The ability to get this information is provided using the administration GUI tool.
Overseas banking network	Any additional banks can use SSL and the Internet Inter-ORB Protocol (IIOP) to allow secure, standards-based communication to take place. The FNB Business Architecture exposes IDL interfaces, which can be used by these banks to update their systems and allow better integration.

Orbix Development Process

Overview

This section describes the basic CORBA development process applied to the preceding banking application where a Java applet running in a browser window (a client), communicates with a COBOL server program that manages database access (a server).

Note: The development steps outlined here represent only a small part of the model outlined in the previous section.

This section discusses the following topics:

- Summary of development steps
- Development steps

Summary of development steps

Every Orbix application begins with the creation of the IDL interfaces, which are then compiled by the Orbix IDL compiler. The Orbix IDL compiler uses the Java plug-in for the client program, resulting in Java class stub code for each client-side IDL interface. The COBOL IDL compiler plug-in is used for the server program, which generates three copybooks and two source members for each server-side IDL interface. When the application is ready to execute, the client locates the server in its z/OS environment, supports the interface or interfaces, establishes an Object Reference to this service, and calls operations on the service (in Java these appear as simple local object calls).

Development steps

The development steps are as follows (steps to execute the application are not included):

Step	Action
1	Define public IDL interfaces to the objects defined in your system.
2	Compile the IDL definitions, using the Orbix IDL compiler.

Step	Action
3	Develop the server programs that implement the IDL interface, using the IDL compiler output as a starting point. The generated COBOL code can be used to call legacy COBOL applications.
4	Develop the client program(s), using the IDL compiler output as a starting point.

Defining IDL Interfaces

Overview

This section defines and explains the IDL interfaces Bank and Account from the `Bankdemo` IDL member, which ships as one of the demonstrations with Orbix Mainframe. This IDL is used to illustrate the "Orbix Development Process" on page 87.

This section discusses the following topics:

- IDL example
- IDL example explained

IDL example

The `Bankdemo` IDL contains two interfaces, `Bank` and `Account`:

Example 1: *Bankdemo IDL*

```
//IDL

// Bank interface...used to create Accounts

interface Bank

  {

      exception AccountAlreadyExists { AccountId account_id; };

      exception AccountNotFound { AccountId account_id; };


      Account

      find_account(

          in AccountId account_id

      ) raises(AccountNotFound);


      Account
```

Example 1: *Bankdemo IDL*

```
        create_account(
            in AccountId account_id,
            in CashAmount initial_balance
        ) raises (AccountAlreadyExists);

        void
        shutdown_bank();
    };
// Account interface...used to deposit, withdraw, and query
// available funds.
interface Account
    {
        exception InsufficientFunds {};

        readonly attribute AccountId account_id;
        readonly attribute CashAmount balance;

        void
        withdraw(
            in CashAmount amount
        ) raises (InsufficientFunds);


        void
```

Example 1: *Bankdemo IDL*

```
        deposit(
            in CashAmount amount
        );
    };
```

IDL example explained

The preceding IDL Bank interface defines operations which:

- Create new accounts.
- Find existing accounts.

It also defines exceptions that can be raised by these operations and an operation to shut down the application.

The preceding IDL Account interface defines two attributes:

- `account_id` which is of type `AccountId`.
- `balance` which is of type `CashAmount`.

It defines operations to withdraw money from an account or deposit money to it. It also defines an exception that can be raised by the withdraw operation if there are insufficient funds in the account balance.

Orbix IDL Compiler Arguments

Overview

This section describes the various arguments that you can specify as parameters to the Orbix IDL compiler.

For a complete description of these options see the *COBOL Programmer's Guide and Reference* or *PL/I Programmer's Guide and Reference*.

This section discusses the following topics:

- Summary of the Arguments for COBOL
- Summary of the Arguments for PL/I
- The IDLPARM DD name
- IDLPARM line format

Summary of the Arguments for COBOL

The Orbix IDL compiler arguments can be summarized as follows:

`-D`	Generates source code and copybooks into specified directories rather than the current working directory. (This is relevant to z/OS UNIX System Services only.)
`-E`	Generates support for arithmetic extended types.
`-I`	Used to process a specific interface.
`-M`	Set up an alternative mapping scheme for data names.
`-O`	Override default copybook names with a different name.
`-Q`	Indicate whether single or double quotes are to be used for string literals in COBOL copybook members.
`-S`	Generate server mainline source code.
`-T`	Indicate whether server code is for batch, IMS, or CICS.
`-Z`	Generate server implementation source code.

All of these arguments are optional.

Summary of the Arguments for PL/I

The Orbix IDL compiler arguments can be summarized as follows:

`-D`	Generates source code and include files into specified directories rather than the current working directory. (This is relevant to z/OS UNIX System Services only.)
`-E`	Generate support for Enterprise PL/I.
`-L`	Limit code and typecode generation; generate inherited typedefs.
`-M`	Set up an alternative mapping scheme for data names.
`-O`	Override default include member names with a different name.
`-S`	Generate server implementation skeleton code.
`-T`	Indicate whether server code is for batch, IMS, or CICS.
`-V`	Do not generate the server mainline code.
`-W`	Control whether generated code uses `display` or `put` for messages

All of these arguments are optional.

The IDLPARM DD name

To denote the arguments that you want to specify as parameters to the compiler, you can use the DD name, `IDLPARM`, in the JCL that you use to run the compiler.

IDLPARM line format

The parameters for the `IDLPARM` entry in the JCL take the following format:

```
// IDLPARM='-cobol[:-M[option][membername]][:-Omembername]
      [:-Q[option]][:-S][:-T[option]][:-Z]'
```

Running the Orbix IDL Compiler

Overview

You can use the Orbix IDL compiler to generate COBOL source modules and copybooks, and PL/I modules and include members from IDL definitions.

This section discusses the following topics:

- Orbix IDL Compiler Configuration
- Required DD Cards
- Running the Orbix IDL Compiler

Orbix IDL Compiler Configuration

The Orbix IDL compiler uses the Orbix configuration member for its settings. The JCL that runs the compiler uses the IDL member in the `orbixhlq.CONFIG configuration` PDS.

Required DD Cards

Before you run the Orbix IDL compiler to create COBOL or PL/I source, ensure that the `IDLMAP` DD card is defined. This PDS contains the mapping member, which is generated if you use the `-M` argument with the Orbix IDL compiler.

Running the Orbix IDL Compiler

For the purposes of this example, the source module is generated in the first step of the following job (that is, the JCL supplied with the `bankdemo` demonstration). For COBOL this is:

```
orbixhlq.DEMO.COBOL.BLD.JCLLIB(BANKIDL)
```

For PL/I this is:

```
orbixhlq.DEMO.PLI.BLD.JCLLIB(BANKIDL)
```

Generated COBOL Members

Overview

This section describes the various COBOL source code and copybook members that the Orbix IDL compiler generates.

This section discusses the following topics:

- Generated members
- Member name restrictions

Generated members

Table 4 provides an overview and description of the COBOL members that the Orbix IDL compiler generates, based on the IDL member name.

Table 4: *COBOL Members Generated by the Orbix IDL Compiler (Sheet 1 of 2)*

Member Name	Member Type	Compiler Argument Used to Generate	Description
*idlmembername*S	Source code	-Z	This is server implementation source code member. It contains stub paragraphs for all the callable operations. It is only generated if you specify the -Z argument.
*idlmembername*SV	Source code	-S	This is the server mainline source code member. It is only generated if you specify the -S argument.
idlmembername	Copybook	Generated by default	This copybook contains data definitions that are used for working with operation parameters and return values for each interface defined in the IDL member.
*idlmembername*X	Copybook	Generated by default	This copybook contains data definitions that are used by the Orbix COBOL runtime to support the interfaces defined in the IDL member. This copybook is automatically included in the *idlmembername* copybook.

Table 4: *COBOL Members Generated by the Orbix IDL Compiler (Sheet 2 of 2)*

Member Name	Member Type	Compiler Argument Used to Generate	Description
*idlmembername*D	Copybook	Generated by default	This copybook contains procedural code for performing the correct paragraph for the request operation. This copybook is automatically included in the *idlmembername*S source code member.

Member name restrictions

If the IDL member name exceeds six characters, the Orbix IDL compiler uses only the first six characters of the IDL member name when generating the source and copybook member names. This allows space for appending the two-character SV suffix to the name for the server mainline code member, while allowing it to adhere to the eight-character maximum size limit for z/OS member names. In such cases, each of the other generated member names is also based on only the first six characters of the IDL member name, and is appended with its own suffix, as appropriate.

COBOL API Reference Summary

Overview

This section provides a summary of the API functions, in alphabetic order, which appear in the sample code in IDL to COBOL Mapping.

Summary Listing

The following is a list of COBOL APIs that appear in the sample code produced by the IDL compiler for the `bankdemo` IDL. For a complete list of the COBOL APIs available refer to the *COBOL Programmer's Guide and Reference*.

Example 2: *COBOL APIs used in the bankdemo IDL Output*

```
COAERR(in buffer user-exception-buffer)

// Allows a COBOL server to raise a user exception for an
// operation.

COAGET(in buffer operation-buffer)

// Marshals in and inout arguments for an operation on the server
// side from an incoming request.

COAPUT(out buffer operation-buffer)

// Marshals return, out, and inout arguments for an operation on
// the server side from an incoming request.

COAREQ(in buffer request-details)

// Provides current request information.

ORBEXEC(in POINTER object-reference,
        in X(nn) operation-name,
        inout buffer operation-buffer,
        inout buffer user-exception-buffer)
// Invokes an operation on the specified object.

ORBSTAT(in buffer status-buffer)

// Registers the status information block.
// Frees the memory allocated to a bounded string.
```

Example 2: *COBOL APIs used in the bankdemo IDL Output*

```
STRGET(in POINTER string-pointer,
       in 9(09) BINARY string-length,
       out X(nn) string)

// Copies the contents of an unbounded string to a bounded
   string.
```

IDL to COBOL Mapping

Overview

This section illustrates how IDL is mapped by the Orbix IDL compiler to COBOL code. It uses the `bankdemo` IDL to illustrate how IDL operations, attributes, and user defined exceptions are mapped to COBOL. For a complete list of the IDL to COBOL mappings refer to the *COBOL Programmer's Guide and Reference*.

In This Section

This section discusses the following topics:

Mapping for Operations

Overview

This subsection describes how IDL operations are mapped to COBOL. The code samples are based on the bank interface in the “IDL example” on page 89.

IDL-to-COBOL Mapping for Operations

An IDL operation maps to a number of statements in COBOL, as follows:

1. A `01` group level is created for each operation. This group level is defined in the `bank` copybook and contains a list of the parameters and the return type of the operation. If the parameters or the return type are of a dynamic type (for example, sequences, unbounded strings, or anys), no storage is assigned to them. The `01` group level is always suffixed by `-ARGS` (that is, `BANK-FIND-ACCOUNT-ARGS`).
2. A `01` level is created for each operation, with a `PICTURE` clause that contains a size that can hold the largest operation name length plus one, which is, for the preceding IDL:

```
* COBOL
01 BANK-OPERATION  PICTURE X(37).
```

The extra space is added because the operation name must be terminated by a space when it is passed to the COBOL runtime by `ORBEXEC`.

A level `88` item is also created as follows for each operation, with a value clause that contains the string literal representing the operation name. For the `find_account`, `create_account`, and `shutdown_bank` operations in the bank interface the following level `88` items are generated:

```
 * COBOL
88 BANK-FIND-ACCOUNT              VALUE
            "find_account:IDL:BankDemo/Bank:1.0".
88 BANK-CREATE-ACCOUNT            VALUE
            "create_account:IDL:BankDemo/Bank:1.0".
88 BANK-SHUTDOWN-BANK             VALUE
            "shutdown_bank:IDL:BankDemo/Bank:1.0".
```

A level 01 item is also created, as follows, that defines the length of the maximum string representation of the interface operation. For the bank interface this 01 level item is:

```
 * COBOL
01 BANK-OPERATION-LENGTH            PICTURE 9(09) BINARY
                                           VALUE 37.
```

3. The preceding identifiers in point 2 are referenced in a select clause that is generated in the BANKD copybook. This select clause calls the appropriate operation paragraphs, which are discussed next.
4. The operation procedures are generated in the BANKS source member when you specify the -Z argument with the Orbix IDL compiler. For example:

   i. Consider the bank interface IDL:

```
//IDL
interface Bank
{
   exception AccountAlreadyExists { AccountId account_id; };
   exception AccountNotFound      { AccountId account_id; };

   Account
   find_account(
       in AccountId account_id
   ) raises(AccountNotFound);

    Account
    create_account(
        in AccountId account_id,
        in CashAmount initial_balance
    ) raises (AccountAlreadyExists);

    void
    shutdown_bank();
};
```

   ii. Based on the preceding IDL, the following COBOL is generated in the BANK copybook:

```
* COBOL
*****************************************************************
* Interface:,
*     BankDemo/Bank
*
* Mapped name:
*     Bank
*
* Inherits interfaces:
*     (none)
*****************************************************************
*****************************************************************
* Operation:       find_account
* Mapped name:     Bank-find_account
* Arguments:       <in> BankDemo/AccountId account_id
* Returns:         BankDemo/Account
* User Exceptions: BankDemo/Bank/AccountNotFound
*****************************************************************
 01 BANK-FIND-ACCOUNT-ARGS.
  03 ACCOUNT-ID                        POINTER
                                       VALUE NULL.
  03 RESULT                            POINTER
                                       VALUE NULL.
*****************************************************************
* Operation:       create_account
* Mapped name:     Bank-create_account
* Arguments:       <in> BankDemo/AccountId account_id
*                  <in> BankDemo/CashAmount initial_balance
* Returns:         BankDemo/Account
* User Exceptions: BankDemo/Bank/AccountAlreadyExists
*****************************************************************
 01 BANK-CREATE-ACCOUNT-ARGS.
  03 ACCOUNT-ID                        POINTER
                                       VALUE NULL.
  03 INITIAL-BALANCE                   COMPUTATIONAL-1.
  03 RESULT                            POINTER
                                       VALUE NULL.
```

```
*****************************************************************
* Operation:        shutdown_bank
* Mapped name:      Bank-shutdown_bank
* Arguments:        None
* Returns:          void
* User Exceptions: none
*****************************************************************
 01 BANK-SHUTDOWN-BANK-ARGS.
  03 FILLER                         PICTURE X(01).
```

iii. The following code is also generated in the BANK copybook:

```
* COBOL
***********************************************************
* Operation List section
* This lists the operations and attributes which an
* interface supports
***********************************************************
01 BANK-OPERATION                   PICTURE X(37).
        88 BANK-FIND-ACCOUNT               VALUE
            "find_account:IDL:BankDemo/Bank:1.0".
        88 BANK-CREATE-ACCOUNT             VALUE
            "create_account:IDL:BankDemo/Bank:1.0".
        88 BANK-SHUTDOWN-BANK              VALUE
            "shutdown_bank:IDL:BankDemo/Bank:1.0".
01 BANK-OPERATION-LENGTH            PICTURE 9(09) BINARY
                                           VALUE 37.
```

iv. The following code is generated in the BANKD copybook member:

```
* COBOL
EVALUATE TRUE
    WHEN BANK-FIND-ACCOUNT
      PERFORM DO-BANK-FIND-ACCOUNT
    WHEN BANK-CREATE-ACCOUNT
      PERFORM DO-BANK-CREATE-ACCOUNT
    WHEN BANK-SHUTDOWN-BANK
      PERFORM DO-BANK-SHUTDOWN-BANK
END-EVALUATE
```

v. The following is an example of the code in the BANKS source member:

```
* COBOL
PROCEDURE DIVISION.
    ENTRY "DISPATCH".
    CALL "ORBSTAT" USING ORBIX-STATUS-INFORMATION.
    CALL "COAREQ" USING REQUEST-INFO.
    SET WS-COAREQ TO TRUE.
    PERFORM CHECK-STATUS.

* Resolve the pointer reference to the interface name which
* is the fully scoped interface name

CALL "STRGET" USING INTERFACE-NAME
                        WS-INTERFACE-NAME-LENGTH
                        WS-INTERFACE-NAME.
    SET WS-STRGET TO TRUE.
    PERFORM CHECK-STATUS.

***********************************************************
* Interface(s) :
***********************************************************
    MOVE SPACES TO BANK-OPERATION.
***********************************************************
* Evaluate Interface(s) :
***********************************************************

    EVALUATE WS-INTERFACE-NAME
    WHEN 'IDL:BankDemo/Bank:1.0'

* Resolve the pointer reference to the operation information
    CALL "STRGET" USING OPERATION-NAME
         BANK-OPERATION-LENGTH
         BANK-OPERATION
    SET WS-STRGET TO TRUE
    PERFORM CHECK-STATUS
    WHEN 'IDL:BankDemo/Account:1.0'
    END-EVALUATE.

    COPY BANKD.
    GOBACK.

    DO-BANK-FIND-ACCOUNT.
    SET D-NO-USEREXCEPTION TO TRUE.
    CALL "COAGET" USING BANK-FIND-ACCOUNT-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.
```

```
* TODO:  Add your operation specific code her

    EVALUATE TRUE
    WHEN D-NO-USEREXCEPTION
    CALL "COAPUT" USING BANK-FIND-ACCOUNT-ARGS
    SET WS-COAPUT TO TRUE
    PERFORM CHECK-STATUS

    END-EVALUATE.

    DO-BANK-CREATE-ACCOUNT.
    SET D-NO-USEREXCEPTION TO TRUE.
    CALL "COAGET" USING BANK-CREATE-ACCOUNT-ARGS.
    SET WS-COAGET TO TRUE.
    PERFORM CHECK-STATUS.

* TODO:  Add your operation specific code here

     EVALUATE TRUE
     WHEN D-NO-USEREXCEPTION
     CALL "COAPUT" USING BANK-CREATE-ACCOUNT-ARGS
     SET WS-COAPUT TO TRUE
     PERFORM CHECK-STATUS
     END-EVALUATE.

     DO-BANK-SHUTDOWN-BANK.
     CALL "COAGET" USING BANK-SHUTDOWN-BANK-ARGS.
     SET WS-COAGET TO TRUE.
     PERFORM CHECK-STATUS.

* TODO:  Add your operation specific code here

     CALL "COAPUT" USING BANK-SHUTDOWN-BANK-ARGS.
     SET WS-COAPUT TO TRUE.
     PERFORM CHECK-STATUS.
***********************************************************
* Check Errors Copybook
***********************************************************
     COPY CHKERRS.
```

Mapping for Attributes

Overview

This section describes how IDL attributes are mapped to COBOL.

This subsection discusses the following topics:

- IDL-to-COBOL Mapping for Attributes
- Mapping Example

IDL-to-COBOL Mapping for Attributes

IDL attributes are mapped to COBOL declarations, with a `_GET_` and `_SET_` prefix. If an attribute is not read-only, two declarations are created for it (that is, one declaration with a `_GET_` prefix, and one declaration with a `_SET_` prefix). If an attribute is read-only, only one declaration is created for it, with a `_GET_` prefix.

Mapping Example

The example can be broken down as follows:

Consider the two attributes in the account interface in the bankdemo IDL:

```
//IDL
interface Account
{
     readonly attribute AccountId  account_id;
     readonly attribute CashAmount balance;
};
```

The preceding IDL maps to the following COBOL:

```
* COBOL
01 ACCOUNT-OPERATION               PICTURE X(41).
        88 ACCOUNT-GET-ACCOUNT-ID         VALUE
            "_get_account_id:IDL:BankDemo/Account:1.0".
        88 ACCOUNT-GET-BALANCE            VALUE
            "_get_balance:IDL:BankDemo/Account:1.0".
01 ACCOUNT-OPERATION-LENGTH        PICTURE 9(09) BINARY
                                          VALUE 41.
```

Mapping for User Exceptions

Overview

This section describes how IDL user exceptions are mapped to COBOL.

This subsection discusses the following topics:

- IDL-to-COBOL Mapping for User Exceptions
- Raising a User Exception
- Example of IDL-to-COBOL Mapping for Exceptions

IDL-to-COBOL Mapping for User Exceptions

An IDL exception maps to the following in COBOL:

- A level `01` group item that contains the definitions for all the user exceptions defined in the IDL. This group item is defined in COBOL as follows:

  ```
  01 BANK-USER-EXCEPTIONS.
  ```

  The group item contains the following level `03` items:

  - An `EXCEPTION-ID` string that contains a textual description of the exception.
  - A `D` data name that specifies the ordinal number of the current exception. Within this, each user exception has a level `88` data name generated with its corresponding ordinal value.
  - A `U` data name.
  - A data name for each user exception, which redefines `U`. Within each of these data names are level `05` items that are the COBOL-equivalent user exception definitions for each user exception, based on the standard IDL-to-COBOL mapping rules.

- A level `01` data name with an `EX-`*`FQN-userexceptionname`* format, which has a string literal that uniquely identifies the user exception. *`FQN`* stands for the Fully Qualified Name which is the name and the name of the enclosing interface and the enclosing module. For the bankdemo IDL which has no modules the *`FQN`* for `AccountNotFound` exception is bank and the corresponding `01` data name is `EX-BANK-ACCOUNTNOTFOUND`.
- A corresponding level 01 data name with an EX-FQN-userexceptionname-LENGTH format, which has a value specifying the length of the string literal.

Note: If `D` and `U` are used as IDL identifiers, they are treated as reserved words. This means that they are prefixed with `IDL-` in the generated COBOL. For example, the IDL identifier, `d`, maps to the COBOL identifier, `IDL-D`.

Raising a User Exception

Use the `COAERR` function to raise a user exception. Refer to the *COBOL Programmer's Guide and Reference* for more details.

Example of IDL-to-COBOL Mapping for Exceptions

The example can be broken down as follows:

1. Consider the exceptions in the bank and account interfaces in the `bankdemo` IDL:

```
 //IDL
interface Bank
{
   exception AccountAlreadyExists { AccountId account_id; };
   exception AccountNotFound      { AccountId account_id; };
};
interface Account
{
   exception InsufficientFunds {};
};
```

2. The preceding IDL maps to the following COBOL:

```
* COBOL
******************************************************
* User exception block
***********************************************************
01 EX-BANK-ACCOUNTALREADYEXISTS    PICTURE X(42)
                                          VALUE
            "IDL:BankDemo/Bank/AccountAlreadyExists:1.0".
01 EX-BANK-ACCOUNTALREADYEXI-8E46  PICTURE 9(09) BINARY
                                          VALUE 42.
01 EX-BANK-ACCOUNTNOTFOUND         PICTURE X(37)
                                          VALUE
            "IDL:BankDemo/Bank/AccountNotFound:1.0".
01 EX-BANK-ACCOUNTNOTFOUND-LENGTH  PICTURE 9(09) BINARY
                                          VALUE 37.
01 EX-ACCOUNT-INSUFFICIENTFUNDS    PICTURE X(42)
                                          VALUE
            "IDL:BankDemo/Account/InsufficientFunds:1.0".
01 EX-ACCOUNT-INSUFFICIENTFU-3EA4  PICTURE 9(09) BINARY
                                          VALUE 42.
01 BANK-USER-EXCEPTIONS.
   03 EXCEPTION-ID                   POINTER
                                          VALUE NULL.
   03 D                              PICTURE 9(10) BINARY
                                          VALUE 0.
         88 D-NO-USEREXCEPTION            VALUE 0.
         88 D-BANK-ACCOUNTALREADYEXISTS   VALUE 1.
         88 D-BANK-ACCOUNTNOTFOUND        VALUE 2.
         88 D-ACCOUNT-INSUFFICIENTFUNDS   VALUE 3.
   03 U                              PICTURE X(04)
                                         VALUE LOW-VALUES.
   03 EXCEPTION-BANK-ACCOUNTALR-71CB REDEFINES U.
     05 ACCOUNT-ID                    POINTER.
   03 EXCEPTION-BANK-ACCOUNTNOTFOUND REDEFINES U.
     05 ACCOUNT-ID                    POINTER.
   03 EXCEPTION-ACCOUNT-INSUFFI-DDCB REDEFINES U PICTURE
   X(04).
```

Generated PL/I Members

Overview

This section describes the various PL/I source code and include members that the Orbix IDL compiler generates.

This section discusses the following topics:

- Generated Members
- Member name restrictions

Generated Members

Table 5 provides an overview and description of the PL/I members that the Orbix IDL compiler generates, based on the IDL member name.

Table 5: *PL/I Members Generated by the Orbix IDL Compiler*

Member Name	Member Type	Compiler Argument Used to Generate	Description
*bank*I	Source code	-S	This is the server implementation source code member. It is only generated if you use the -S argument with the Orbix IDL compiler.
*bank*V	Source code	Generated by default	This is the server mainline source code member. It is generated by default unless you specify the -V argument with the Orbix IDL compiler.
*bank*D	Include member	Generated by default	This is the select include member. It selects the appropriate implementation function for the attribute or operation being called.
*bank*L	Include member	Generated by default	This is the alignment include member. It contains procedures to perform the PL/I alignment calculations on behalf of the PL/I runtime.
*bank*M	Include member	Generated by default	This is the main include member. It stores all the PL/I structures and declarations.

Table 5: *PL/I Members Generated by the Orbix IDL Compiler*

Member Name	Member Type	Compiler Argument Used to Generate	Description
*bank*T	Include member	Generated by default	This is the typedef include member. It stores the based identifier information (that is, the PL/I structure definitions for which no storage is allocated).
*bank*X	Include member	Generated by default	This is the runtime include member. It contains information for the PL/I runtime about the contents of each interface.

Member name restrictions

If the IDL member name exceeds six characters, the Orbix IDL compiler uses only the first six characters of the IDL member name when generating the source and include member names. This allows space for appending a one-character suffix to each generated member name, while allowing it to adhere to the seven-character maximum size limit for PL/I external procedure names, which are based by default on the generated member names.

PL/I API Reference Summary

Introduction

This section provides a summary of the API functions, in alphabetic order that appear in the code in IDL to PL/I Mapping. For details of all PL/I APIs refer to the *PL/I Programmer's Guide and Reference*.

Summary Listing

The following is a list of COBOL APIs that appear in the sample code produced by the IDL compiler for the bankdemo IDL.

```
// Allows a PL/I server to raise a user exception for an
// operation.

PODEXEC(in PTR object_reference,
        in CHAR(*) operation_name,
        inout PTR operation_buffer,
        inout PTR user_exception_buffer)
// Invokes an operation on the specified object.

PODGET(in PTR operation_buffer)
// Marshals in and inout arguments for an operation on the server
// side from an incoming request.

PODPUT(out PTR operation_buffer)
// Marshals return, out, and inout arguments for an operation on
// the server side from an incoming request.
STRSET(out PTR string_pointer,
       in CHAR(*) string,
       in FIXED BIN(31) string_length)
// Creates an unbounded string from a CHAR(n) data item.
```

IDL to PL/I Mapping

Overview

This section illustrates how IDL is mapped by the Orbix IDL compiler to PL/I code. It uses the bankdemo IDL to illustrate how IDL operations, attributes, and user defined exceptions are mapped to PL/I. For a complete list of the IDL to PL/I mappings refer to the *PL/I Programmer's Guide and Reference*.

In This Section

This section discusses the following topics:

Mapping for Operations

Overview

This section describes how IDL operations are mapped to PL/I. The code samples are based on the bank interface in the IDL Example.

This subsection discusses the following topics:

- The Bankdemo IDL bank Interface
- IDL-to-PL/I Mapping for Operations

The Bankdemo IDL bank Interface

The bank interface:

```
//IDL
interface Bank
{
   exception AccountAlreadyExists { AccountId account_id; };
   exception AccountNotFound      { AccountId account_id; };

   Account
   find_account(
       in AccountId account_id
   ) raises(AccountNotFound);

    Account
    create_account(
        in AccountId account_id,
        in CashAmount initial_balance
    ) raises (AccountAlreadyExists);

    void
    shutdown_bank();
};
```

IDL-to-PL/I Mapping for Operations

An IDL operation maps to a number of statements in PL/I as follows:

1. A structure is created for each operation. This structure is declared in the `bankT` include member as a based structure and contains a list of the parameters and the return type of the operation. An associated declaration, which uses this based structure, is declared in the `bankM` include member. Memory is allocated only for non-dynamic types,

such as bounded strings and longs. The top-level identifier (that is, at `dcl 1` level) for each operation declaration is suffixed with `_type` in the `bankT` include member, and with `_args` in the `bankM` include member.

2. A declaration is generated in the `bankT` include member for every IDL operation. The declaration contains the fully qualified operation name followed by a space, which is used when calling `PODEXEC` to invoke that operation on a server.

   Based on the bank interface IDL, the following operation structures are generated in the `bankT` include member:

Example 3: *The bankT Include Member for the bank Interface (Sheet 1 of 2)*

```
/*-------------------------------------------------------*/
/* Interface:                                            */
/*     BankDemo/Bank                                     */
/*                                                       */
/* Mapped name:                                          */
/*     Bank                                              */
/*                                                       */
/* Inherits interfaces:                                  */
/*     (none)                                            */
/*-------------------------------------------------------*/
/*-------------------------------------------------------*/
/* Operation:       find_account                         */
/* Mapped name:     Bank_find_account                    */
/* Arguments:       <in> BankDemo/AccountId account_id   */
/* Returns:         BankDemo/Account                     */
/* User Exceptions: BankDemo/Bank/AccountNotFound        */
/*-------------------------------------------------------*/
dcl 1 Bank_find_account_type based,
      3 account_id             ptr            init(sysnull()),
      3 result                 ptr            init(sysnull());
/*-------------------------------------------------------*/
/* Operation:      create_account                        */
/* Mapped name:    Bank_create_account                   */
/* Arguments:      <in> BankDemo/AccountId account_id    */
/*                 <in> BankDemo/CashAmount initial_balance*/
/* Returns:         BankDemo/Account                     */
/* User Exceptions: BankDemo/Bank/AccountAlreadyExists   */
/*-------------------------------------------------------*/
```

Example 3: *The bankT Include Member for the bank Interface (Sheet 2 of 2)*

```
 dcl 1 Bank_create_account_type based,
       3 account_id             ptr            init(sysnull()),
       3 initial_balance        float dec(6)   init(0.0),
       3 result                 ptr            init(sysnull());
/*--------------------------------------------------------*/
/* Operation:       shutdown_bank                         */
/* Mapped name:     Bank_shutdown_bank                    */
/* Arguments:       None                                  */
/* Returns:         void                                  */
/* User Exceptions: none                                  */
/*--------------------------------------------------------*/
dcl 1 Bank_shutdown_bank_type based,
      3 filler_0002               char(01);
```

Based on the bank interface IDL, the following operation structures are generated in the `bankM` include member:

Example 4: *The bankM Include Member for the bank Interface*

```
/*PL/I */
%include BANKT;

/*--------------------------------------------------------------------*/
/* Interface:                                                         */
/*     BankDemo/Bank                                                  */
/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/
/* Operation:       find_account                                      */
/*--------------------------------------------------------------------*/
dcl 1 Bank_find_account_args aligned like Bank_find_account_type;

/*--------------------------------------------------------------------*/
/* Operation:       create_account                                    */
/*--------------------------------------------------------------------*/
dcl 1 Bank_create_account_args aligned like Bank_create_account_type;

/*--------------------------------------------------------------------*/
/* Operation:       shutdown_bank                                     */
/*--------------------------------------------------------------------*/
dcl 1 Bank_shutdown_bank_args aligned like Bank_shutdown_bank_type;
```

3. The operation declaration is also used in the `bankD` include member. It is used within the select clause, which is used by the server program to call the appropriate operation procedure described next in point 4.
4. The following select statement is also generated in the `bankD` include member for the `bank` interface:

Example 5: *PL/I bankD Include Member select Statement Code (Sheet 1 of 2)*

```
/*PL/I*/
select(operation);
   when (Bank_find_account) do;
     BANK_user_exceptions.d=0;
     call podget(addr(Bank_find_account_args));
     if check_errors('podget') ^= completion_status_yes then
    return;

     call proc_Bank_find_account(addr(Bank_find_account_args));

     if BANK_user_exceptions.d=0 then
       do;
         call podput(addr(Bank_find_account_args));
         if check_errors('podput') ^= completion_status_yes then
    return;
```

Example 5: *PL/I bankD Include Member select Statement Code (Sheet 2 of 2)*

```
    end;
 end;
when (Bank_create_account) do;
         BANK_user_exceptions.d=0;
         call podget(addr(Bank_create_account_args));
        if check_errors('podget') ^= completion_status_yes then
   return;

        call
   proc_Bank_create_account(addr(Bank_create_account_args));

         if BANK_user_exceptions.d=0 then
           do;
             call podput(addr(Bank_create_account_args));
             if check_errors('podput') ^= completion_status_yes
   then
                      return;
           end;
       end;
   end;

 when (Bank_shutdown_bank) do;
         call podget(addr(Bank_shutdown_bank_args));
        if check_errors('podget') ^= completion_status_yes then
   return;

         call
   proc_Bank_shutdown_bank(addr(Bank_shutdown_bank_args));

         call podput(addr(Bank_shutdown_bank_args));
        if check_errors('podput') ^= completion_status_yes then
   return;
       end;
       otherwise do;
         put skip list('ERROR! Operation :',operation);
         put skip list('is not defined in:',interface);
         return;
       end;
 end;
```

5. When you specify the -S argument with the Orbix IDL compiler, an empty server procedure is generated in the `bankI` source member for each IDL operation. (You must specify the -S argument, to generate these operation procedures.). The following skeleton procedures are generated in the `bankI` member:

Example 6: *PL/I bankI Source Member for bank Interface (Sheet 1 of 3)*

```
BANKI: PROC;

/* The following line enables the runtime to call this procedure     */
DISPTCH: ENTRY;

dcl (addr,length,low,sysnull)     builtin;

%include CORBA;
%include CHKERRS;
%include BANKM;
%include DISPINIT;

/* =================== Start of global user code =================== */
/* ==================== End of global user code ==================== */

/*------------------------------------------------------------------*/
/*                                                                  */
/*  Dispatcher : select(operation)                                  */
/*                                                                  */
/*------------------------------------------------------------------*/
%include BANKD;

/*------------------------------------------------------------------*/
/* Interface:                                                       */
/*     BankDemo/Bank                                                */
/*                                                                  */
/* Mapped name:                                                     */
/*     Bank                                                         */
/*                                                                  */
/* Inherits interfaces:                                             */
/*     (none)                                                       */
/*------------------------------------------------------------------*/
```

Example 6: *PL/I bankI Source Member for bank Interface (Sheet 2 of 3)*

```
/*-------------------------------------------------------------------*/
/* Operation:       find_account                                     */
/* Mapped name:     Bank_find_account                                */
/* Arguments:       <in> BankDemo/AccountId account_id               */
/* Returns:         BankDemo/Account                                 */
/* User Exceptions: BankDemo/Bank/AccountNotFound                    */
/*-------------------------------------------------------------------*/
proc_Bank_find_account: PROC(p_args);

dcl p_args                        ptr;
dcl 1 args                        aligned based(p_args)
                                  like Bank_find_account_type;

/* =============== Start of operation specific code ================ */
/* ================ End of operation specific code ================= */

END proc_Bank_find_account;

/*-------------------------------------------------------------------*/
/* Operation:       create_account                                   */
/* Mapped name:     Bank_create_account                              */
/* Arguments:       <in> BankDemo/AccountId account_id               */
/*                  <in> BankDemo/CashAmount initial_balance         */
/* Returns:         BankDemo/Account                                 */
/* User Exceptions: BankDemo/Bank/AccountAlreadyExists               */
/*-------------------------------------------------------------------*/
proc_Bank_create_account: PROC(p_args);

dcl p_args                        ptr;
dcl 1 args                        aligned based(p_args)
                                  like Bank_create_account_type;
/* =============== Start of operation specific code ================ */
/*,================,End of operation specific code ================= */

END proc_Bank_create_account;
```

Example 6: *PL/I bankI Source Member for bank Interface (Sheet 3 of 3)*

```
/*-------------------------------------------------------------------*/
/* Operation:       shutdown_bank                                    */
/* Mapped name:     Bank_shutdown_bank                               */
/* Arguments:       None                                             */
/* Returns:         void                                             */
/* User Exceptions: none                                             */
/*-------------------------------------------------------------------*/
proc_Bank_shutdown_bank: PROC(p_args);

dcl p_args                        ptr;
dcl 1 args                        aligned based(p_args)
                                  like Bank_shutdown_bank_type;

/* =============== Start of operation specific code ================ */
/* ================ End of operation specific code ================= */

END proc_Bank_shutdown_bank;
```

Mapping for Attributes

Overview

This section describes how IDL attributes are mapped to PL/I.

This subsection discusses the following topics:

- IDL-to-PL/I Mapping for Attributes
- Example Mapping

IDL-to-PL/I Mapping for Attributes

IDL attributes are mapped to PL/I declarations, with a `_get_` and `_set_` prefix. If an attribute is not read-only, two declarations are created for it (that is, one declaration with a `_get_` prefix, and one declaration with a `_set_` prefix). If an attribute is read-only, only one declaration is created for it, with a `_get_` prefix.

Example Mapping

The example can be broken down as follows:

1. Consider the following IDL:

```
 //IDL
interface Account
  {
      readonly attribute AccountId  account_id;
      readonly attribute CashAmount balance;
  }
```

2. The preceding IDL maps to the following operation list in the `bankT` include member:

```
 /*PL/I */
/*---------------------------------------------------------*/
/* Operation List section                                */
/* Contains a list of the interface's operations and
/* attributes.*/
/*---------------------------------------------------------*/
dcl Account_get_account_id  char(41) init('_get_account_id:
 IDL:BankDemo/Account:1.0 ');
dcl Account_get_balance     char(38) init('_get_balance:IDL
 :BankDemo/Account:1.0 ');
```

3. The following operation procedure names are generated in the `bankI` member:

Example 7: *PL/I Operation Procedure Code in the bankI Member*

```
/*PL/I */
/*-----------------------------------------------------------*/
/* Attribute:    account_id (get)                            */
/* Mapped name:  Account_account_id                          */
/* Type:         BankDemo/AccountId (readonly)               */
/*-----------------------------------------------------------*/
 proc_Account_get_account_id: PROC(p_args);

 dcl p_args                        ptr;
 dcl 1 args                        aligned based(p_args)
                                  like Account_account_id_type;

/* ========== Start of operation specific code ========= */
/* ========= End of operation specific code ============ */

 END proc_Account_get_account_id;
/*-----------------------------------------------------------*/
/* Attribute:    balance (get)                               */
/* Mapped name:  Account_balance                             */
/* Type:         BankDemo/CashAmount (readonly)              */
/*-----------------------------------------------------------*/
 proc_Account_get_balance: PROC(p_args);

 dcl p_args                        ptr;
 dcl 1 args                        aligned based(p_args)
                                   like Account_balance_type;

/* ========== Start of operation specific code ==========*/
/* ============= End of operation specific code =========*/
 END proc_Account_get_balance;
```

Mapping for User Exceptions

Overview

This section describes how IDL exceptions are mapped to PL/I.

This section discusses the following topics:

- IDL-to-PL/I Mapping for Exceptions
- Raising a User Exception
- Example of Raising a User Exception
- Example of Testing a User Exception

IDL-to-PL/I Mapping for Exceptions

1. An IDL exception type maps to a PL/I structure and a character data item with a value that uniquely identifies the exception. Consider the bank interface:

```
 //IDL
interface Bank
{
   exception AccountAlreadyExists { AccountId account_id; };
   exception AccountNotFound      { AccountId account_id; };
};
interface Account
{
   exception InsufficientFunds {};
};
```

2. Based on the preceding IDL, the Orbix IDL compiler generates the following code in the `bankT` include member:

```
/*PL/I */
/*-------------------------------------------------------*/
/* User Exception: BankDemo/Bank/AccountAlreadyExists    */
/*-------------------------------------------------------*/
 dcl 1 Bank_AccountAlreadyExists_type based,
       3 account_id             ptr        init(sysnull());
/*-------------------------------------------------------*/
/* User Exception: BankDemo/Bank/AccountNotFound         */
/*-------------------------------------------------------*/
 dcl 1 Bank_AccountNotFound_type based,
       3 account_id             ptr        init(sysnull());
/*-------------------------------------------------------*/
/* User Exception: BankDemo/Account/InsufficientFunds    */
/*-------------------------------------------------------*/
 dcl 1 Account_InsufficientFunds_type based,
       3 filler_0003               char(01);
```

3. The Orbix IDL compiler generates the following code in the `bankM` include member:

```
/*PL/I                                                      */
/*----------------------------------------------------------*/
/* Defined User Exceptions                                  */
/*----------------------------------------------------------*/
 dcl 1 BANK_user_exceptions          static ext,
       3 exception_id                ptr,
       3 d                           fixed bin(31)  init(0),
       3 u                           area(04);

 dcl 1 Bank_AccountAlreadyExists_exc_d
                                     fixed bin(31)  init(1);
 dcl 1 Bank_AccountNotFound_exc_d    fixed bin(31)  init(2);
 dcl 1 Account_InsufficientFunds_exc_d
                                     fixed bin(31)  init(3);

 dcl 1 Bank_AccountAlreadyExists_exc
                        based(addr(BANK_user_exceptions.u)),
       3 account_id                  ptr
   init(sysnull());

 dcl 1 Bank_AccountNotFound_exc
   based(addr(BANK_user_exceptions.u)),
       3 account_id                  ptr
   init(sysnull());

 dcl 1 Account_InsufficientFunds_exc
                        based(addr(BANK_user_exceptions.u)),
       3 filler_0001                 char(01);
```

Raising a User Exception

The server can raise these user exceptions by performing the following sequence of actions:

Stage	Description
1	It calls `STRSET` to set the `exception_id` identifier of the user exception structure with the appropriate exception identifier defined in the `bankT` include member. The exception identifier in this case is suffixed with `_exid`.

Stage	Description
2	It sets the d discriminator with the appropriate exception identifier defined in the `bankM` include member. The exception identifier in this case is suffixed with _d.
3	It fills in the exception branch block associated with the exception.
4	It calls PODERR with the address of the user exception structure.

Example of Raising a User Exception

The following code shows how to raise the `AccountNotFound` user exception defined in the preceding example:

```
/*PL/I */
/* Server implementation code  */
if name='' then
    do;
        strset(bank_user_exceptions.exception_id,
               SimpleObject_AccountNotFound_exid,
               length(SimpleObject_AccountNotFound_exid));
        bank_user_exceptions.d=bank_bad_exc_d;
        call poderr(addr(bank_user_exceptions));
    end;
```

Example of Testing a User Exception

To test for the `AccountNotFound` user exception, the client side tests the discriminator value of the user exception structure after calling PODEXEC on the server function, which is able to raise a user exception. For example, the following code shows how the client can test whether the server set an exception after the call to:

Example 8: *PL/I Code for Testing User Exceptions (Sheet 1 of 2)*

```
/*PL/I */
/* Call podexec to perform operation addName.             */
/* Note the user exception block in the fourth parameter. */
call podexec(bank_obj,
             bank_find_account,
             bank_find_account_account_id,
             addr(BANK_user_exceptions));
```

Example 8: *PL/I Code for Testing User Exceptions (Sheet 2 of 2)*

```
if BANK_user_exceptions.d ^= 0 then
   do;
      /* a user exception has been thrown */
      put skip list('Operation find_account threw a user
   exception!');
     put skip list(' Discriminator: ',EXCEPT_AccountNotFound.d);

      select(BANK_user_exceptions.d);
         when(bank_AccountNotFound_exc_d)
            do;
              put list('Exception thrown: AccountNotFound_exc');
               put skip
   list('value1:',bank_AccountNotFound_exc.value1);
               put skip
   list('reason:',bank_AccountNotFound_exc.reason);
            end;
              otherwise
            put list('Unrecognized exception!');
      end;
   end;
else  /* no exception has been thrown  */
   do;
      put skip list('Operation find_account completed
   successfully');
   end;
```

Part 2

IDL Design and CORBA Object Location

In this part

This part contains the following chapters:

CHAPTER 5

IDL Design

Before reading this chapter you should already be familiar with the IDL concepts as described in either the COBOL or PL/I Programmer's Guides as well as the section on IDL in part one of this document. This chapter explores design issues for writing efficient IDL.

In this chapter

This chapter discusses the following topics:

IDL Constructs

Overview

IDL in syntax and semantics is similar to Java interface or C++ type definitions. However, because of the need to cater for distribution and to remain language independent, IDL constructs sometimes differ in a number of ways from their C++ and Java counterpart.

This section discusses the following topics:

- Storing IDL definitions
- Storing IDL definitions in z/OS data sets
- IDL keywords
- Layout and Indentation styles
- The IDL preprocessor
- Order of definitions in a source file

Storing IDL definitions

The CORBA specification states that IDL definitions must be placed in source files with a `.idl` extension. For file systems that have case-sensitive names, you must use a lowercase file name extension (`.IDL` is illegal with such file systems). For file systems that are case-insensitive, such as Windows, `.IDL` is legal.

Storing IDL definitions in z/OS data sets

In regard to z/OS data sets, there is no requirement that the lowest qualifier should end in `.idl`; however, it is generally used (by Micro Focus) for the purpose of clarity.

IDL keywords

Keywords are in lowercase, except for the Object keyword and the `FALSE` and `TRUE` Boolean constants, which must be capitalized as shown.

Layout and Indentation styles

You can use any layout and indentation style you prefer. IDL is a free-form language that does not place meaning on white space or indentation. Comments can use either `//` (that is, C++ style) or `/*…*/` (that is, C style) syntax. For example:

```
//IDL
//Example of a comment extending to the end of this line
```

```
/*
* Example of a multi-line comment
* that runs on to the next line
*/
```

The IDL preprocessor

IDL files are preprocessed by the preprocessor which behaves exactly the same as a C++ preprocessor. This means that you can use the usual preprocessor directives such as `#define`, `#ifdef`, macro definitions, and so on. The most common use of the preprocessor is inclusion of another specification using `#include`.

Order of definitions in a source file

Definitions in a source file can appear in any order, with the usual caveat that you must define or declare things before using them

IDL Interface Semantics

Overview

This section discusses the semantics of passing IDL interfaces across a network.

This section discusses the following topics:

- Passing IDL interface instances
- Semantics of passing an object reference

Passing IDL interface instances

IDL interfaces are types and interface instances can be passed as parameters. In the following example, the `eat` operation is passed a parameter of the `Haystack` type. Passing an interface as a parameter, passes a reference to a particular CORBA object to the operation. The called operation has full access to the passed object via the object's reference. In this example, the implementation of `eat` can invoke the `destroy` operation on the `haystack`. This results in a message to the `haystack` object.

```
//IDL
  interface Haystack {
  void destroy(); // Ends life cycle of haystack
// ...
};

  interface Camel {
  void eat(in Haystack hs);
// ...
};
```

Semantics of passing an object reference

Conceptually, passing an object reference is like passing a pointer to the object. However, an object reference can point across a network to an object in a different address space (which can be on a different host). When a client invokes an operation via a reference, it makes an RPC call that is delivered to the object implementation.

Note: References are strongly typed. The `eat` operation expects a reference of the `Haystack` type and the ORB ensures that it impossible to pass a reference of the wrong type. With statically typed languages, such as C++ and Java, type checking happens at compile time, so if you attempt to pass a reference to an object that does not support the `Haystack` interface, your code will not compile. For languages such as Smalltalk, which use dynamic type checking, the type mismatch is detected at runtime instead.

Interface Support in IMS or CICS

For environments such as IMS and CICS, where state is typically not maintained across multiple server-side operation invocations, this pattern is not suitable and is therefore not supported. This scenario would involve the passing of an object reference from a factory type service implemented in an IMS or CICS Orbix server (written in COBOL or PL/I) that is to be invoked on by the client in a subsequent IDL operation.

If the factory resides in some other CORBA environment, the IMS or CICS Orbix application can invoke on it to obtain and use an object reference from a client-side perspective. For example, this is a common pattern used when interacting with the CORBA Naming Service.

IDL Identifiers and Repository IDs

Overview

This section discusses the basics of IDL identifiers and repository IDs

In this section

This section discusses the following topics:

IDL Identifiers

Overview

IDL identifiers must begin with a letter or underscore, followed by any number of letters, digits, and underscores. Because CORBA 2.3 supports an IDL identifier with a leading underscore, it avoids clashes with IDL keywords. For example, in the `CosLifecycle` specification, the old `factory` interface is now the `_factory` interface, to avoid clashing with the new reserved IDL keyword, `factory`. OMG specifications adhere to a naming convention that uses mixed case identifiers with a leading uppercase letter (or underscore) for module, interface, and type names, and uses lowercase identifiers with an underscore separator for operation, attribute, parameter, and member names. You might wish to adopt the same convention.

This section discusses the following topics:

- Capitalization
- Programming keywords

Capitalization

IDL enforces consistent capitalization. Within the same scope, identifiers that differ only in case are not allowed. This means that if you define the identifier `foo` in a scope, all other capitalizations of the same identifier, such as `FOO` and `fOo`, are "used up" within that scope. When you have introduced `foo` into a scope, you must continue to refer to it with that capitalization; otherwise, the compiler complains. IDL uses these capitalization rules to make it easier to map identifiers into different target languages. The capitalization rules ensure that identifiers can be mapped easily into languages that consider case to be significant (such as C++ and Java), as well as into languages that ignore case (such as Pascal).

Programming keywords

It is good practice to avoid IDL identifiers that are keywords in some programming language. For example, `PROCEDURE`, `EVALUATE`, `USING`, `CALL`, `BINARY`, `DO` and `DECLARE` are legal IDL identifiers, but are also keywords in COBOL or PL/I.

If a reserved COBOL or PL/I keyword is used as an IDL interface or module name, the Orbix IDL compiler prefixes it with `IDL-`. This mechanism avoids a keyword conflict in the target language but also results in less-readable code; it is better to avoid keywords in the first place.

Repository IDs

Overview

This section illustrates how the IDL compiler assigns repository IDs to each IDL identifier it parses.

This section discusses the following topics:

- Repository ID structure
- Repository ID example
- Repository IDs and ORBs
- Avoiding repository ID clashes
- #pragma prefix directives

Repository ID structure

Every IDL name is assigned a unique Repository ID, which is a string. IDL allows for the unique identification of all components in your IDL files. All of the following are assigned Repository Identifiers: modules, interfaces, constants, type definitions, exceptions, attributes, operations.

The Repository ID consists of 3 parts separated by colons:

```
format name: identifier: version.
```

The default `format name` is `IDL`, although others are valid. The `identifier` is typically the scoped name for the IDL object—scoped by the containing module, interface, and so on. The `version` is made up of a major and minor version number.

Repository ID example

The IDL compiler automatically assigns a repository ID to each IDL identifier. The repository ID is formed by concatenating the names of the nested scopes with slashes and adding an `IDL:` prefix and a `:1.0` suffix: The following IDL illustrates this process.

```
//IDL
module M {                      // IDL:M:1.0
      interface I {             // IDL:M/I:1.0
          typedef long T;       // IDL:M/I/T:1.0
          T op();               // IDL:M/I/op:1.0
    };
};
```

Repository IDs and ORBs

The repository IDs formed this way are the ORBs only handle to the meaning of a type. In particular, if in any two IDL files the same name is used to describe different IDL entities a name clash results.

Avoiding repository ID clashes

You can avoid repository ID clashes by enclosing all of your IDL in a module with a suitably unique name. For example:

```
//IDL
module AcmePtyLtd {
// All of the definitions for Acme's software here...
};
```

This approach has the drawback that it leads to very long identifiers at the implementation language level. All identifiers have to be qualified with AcmePtyLtd, which is awkward. In addition, large modules such as this can lead to version control problems and make it difficult to decouple developers working in different teams.

#pragma prefix directives

It is a good idea to add a #pragma prefix directive to all your IDL definitions. You should use a prefix that you own in some sense, and that therefore doesn't clash with other developers. An Internet domain name makes for a good prefix because it is registered with a naming authority.

A #pragma prefix directive affects all definitions that are at the same scope or in nested scopes following the pragma; the directive qualifies the repository IDs with the specified prefix. This mechanism guarantees uniqueness of the generated IDs for a domain (barring malicious reuse of the same prefix). The biggest advantage of the pragma over using a module name for the same purpose is that the effects of the pragma are invisible at the language mapping level. In other words, you do not end up with long and cumbersome identifiers or an additional level of namespace or package nesting.

If two IDL definitions use the same qualified name for different types but each have a different prefix, the ORB's type system stays same because, internally, types are identified by their repository IDs. This means that identical type names for different types can coexist in the same CORBA domain (assuming they have different prefixes). Note that it is still impossible to use both these types from within the same program because, at the language mapping level, the prefix is not visible and the two types become indistinguishable. However, different programs can each use their respective version of the identically named types without problems.

Note: CORBA also defines a version pragma, but as its semantics are currently undefined, its use can cause interoperability problems. Use of the version pragma is not recommended.

IDL Versioning

Overview

One area of concern in CORBA systems is the versioning of IDL interfaces. Given that most systems change over time, changes to IDL must be handled smoothly.

In this section

This section discusses the following topics:

Working with more than One Version of IDL

Overview

IDL can change either by having additions made to it, or by being modified. When you release (and deploy) a CORBA system, you inevitably continue to work on it, and at some point later have a new version to be deployed. Typically, you have changed the IDL that the system uses. These IDL changes might be due to additional requirements imposed on the system, or design errors that have appeared as the system has been deployed. In any case, accompanying your IDL changes are new client and server applications that have been modified to use the new interface.

This subsection discusses the following topics:

- Additions to IDL
- Problems with additional IDL
- Modifications to IDL
- Problems with modified IDL

Additions to IDL

If new interfaces, methods, data structures, or other components are simply added to the IDL (but no existing interfaces are changed) then old clients can simply use the new servers. This is due to the way that requests are marshaled in IIOP. Operations (method or attribute invocations on objects) are identified by the operation name only, so new attributes or methods added to an interface has no effect on calls to existing methods.

Problems with additional IDL

If you are very careful, you can run new servers without affecting existing old clients—If all your IDL changes are additive, and do not involve changes to any interface component currently in use, and if there are no changes to the semantics of the calls. For instance, if you add a new method, `login()` to some interface, and update the server so that a client is required to login before it can invoke any other method. In this case, even though existing client programs are required to logon by the new server but won't be able to do so until they are updated, so they are not be able to do any work. Here, application-level logic is preventing the client from working, even though the changes to the IDL are strictly additive.

Modifications to IDL

If the IDL is changed, then existing clients are not able to interact with the modified interfaces. Consider the following fragments of IDL, showing modifications:

```
//IDL
// Original IDL file
interface Broker {
Account newAccount(in string name);
};
```

```
//IDL
// Updated IDL file
interface Broker {
Account newAccount(in string name, in Money initialDeposit);
};
```

Problems with modified IDL

In this situation, old clients are not be able to interact with the new server. The server is unable to properly unmarshal the request, since it is missing the expected `initialDeposit` argument. The server either interprets the memory near the request as its missing parameter, or throw an exception.

In the cases where the server accesses memory near the request, looking for its expected additional parameters, bogus values, unpredictable behavior, and process terminations occur.

If additional parameters are sent, the server program generally ignores them, but might exhibit unexpected or unpredictable behavior as well. You must not send requests to a server (or replies to a client) with an incorrect set of parameters.

CORBA requests are sent in a very efficient format, without any unnecessary type information. In particular, no type code information is sent along with the requests. This protocol, specified by the OMG, increases the speed and efficiency of processing requests. However, it does prevent a server from performing type-checking. The client simply sends a sequence of bytes to the server, which assumes that it knows the proper order and structure of the data.

In the case where a client inadvertently sends an incorrect set of parameters, or an incorrect version of a structure, the server is unable to determine this, and simply attempts to extract the correct parameters from the request object. The results of this behavior are undefined. In general, mismatched clients and servers result in unexpected behavior, and program termination.

What you need is to have two versions of the broker object available—one supporting the original interface, and one supporting the new interface.

Distinguishing IDL Versions

Overview

The advantages and disadvantages of distinguishing IDL versions in the following ways are discussed in this subsection.

This subsection discusses the following topics:

- Distinguishing IDL versions at the application level
- Distinguishing IDL versions at the ORB level
- Distinguishing IDL versions using the module keyword

Distinguishing IDL versions at the application level

This approach keeps the IDL elements named the same, and only changes their contents. This approach is very confusing, since both of these IDL elements map to the same programming language elements. The ORB cannot distinguish between them, so the application programmer has to. This approach is not recommended, since it is so likely to result in confusion.

Distinguishing IDL versions at the ORB level

A better approach is to use different IDL elements for different versions of application elements. One approach is to modify the original `Customer` interface and rename it `Customer2`. The ORB treats them as distinct elements, so that there can be no confusing the two. They are not related by inheritance, or in any other way, so you cannot mistakenly treat a `Customer2` as a `Customer`.

Distinguishing IDL versions using the module keyword

The recommended approach is through the IDL module keyword. This allows you to version all enclosed IDL elements with only a one-line change to your IDL file. This also gives you a coarse, freshening granularity, so that each and every IDL element does not have to be individually freshened. In COBOL and PL/I, modules map to 01 levels 1 levels respectively.

CORBA Object Granularity

Overview

Object granularity refers to the number of CORBA objects in an application: the more objects the finer the granularity. This section discusses some of the issues concerning object granularity when designing CORBA applications.

This section discusses the following topics:

The consequences of using CORBA object

As discussed earlier, CORBA objects are relatively expensive to send, because they require the construction of a proxy on the receiving side. In addition, When a client has an object reference, retrieving any of the object's data requires additional remote calls.

Clients and object references

Sometimes, clients do not really need object references, and the use cases could more efficiently be satisfied by the use of data structures instead. This is the area of object granularity, where you need to consider the size of your CORBA objects, based on the amount of data and functionality they contain. A system with fine-grained objects tends to have many 'small' objects, which means that clients typically hold proxies to many objects, and make many remote invocations. Compare this with a coarse-grained system, which has fewer CORBA objects, each of which manages a relatively large amount of data, functionality, or both.

Search-and-select scenario

Examine the use case for the classic search-and-select-one-customer situation, as follows:

- User enters a last name.
- Application searches customers by last name.
- Application displays summary information (name/addresses/account number) of all matching customers.
- User chooses the customer to work with.

Consider an OO approach to this.

Search-and-select use case (first try)

Search-and-select use case (first try) shows the first try at the IDL for the system.

```
//IDL
// IDL—First try
interface Customer {
string getFirstName();
string getLastName();
string getAddress();
string getCity();
string getState();
string getZip();
string getPhoneNumber();
// add some interesting & realistic methods
Date getDateOfMostRecentPurchase();
};
```

All customer data and behavior is encapsulated in the Customer interface. This is object oriented, but has the downside of performing poorly in the use case. The search-and-select requires (at least) `N+1` remote calls to display the information on the `N` matching customers (depending on the IDL, this approach could actually require `N*M` calls to retrieve each of the `M` attributes from the `N` matching customers).

Search-and-select use case (second try)

The code shown below is optimized for the use case—you can retrieve matching customer information in one remote invocation, because you receive a sequence of `CustomerInfo` data structures as output.

```
//IDL
// IDL—Second try
typedef struct CustomerInfo {
string firstName;
string lastName;
string address;
string city;
string state;
string zip;
string phoneNumber;
Date dateOfBirth;
string creditCardNumber;
};
interface Customer {
CustomerInfo getCustomerInfo(void);
void setCustomerInfo(in CustomerInfo newInfo);
// other set and get methods as before...
};
```

Note: You are still encapsulating the behavior in a customer object. This balanced approach treats things as objects when it makes sense, and as data structures when it makes sense.

Object granularity summary

IDL generally has a mixture of data structures and objects. Often, the same data can be manipulated as both structures and as objects. This is because the IDL has different facets, intended for use by different use cases. That is, it is commonly not strictly segregated to one type. Very often, the same data is accessible via multiple structures as well as via an object. This often brings up the question of why even bother with representing business objects as CORBA objects at all; why not just represent everything as data structures. This is discussed next.

Manager objects and data structures

Can persistent CORBA objects be replaced by manager-style objects and data structures? The short answer is yes, but you don't want to, because:

- This is not an object oriented approach. This means that client and server programmers have to be able to work with this remote-procedure-call style interface from within their OO programming language. Also, this approach no longer relies on the ORB for object management features, but instead burdens application programmers with this effort. This approach is no longer location-transparent, which means that client programmers have to associate ObjectIds with their server-side management objects.
- The CORBA services are focused around object instances, not around ObjectIds. Systems relying solely on manager-style objects are not able to make use of services such as naming, transactions, or security.

IDL Data Types and Performance

Overview

There are three general ways that IDL design affects system performance: number of remote invocations, the type of data sent or returned, and the amount of data sent or returned.

Clearly, the number of remote invocations has a large impact on the performance of a system. Each remote invocation typically causes the invoking process to block until a reply is received. Each remote call involves formatting a message buffer, sending across the network, and waiting for a reply, plus waiting for any business processing to occur. It is important to understand this, because it is too easy to treat remote invocations as if they are the same as local invocations—syntactically, this is true, but performance-wise it is not true.

In this section

This section discusses the following topics:

Type of Data Sent

Overview

This subsection discusses the marshalling cost for IDL data types and IORs.

This subsection discusses the following topics:

- Cost of marshalling IDL data types
- Cost of marshalling IORs

Cost of marshalling IDL data types

Different IDL data types have different marshalling and unmarshalling costs. Every time a message is sent (whether a request or a reply), the data has to be copied from variables into a buffer by the sender, and extracted from the buffer into variables by the receiver. Because IDL data types map to different programming language constructs, they have different costs associated with them. For instance, an IDL short is relatively fast to marshal and unmarshal in Java, since it is small and of fixed size, and maps to a native data type. An IDL string, on the other hand, is more expensive, since it is of variable length, and maps to an instance of the String class.

Figure 11 shows the relative cost for sending or receiving IDL data types. Keep in mind that for the container types (such as an `any` or a `struct`), the timing shown only includes the cost of the container, and not that of the contained data items.

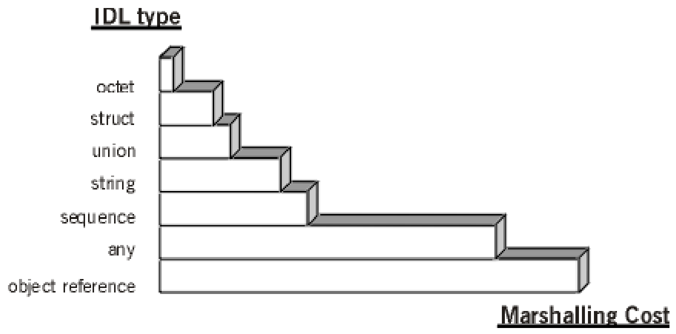

Figure 11: *Relative Marshalling Cost of Type of IDL Data Sent*

Cost of marshalling IORs

Object references are somewhat special for several reasons:

- They are variable length; the size of an object reference depends on the length of the interface, the size of the ORB-specific object key, and the number of additional profiles associated with the IOR.
- Unmarshalling an IOR within a receiving process involves more than simply extracting data from a buffer. A proxy object must be instantiated and initialized, which generally involves some non-trivial interaction with the ORB runtime.

As such, object references are the most expensive type of object to transfer.

Note: Returning an IOR usually implies that it will be used, which results in additional remote calls. In this regard, if the use cases are such that IORs are typically immediately used to retrieve state from the object, they are doubly inefficient.

Amount of Data Sent

Overview

The amount of data sent affects a system's performance—the more data being sent, the longer it takes.

This subsection discusses the following topics:

- Measuring throughput performance
- Sending raw data
- Iterators

Measuring throughput performance

The throughput graph shown in Figure 12 is non-linear. If your system has a large amount of data to transmit, it is important to send it in chunks that are large enough, but not too large. Exact measurements are very platform, product, and environment-specific, which is why they are not shown here. The key point is that if you expect to send relatively large amounts of data (anything over 100K), you should measure the performance of your ORB in your environment with varying chunk sizes, to determine where performance begins to degrade.

If your expected amount of data is close to this figure (or, to be safe, close to fifty percent of this figure), you should strongly consider breaking up the data and sending it in chunks, rather than in one large block.

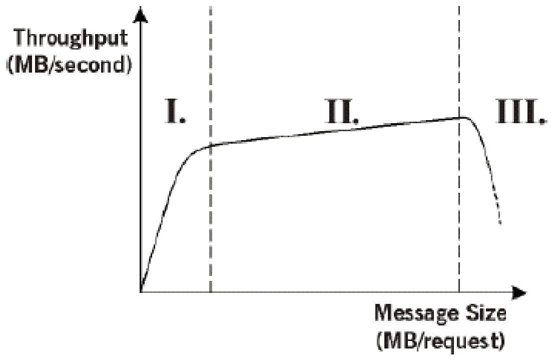

Figure 12: *Throughput Graph for CORBA Messages Across a Network*

Sending raw data

The most common way of doing this is through an Iterator, which is discussed next. Another approach is to avoid sending the raw data altogether. In some environments, clients do not need the raw data, but are just retrieving the data in order to perform some kind of processing on it. It might be possible to perform some or all of this processing on the server, and have the server return just the (smaller) results. For example, if a client needs to display a graph showing trends in some data, it might be possible for the server to render the graph and return a bitmap, rather than returning the large amount of underlying data to the client for local processing.

While this design might lighten the load on the network it does so at the cost of overburdening the server's CPU. Clearly, this is a non-trivial design area that warrants early and earnest proof-of-concepts to try out different approaches in a specific deployment environment.

Iterators

Iterators are a very common and powerful design pattern. Rather than returning the entire result set to the caller (which is potentially very large), servers instead return an initial chunk of data plus an object reference (the Iterator). Clients then make invocations on this iterator to obtain further chunks of data.

```
// IDL
interface StockPriceHistory {
   void getHistory(in Date start, in Date end, out
    PriceHistorySeq initialChunk, out PriceHistoryIterator
    iterator);
};
interface PriceHistoryIterator {
   PriceHistorySeq getNextChunk() raises endOfData;
};
```

Note: Iterators are useful if clients generally only use a subset of the data. If clients always use the full set of data, they actually slow things down by imposing additional calls. However, if the data is such that it can begin to be used incrementally, then iterators are useful. Also, it helps by reducing message size and not overwhelming the ORB with really large messages.

IDL Definition Design Guidelines

Overview

The golden rule of IDL design is that just because something is implemented in a certain way does not necessitate that its IDL reflect that implementation. You might have an OO model that accurately reflects your business, and implementation of these business entities as objects. However, you might not want to expose all these elements as CORBA objects. The remainder of this chapter discusses some of the reasons behind this.

In this section

This section discusses the following topics:

Basic Design Guidelines for IDL

Overview

This sections the basic considerations for designing IDL interfaces.

This section discusses the following topics:

- IDL design and remote invocations
- IDL basic guidelines

IDL design and remote invocations

IDL design fundamentally affects system performance and usability, much more so than for non-distributed systems. This is because remote method invocations are much more expensive than local invocations. If there are any inefficiencies, they become more readily apparent in a distributed system than in a local system.

IDL basic guidelines

There are a number of basic IDL rules that are driven by the CORBA specification that you need to be aware of:

- IDL is case sensitive in the following way; identifiers are case sensitive, but cannot differ only in case. That is, the customer interface is distinct from the Customer interface, but they cannot both exist within the same scope.
- IDL does not support overloading of operations, either within a single interface or in a derived interface. Attempting to compile IDL that has multiple operations with the same name (and different arguments or return values) result in an IDL compilation error.
- IDL attributes have restrictions/features that make them arguably undesirable. Some people view these features as restrictive or confusing enough to recommend avoiding the use of attributes altogether. Others do not mind, and use attributes.

Operation Design Guidelines

Overview

This subsection discusses basic design considerations for designing IDL operations.

This subsection discusses the following topics:

- Operation examples
- Operation design considerations
- inout parameters

Operation examples

The following interface illustrates a number of operation definitions:

```
//IDL
interface Example {
exception Failed { };

    void may_fail ( ) raises (Failed);
    unsigned long       rand( );
    unsigned long       next_prime1 (in unsigned long n);
    void                next_prime2 (in unsigned long n,
                          out unsigned long next_prime);
    void                next_prime3 (inout unsigned long n);
};
```

Operation design considerations

The three prime number operations illustrated in the preceding IDL all achieved the same thing using a different style of interface. The question is, which interface should you use?

- The first version, `unsigned long next_prime1 (in unsigned long n)`, which returns the prime number as the return value, is probably best from a stylistic point of view. If there is only one output value, make that value the return value; this style is simple and familiar to programmers.
- For operations that return several values of equal importance, it is best to make all of them `out` parameters. This avoids giving the caller the impression that the return value is somehow special or different.

- On the other hand, iterator operations frequently return two values but have one return value that has special significance because it indicates end of iteration:

```
//IDL
// An example of an operation with one special return value.

interface Iterator {
      boolean get_next( // True indicates end of iteration
         in Position start_pos,
         in unsigned short count,
         out ItemSequence items,
         out Position new_pos
      );
// ...
};

// The interface above allows the following type of code:

while (it.get_next(cursor, batch_size, item_list,
   new_cursor)) {
// Process batch of items...
cursor = new_cursor;
}
```

This operation returns the next count items from some collection as an `out` parameter, and uses the return value to indicate if there are more items in the collection.

Using the return value instead of an `out` parameter in such operations allows for a natural coding style, because the caller can write something like the following.

```
//IDL
interface Example {
void op1(inout ValType inout_param);
void op2(in ValType in_param, out ValType out_param);
}
```

If the boolean return value is an `out` parameter instead, the caller could not control the loop as easily.

inout parameters

In general, you should avoid using `inout` parameters because they dictate interface policy. An `inout` parameter overwrites its initial value with a new value. In other words, if, as the designer, you choose to use an `inout` parameter, you are making an implicit assumption that the caller does not want to keep the parameter value. If the caller wants to keep the parameter for some reason, it must make a copy first. On the other hand, if you use an in and an `out` parameter, you leave the choice as to when to discard a parameter up to the caller. As far as performance is concerned, there is no difference between the following two styles of operation definition:

```
//IDL
interface Example {
void op1(inout ValType inout_param);
void op2(in ValType in_param, out ValType out_param);
}
```

If the object instance is remote, the marshaling effort is the same for either style of definition: a value of the `ValType` type is marshaled from client to server, and a value of the `ValType` type is marshaled from server to client. It is secondary whether you use a single parameter or two separate parameters because call dispatch overhead is largely dominated by the cost of marshaling.

If `ValType` denotes a very large type, then the `inout` version of the operation is slightly more efficient because less memory is required for the duration of the call (only one `ValType` copy must be held in memory in each client and server, whereas the two parameter version requires two copies at either end). However, you notice the difference only in extreme cases (for parameters consuming several hundred kB or more).

Attribute Design Guidelines

Overview

This subsection discusses basic design considerations for designing IDL attributes.

This subsection discusses the following topics:

- Attributes and exceptions
- Error reporting
- Attributes versus operations
- Attributes versus variables

Attributes and exceptions

Attributes do not support user exceptions. Consider a `phoneNumber` attribute.

```
//IDL
void setPhoneNumber(in string theNumber) raises (InvalidNumber);
```

Because phone numbers follow a specific format, you might want your server to impose some validation on them when they are set by the client. For instance, your Customer interface might contain the following method: This approach is appealing to some because it publicly exposes the fact that the server imposes phone number validation, by including the raises clause. Compare this to the case where `phoneNumber` is exposed as an attribute. You cannot raise a user exception with attributes, so you must either validate the phone number on the client-side, allow your server to accept a mis-formatted phone number, or raise a CORBA system exception. The latter approach is unappealing, misuses the system exception, and is open to misinterpretation by clients.

Note: Many system designers want to place these kinds of data integrity/ validation on the client-side, and therefore don't view attributes as being restricted in this regard.

Error reporting

Because of the limited error reporting associated with attributes, they are not recommended. If you decide to use attributes, it is probably best to restrict yourself to read-only attributes. Writable attributes are problematic if not all values in the range of the attribute's type are legal, because the best form of error reporting you can have is to raise a `BAD_PARAM` or other system exception.

Attributes versus operations

The following two sections of IDL is semantically equivalent.

```
//IDL
interface Thermostat {
readonly attribute TempType temperature;
attribute TempType nominal_temp;
};
```

```
//IDL
// Attributes are not necessary. The equivalent IDL without
attributes:
interface Thermostat {
TempType get_temperature();
TempType get_nominal_temp();
void set_nominal_temp(in TempType t);
};
```

There is no difference as far as performance is concerned between attributes and operations. Attributes are in fact implemented as a pair of operations or, in case of a read-only attribute, as a single accessor operation. However, IDL does not permit you to add a raises expression to an attribute definition, or define an attribute as oneway. This means that error reporting for reading and writing of attributes is limited to system exceptions, which are less informative than user exceptions.

Attributes versus variables

Attributes are not member variables, even though they appear to be. An attribute need not correspond to a variable (or any other piece of object state) in an object's implementation. As far as CORBA is concern, attribute access is the same as any other remote procedure call.

Exception Design Guidelines

Overview

The following guidelines help you to design APIs that are easier to use and understand, which result in code that is easier to develop and maintain, and has lower defect rates.

This section discusses the following topics:

- When to use exceptions
- What information exception should contain
- What information exceptions should not contain
- Multiple error conditions and exceptions

When to use exceptions

Do not raise exceptions for expected outcomes. For example, a database lookup operation should not raise an exception if a lookup does not locate anything; it is normal for clients to occasionally look for things that are not there. It is substantially harder for the caller to deal with exceptions than with return values because exceptions break the normal flow of control. Do not force the caller to handle an exception when a return value is sufficient.

What information exception should contain

Ensure that exceptions carry all of the data the caller requires to handle an error. If an exception carries insufficient information, the caller has to make a second call to retrieve whatever information is missing. However, if the first call failed, there is a good chance that subsequent calls will fail as well.

What information exceptions should not contain

Do not add exception members that are irrelevant to the caller. In particular, telling the caller something it already knows is worse than useless.

Multiple error conditions and exceptions

Do not lump multiple error conditions into a single exception type. Instead, use a different exception for each semantic error condition; otherwise, the caller can no longer distinguish between different causes for an error.

Sequences Versus Arrays

Overview

This section provides basic guidelines for using sequences and arrays.

This section discusses the following topics:

- Differences between sequences and arrays
- Guidelines to using sequences and arrays

Differences between sequences and arrays

The main difference between sequences and arrays is that the number of elements in a sequence can vary at runtime, whereas the number of elements in an array is fixed at compile time.

Guidelines to using sequences and arrays

The following guidelines help you to decide whether to use a sequence or an array:

- Use a sequence if you have a variable number of values in a collection.
- Use an array if you have a fixed number of values, all of which exist at all times.
- Use an array of char to implement fixed-length strings.
- Use a sequence to model sparse arrays, for efficiency.
- Use a sequence to model a recursive (self-referential) data structure.

IDL Modules and Scoping

Overview

Modules group related IDL definitions into an enclosing naming scope: The main purpose of modules is to prevent pollution of the global scope. By choosing an appropriate module name, you make it less likely that clashes with other developers' IDL definitions occur.

This section discusses the following topics:

Reopening modules

Overview

Modules can be reopened (in the same or in a different source file). This feature allows you to define the contents of a module incrementally, as shown in the sample IDL in this subsection.

Uses of reopened modules

The main uses of reopened modules are to decouple developers from each other and to limit the amount of code that must be recompiled after a change. By splitting parts of modules across different IDL source files, you reduce the amount of code that requires recompiling (provided that the change does not affect all source files). For example the `ICS` module is opened in the sample IDL.

```
//IDL
module ICS { // Irrigation Control System
interface RainGauge {
    ...
  };
};

module ICS {
typedef float RainfallType; // Precipitation
typedef string LocType; // Location
typedef string ModelType; // Model description
};
module Weather {
  interface CurrentData {
    ICS::RainfallType todays_rainfall();
    ...
  };
// ...
};
module ICS { // Reopen ICS module
  interface RainGauge {
    RainfallType get_rainfall();
    ...
  };
  ...
};
```

Scope Resolution Operator

Overview

You can use the `::` scope resolution operator to explicitly refer to an identifier that is defined inside an interface.

This section discusses the following topics:

- Scope resolution example
- Scope resolution rules
- Scope resolution examples

Scope resolution example

Consider the following example:

```
//IDL
// You can use the :: scope resolution operator to refer to
// identifiers defined inside an interface
interface Haystack {
typedef short NeedleID;
  ...
};
interface Camel {
void go_through(in Haystack::NeedleID needle);
};
```

Here, the qualified name `Haystack::NeedleID` is used to refer to the `NeedleID` type defined inside the Haystack interface. a leading `::` scope resolution operator indicates the global scope, so `::Haystack::NeedleID` could have been used instead.

Scope resolution rules

IDL applies the same scope resolution rules as Java and C++:

- To locate the definition of a name whose enclosing scope is not an interface, the compiler successively searches enclosing scopes toward the global scope.
- When searching in an interface, the compiler first searches that interface for the definition.

- If the name cannot be found in that interface, the compiler successively searches base interfaces toward the Object root.
- If the name cannot be found in a base interface, the compiler successively searches the enclosing scopes of the derived interface toward the global scope.

Scope resolution examples

The following examples illustrate the scope resolution rules outlined in the previous section.

- The following IDL definition redefines `RainfallType` inside the `RainGauge` interface. This is legal because interfaces form a naming scope.

```
//IDL
// Example of scope resolution rules - NOT a good example of
// IDL style!
//
module ICS {
typedef float RainfallType; // (1)
interface RainGauge {
typedef long RainfallType; // (2) Confusing
but legal
RainfallType get_rainfall(); // Returns a long
(2)
ICS::RainfallType rainfall(); // Returns a
float (1)
};
};
```

- When the compiler parses the definition of `get_rainfall`, it searches the immediately enclosing scope for a definition of `RainfallType` and locates the alias for `long`. On the other hand, for the definition of `rainfall`, the qualified name `ICS::RainfallType` means that rainfall returns a `float`.

- In the presence of inheritance, base interfaces are searched before enclosing scopes, and the enclosing scopes of base interfaces are not searched at all, as shown in the following example.

```
//IDL
// Example of scope resolution with inheritance.
// Base interfaces are searched before enclosing scopes.
// Definitely not a good example of IDL style.
module Sensors {
typedef short RainfallType; // (1)
typedef string ModelType; // (2)
interface RainGauge {
typedef long RainfallType; // (3)
RainfallType get_rainfall(); // Returns a long (1)
ModelType model(); // Returns a string
(2)
};
};
module Regulators {
typedef double RainfallType; // (4)
interface Sprinkler : Sensors::RainGauge {
RainfallType current_setting(); // Returns a
long (1)!
ModelType my_model(); // Error
};
};
```

In this example, `RainGauge::get_rainfall` returns a `long` because the local definition of `RainfallType` hides the definition in the enclosing scope (`Sensors::RainfallType`).

For the definition of `Sprinkler::current_setting`, the return value is a `long` (not a `double`) because base interfaces are searched before the enclosing scopes, so inside interface `Sprinkler`, `Sensors::RainGague::RainfallType` takes precedence over `Regulators::RainfallType`.

The definition of `Sprinkler::my_model` is in error. Even though `ModelType` is defined in the enclosing scope of `Sensors::RainGauge` (which is a base interface), the compiler never searches the enclosing scopes of base interfaces to locate a definition.

Looking at the preceding two examples, you probably find that they are difficult to understand and confusing. If so, you should avoid using the same name in different scopes. That way, you have a more readable specification and avoid having to know the scope resolution rules in intricate detail.

The CORBA Module

Overview

The ORB provides a number of APIs to clients and servers, for example, to initialize and finalize the ORB runtime. The OMG publishes APIs in IDL because this permits a single specification to cover the APIs for all possible implementation languages. The mapping rules from IDL to each particular implementation language take care of defining the precise appearance of each API in its target language.

This subsection discusses the following topics:

- Location of the ORB runtime APIs
- The orb.idl

Location of the ORB runtime APIs

APIs related to the ORB runtime (also called the CORBA Core) are grouped into the CORBA module. The prefix `omg.org` is used for all specifications published by the OMG to ensure unique repository IDs.

The orb.idl

File If you want to use IDL types defined in the CORBA module, such as `CORBA::TypeCode`, you must include `orb.idl` in your IDL source file to import the relevant definitions. Note that this is necessary only if you actually use types defined in the CORBA module. If you include `orb.idl` in specifications that do not require it, it unnecessarily bloats the size of the generated code.

Orbix COBOL and PL/I application servers stores all OMG-specified IDL files in the *orbixhlq*`.INCLUDE.OMG.IDL` data set. CORBA does not specify where standard IDL files should be stored, so this is a Micro Focus-specific detail.

CHAPTER 6

Locating CORBA Objects

This chapter introduces the fundamental concepts of CORBA object location. Clients know that services exist, but require an object reference before being able to make any invocations on the service. How do clients obtain these object references? This chapter discusses the contents of an IOR, and introduces the CORBA Naming Service used for publishing and obtaining IORs. Several approaches to publishing and locating objects are discussed, and their strengths and weaknesses are explored.

In this chapter

This chapter discusses the following topics:

CORBA Object Location

Overview

A fundamental question that every CORBA system must answer is: How do clients obtain object references? Before a client can make use of any server, it must first have a reference to the server object in question. In every distributed system, clients must know where to go to find their services. Clients can obtain this information in a number of ways; for instance, it can be stored in a client-side configuration file, or provided by a user upon application startup. The business applications used every day (such as email clients and web browsers) are examples of these. CORBA objects can be located on many different hosts, or even different hosts over time, so clients need a flexible way to be able to find objects.

This section discusses the following topics:

- Object location goals
- Object location model
- Interoperable object references (IORs)
- Locating the naming service

Object location goals

A client can be hard coded with an object's location, but this is very restrictive. If the object is relocated to a different host, you want an easy way to point the clients to the new location. One possibility is to use some information stored on the client side (such as a configuration file or environment variable) to find the object. This is an improvement over hard-coding, but still poses difficulties. Whenever an object's location changes, all clients have to be updated with the new location. A better solution is to remove the object's location from the client side entirely, and have the client determine the object location at runtime. This also gives us the potential for fault tolerance and load balancing. So, the goals for an object location solution are as follows:

- A client should be able to efficiently obtain a reference to, and begin using an arbitrary object on an arbitrary host.
- The client program should be unaware of the host and server in which the object lives.

- The solution should support easy reconfiguration of the client. If object implementations are moved from one host to another, an ideal solution would involve no reconfiguration of the client, and no recompilation. An acceptable solution might involve reconfiguration of the client.

If the client's environment is flexible, it is easier to introduce and support mechanisms that support fault tolerance and load balancing. For example, load balancing might be implemented by having the client use a mechanism to determine, at runtime, which of a pool of servers should supply the object reference.

Object location model

One approach to solving the problem is to introduce a level of indirection between the client and the server, as shown in Figure 13.

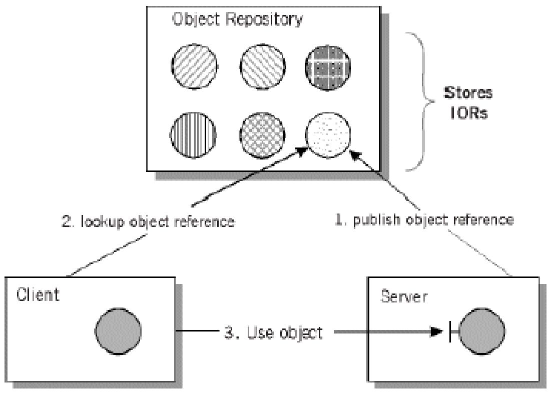

Figure 13: *Object Location Model*

This makes use of an Object Repository, where object references are stored. Conceptually, the following steps are involved.

1. The server publishes an object reference to the repository.
2. The client obtains the object reference from the repository, by specifying some information that identifies the object.
3. The client uses the object, invoking directly on the server.

Interoperable object references (IORs)

IORs are data structures, in a standard format, that contain:

- Information that clients use to establish connections to servers.
- Information that servers use to identify target objects.

The primary elements of an IOR are:

- IP host address: using a dotted IP address or a DNS name. To ensure correct operation in a Dynamic Host Configuration Protocol (DHCP)-configured network, Orbix should be configured to insert the DNS name into the IOR.
- TCP port number: identifies a listening TCP port on the host.
- Object key: vendor specific information interpreted by the server to identify an object.
- Repository ID: identifies the IDL type as described in its IDL of a CORBA object.

IORs contain enough information for a client to be able to find the target object. The IOR is not usually interpreted by the application; the application typically retrieves the IOR from a repository, and just uses it. The ORB is responsible for interpreting and locating the IOR.

Locating the naming service

In order to locate your application objects, you can make use of the CORBA Naming Service. The question then becomes how you obtain references to this service.

Fortunately, the CORBA specification provides a standard API that ORB vendors must implement, to give applications access to these bootstrap services in a standard way.

For COBOL and PL/I this API is `OBJRIR`, `"NamingService"` must be supplied as its argument. The ORB then returns a reference to the Naming Service. Refer to either of the COBOL or PL/I Programmer’s Guide and Reference for more details of `OBJRIR`.

The CORBA Naming Service

Overview

The Naming Service is an example of an object repository. It is a standard CORBA service, implementing the IDL specified by the OMG. Application servers export object references into the naming service, providing an associated name. The naming service stores these object references in its database, keyed by the supplied name. Later, clients retrieve objects from the naming service by providing a name. The naming service returns the object reference with the matching name.

The client and server have been successfully decoupled from one another. The client no longer has to have any configuration information about the server or host in which the object lives. The client simply asks for the object by name, then begins using it. If the object is relocated, the object reference stored in the naming service database can simply be updated. The next time a client asks for the object by name, it automatically obtains the new object reference.

This section discusses the following topics:

- Sample naming service hierarchy
- Naming service IDL interface
- The bind operation
- The resolve operation

Sample naming service hierarchy

The naming service stores its object references in a hierarchical format, analogous to a file system. The naming service stores object references and naming contexts (which can contain object references and other naming contexts).

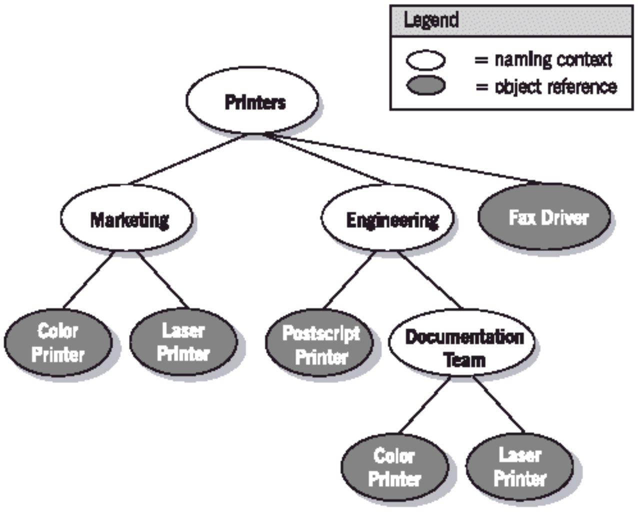

Figure 14: *Sample Naming Hierarchy*

Shows a simple naming service hierarchy that contains several objects and naming contexts.

Naming service IDL interface

The naming service is a CORBA server, and thus has an IDL interface describing it. This interface is used by servers and clients, to store and retrieve object references from the naming service database. All the IDL components are defined in the `CosNaming` module:

```
//IDL
Name components are stored in an IDL structure
struct NameComponent {
           Istring id;
           Istring kind;
};
```

(where `Istring` is simply type defined to a string).

A compound name (IDL type Name) is simply a sequence of these `NameComponent` structures:

```
//IDL
typedef sequence<NameComponent> Name;
```

The primary object that applications interact with is the `NamingContext` interface. Within this interface are a number of operations, only two of which are discussed here:

```
//IDL
voidbind(inNamen,inObjectobj)raises(NotFound,
CannotProceed, InvalidName, AlreadyBound);
Object resolve (in Name n) raises (NotFound, CannotProceed,
InvalidName);
```

The bind operation

The bind operation takes a Name (a sequence of Name-Component structures), and an Object as input. Servers use the `NamingContext::bind` method to store an object reference in the naming service database. The naming service stores the object reference in the appropriate place in its hierarchy. The name specified is interpreted relative to the `NamingContext` object on which the method has been invoked. If the name passed to `bind()` is a compound name (a sequence) with more than one component, then all except the last name component are used to find the naming context to which to add the binding (these naming contexts must already exist). The last name component in the sequence denotes the object reference. The `bind()` operation raises an exception if the specified name is already bound within the final naming context.

The resolve operation

The resolve operation is invoked on a `NamingContext` object by clients, to obtain object references. The application specifies a name sequence, and the naming context returns an object that matches the specified name. Note that the specified name is interpreted relative to the target naming context. The return type from this method is Object (`CORBA::Object_ptr` in C++). The application program must narrow this reference (using the generated `_narrow()` method) to a reference of the appropriate interface class. For both bind and resolve, there are a number of exceptions that can be thrown when errors are encountered.

Federating Naming Hierarchies

Overview

Naming contexts can contain object references as well as other naming contexts. These naming contexts can be remote as well as local (that is, you can federate naming hierarchies together). A naming context from a remote host's naming service can be placed into the local naming service hierarchy, as shown in Figure 15. When an application is browsing the hierarchy, (or, simply obtaining an object reference), it is redirected to the remote naming service.

This section discusses the following topics:

- Initial connection to the naming service
- Naming service applicability

Initial connection to the naming service

There is no central root naming context (unlike file systems). Instead, the initial naming context seen by an application is simply the top-level naming context on the host to which an application initially connects.

Note: Applications must be configured with the host name on which to find the naming service.

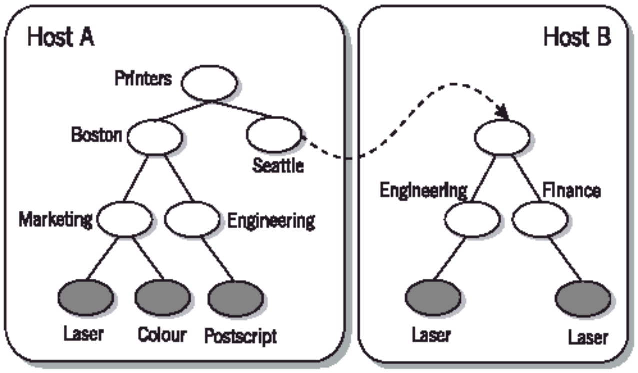

Figure 15: *Federated Naming Hierarchy*

Naming service applicability

Using the naming service is often a good solution when:

- Clients look up objects based on a fixed and consistent set of criteria.
- Clients only want a single object reference returned.
- Lookup properties have static values.

Structuring the Naming Hierarchy

Overview

A naming hierarchy is a two-dimensional space that can be structured in many ways. The best structure depends on how the system uses it. Consider who navigates the hierarchy (people, programs, or both). For:

- Human users, a more descriptive hierarchy is often preferred.
- Programs, a more compact hierarchy is often preferred.

Hierarchies can also be flat or deep, depending on how you choose to identify the objects in them.

This section discusses the following topics:

- Descriptive hierarchy
- Compact hierarchy
- Flat hierarchy
- Deep hierarchy

Descriptive hierarchy

The descriptive hierarchy shown in Figure 16 is well-suited to human browsing (and not well-suited to browsing by a computer program). The descriptions of the objects in particular are irregular and would be difficult for a program to handle.

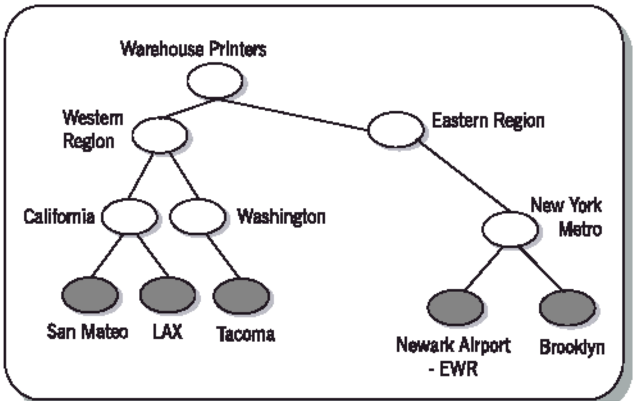

Figure 16: *Descriptive Naming Hierarchy*

Compact hierarchy

The compact hierarchy shown in Figure 17 is not very meaningful to a human (all the entries in the hierarchy are uniform). This would be well-suited to browsing/management by an application, rather than by a person.

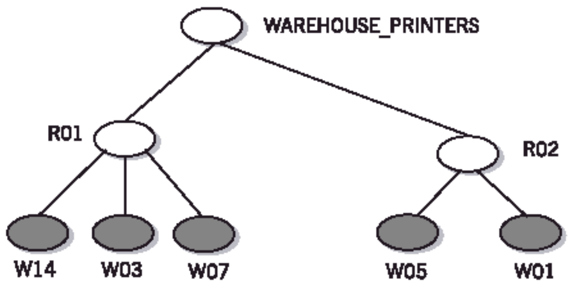

Figure 17: *Compact Naming Hierarchy*

Flat hierarchy

Objects in the flat hierarchy shown in Figure 18 are uniquely identified by only their Name—their id and kind fields must be unique. Names can be either descriptive or compact, but must be unique within the context.

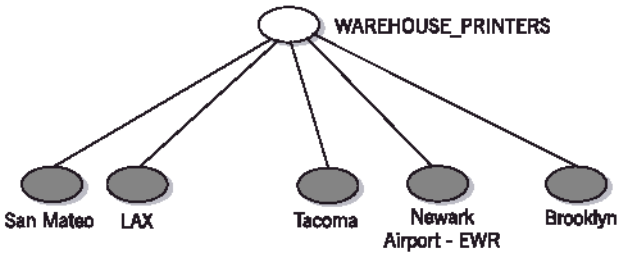

Figure 18: *Flat Naming Hierarchy*

Deep hierarchy

Objects in the deep hierarchy shown in Figure 19 are uniquely identified by their position as well as their Name. Their id and kind fields can be repeated in different contexts In the sample hierarchy shown, two objects have the same (simple) name— their id is Laser Printer, and their kind is HP5. However, they reside in different contexts, so they have unique (compound) names. If your domain is such that more than one object has the name simple name, you must construct different naming contexts to contain these objects.

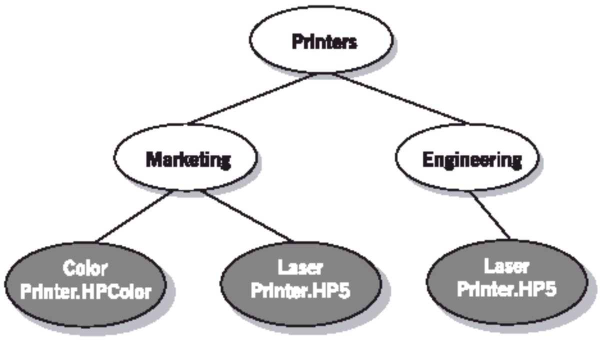

Figure 19: *Deep Naming Hierarchy*

Custom Object Location Mechanisms

Overview

Often an application needs a customized location service to best meet its requirements. Application design might include interfaces that act as factory or look-up objects (these objects act as *entry points* and can be advertised via the naming service). This avoids the requirement that every object be registered with the naming service.

This section discusses the following topics:

- Look-up service
- Advantages of standard interfaces
- CORBA interoperable naming service

Look-up service

An optimized, application-specific look-up service might out-perform generic services like the Naming Service. For example, if names correspond directly to database keys or object markers, then an in-server look-up object can resolve the names very efficiently, because it has direct access to the server's underlying database and object tables.

Advantages of standard interfaces

However, standard interfaces offer important advantages, even if the application requires specialized implementations. For example, suppose a server provides a large number of account objects, where the account number acts as the name of the account. Performance requirements demand a specialized look-up object that is implemented in the account server and has direct access to the account implementation objects and the database where they live.

How can this approach be reconciled with a desire to provide a transparent, standard, easy-to-use location scheme? You can implement the standard `CosNaming::NamingContext` interface for the look-up object. The optimized `resolve()` method uses direct access to the server's internal data structures to efficiently locate the appropriate object; however from a clients perspective this is just a normal naming context, which can be embedded seamlessly into a larger name-space, implemented using a standard naming service.

Often, application designers implement their own object-location mechanisms, customized/optimized for their specific environment

These might use CORBA-standard APIs such as:

- `object_to_string()` in C++ which is equivalent to `OBJTOSTR` in COBOL and `OBJ2STR` in PL/I.
- `string_to_object()` in C++ which is equivalent to `STRTOOBJ` in COBOL and `STR2OBJ` in PL/I.

Implementing (or extending) the standard CORBA IDL in a customized manner is an appealing combination:

- Efficient and customized implementation to meet your specific needs.
- Standard interfaces are familiar and interoperable (and can be federated into standard CORBA services).

CORBA interoperable naming service

The CORBA Interoperable Naming Service is an extension to the Naming Service specification and standardizes a number of elements in the original Naming Service specification, including:

- A standard string representation of names.
- A URL format for names (both with IOR and stringified names).
- Standard configuration of returning a single initial naming context to all clients via `resolve_initial_references`.
- A few other clarifications and enhancements to the specification.

Glossary

A

Abstract class
A class that contains one or more abstract methods, and therefore can never be instantiated. Abstract classes are defined so that other classes scan extend them and make them concrete by implementing the abstract methods.

Adaptive Runtime Technology
Micro Focus's innovative, plug-in runtime architecture supporting dynamic deployment and configuration of core middleware services, as well as application code.

ART plug-in
A code library that can be loaded into an Orbix application at link time or runtime.

Asynchronous communication
A form of communication in which applications operate independently and do not have to be running or available simultaneously. A process sends a request and may or may not wait for a response. It is a non-blocking communications style.

Attribute
An IDL attribute is shorthand for a pair of operations that get and, optionally, set the values of an object.

C

Class
In an object-oriented programming paradigm, refers to a template from which objects can be instantiated. A class defines the state (attributes) and the behavior (methods) that characterize all objects that are instances of that class. Typically, a class can inherit state and behavior from other classes.

Class method
A method that is invoked without reference to a particular object. Class methods affect the class as a whole, not a particular instance of the class. Also called a static method. See also instance method.

Client
An application (process) that typically runs on the desktop and requests services from other applications that often run on different machines (that is, server processes). In CORBA, a program that requests services from a CORBA object.

Client/Server
A relationship between two processes in which one sends requests to the other. Usually a client sends requests to a server. It is possible, however, for the server to send a request to the client and thereby reverse the roles.

Component Object Model (COM)
A model specified by Microsoft to define objects and how they interoperate. A COM object can be accessed by any COM-compliant application. COM is different from CORBA in many ways; for example, there are differences in the mechanisms by which objects are referenced, and in the process by which objects are created.

Configuration domain
A set of common configuration settings that defines the available services, and controls the behavior of ORBs. Related application ORBs usually share configuration domains, which are divided into nested configuration scopes. Configuration domain information can be implemented as either a set of local configuration files or as a centralized configuration repository. A configuration domain can contain multiple location domains.

CORBA Naming Service
The CORBA service that provides the ability to associate a name with an object reference. It provides a standardized service to allow users to locate objects in a system using common names that are bound to the objects. The naming service can be searched to find an object by using, as a reference, the name that is bound to the object. The CORBA naming service is called OrbixNames, and is included in Orbix.

CORBA object
A virtual object that has an associated IDL interface and can receive requests from clients through an ORB. CORBA objects are implemented by servants.

CORBAservices

A set of system services for CORBA objects that were developed for programmers. These services were specified by the OMG and implemented by CORBA vendors. They provide users with CORBA standardized services to create objects, track objects and object references, and control the relationships between types of object. Examples include the CORBA Interoperable Naming Service—a basic directory service which stores name to object reference bindings in a central location and the CORBA persistent state service—a CORBA-type persistence mechanism for defining how objects stored in databases are reused.

D

Daemon

A daemon (pronounced demon) is a program that runs continuously and exists for the purpose of handling periodic service requests that a computer system expects to receive. The daemon program forwards the requests to other programs (or processes) as appropriate. In an Orbix system there is an Orbix daemon called `orbixd`, which locates and activates CORBA servers and services.

Data type

Identifies the set of values that a programming-language object can take, and the operations that can be carried out on them.

Declaration

A statement that establishes an identifier and associates attributes with it, without necessarily reserving its storage (for data) or providing the implementation (for methods). See also definition.

Definition

A declaration that reserves storage (for data) or provides implementation (for methods). See also declaration.

Distributed application

An application made up of distinct components running in separate runtime environments, usually on different platforms connected via a network. Typical distributed applications are two-tier (client-server), three-tier (client-middleware-server), and multi-tier (client-multiple middleware-multiple servers).

Distributed Environment
A configuration of hardware and software that is physically and geographically dispersed but networked together for communication.

Dynamic Invocation Interface (DII)
The component of an ORB that allows clients to access CORBA objects without knowing the object interface at compile-time.

Dynamic Skeleton Interface (DSI)
The component of an ORB that allows a server to receive and process requests for IDL interfaces, without knowing those interfaces at compile-time.

G

Generic Inter-ORB Protocol (GIOP)
A CORBA standard abstract protocol that defines the data representation and basic message formats used by other CORBA protocols, such as IIOP.

Granularity
The relative size, scale, level of detail, or depth of penetration that characterizes an object or activity. If, for example, a bank system is built in which each element of a person's bank account—such as balance, name, account number—is treated as an object, it is consider more fine-grained than a system in which the entire account is treated as a single object.

I

IDL Compiler
The compiler provided with an ORB, it generates stub and skeleton code from supplied IDL definitions.

Implementation Repository (IMR)
The component of an ORB that stores definitions of the servers available in a distributed system. It is managed by the locator daemon. It is a CORBA database that maintains information about servers and controls their activation. An Implementation Repository maintains a mapping from a server name to the file name of the executable code (for example, COBOL or PL/I) that implements the server. A server must be registered in the Implementation Repository to be launched automatically by the Orbix daemon. Users should maintain records in the Implementation Repository of each server in a system.

Inheritance

The ability to incorporate the features and functionality of an existing object into a new object or objects. This is key to code reuse and is a basic property of all objects.

Instance

A specific object within a class. For example, BankAccount1 is a specific instance of the class `BankAccount`.

Instance Variable

The data associated with a specific instance of a class. Sometimes referred to as the attributes of an object.

Interface

The fundamental abstraction mechanism of CORBA. An interface describes a type of object, including the operations and attributes that the object supports in a distributed enterprise application. The definitions of operations and attributes must be defined within the naming scope of an interface. Definitions of exceptions, types, and constants can be defined within the scope of an interface or at a higher scope.

Interface Definition Language (IDL)

The CORBA standard language that allows a programmer to define the interfaces to CORBA objects. Clients use these interfaces to access objects across a network. It is a technology-independent language that describes all component interface characteristics used by CORBA applications. It enables the exchange, flow and control of information between systems.

Interface Repository (IFR)

The component of an ORB that stores IDL definitions for objects in a distributed system. A client can query this repository at runtime to determine information about an object's interface, and then use the DII to make calls to the object.

Internet Protocol (IP)

A protocol that provides the routing mechanism to store and forward data packets from one network to another.

Internet Inter-ORB Protocol (IIOP)

A standard messaging protocol (format for the layout of messages sent over a network) defined by the OMG for communications between ORBs. It is the CORBA standard protocol for communications between distributed applications. IIOP is defined as a protocol layer above TCP/IP.

Interoperable Object Reference (IOR)

Specifies the location of, the unique identity of, and the services supported by a CORBA object. It also has a standard format that allows an ORB implementation that is different to the one that created it to use it to communicate with the object.

L

Location domain

A collection of servers on any number of hosts across a network that is under the control of a single locator daemon. The locator daemon automatically activates remote servers, using stateless activator daemons running on the remote hosts.

Locator daemon

A server host facility that manages an implementation repository. Orbix clients use the locator daemon, often in conjunction with a naming service, to locate the objects they seek. Together with the implementation repository, it also stores server process data for activating servers and objects.

M

Marshalling

The process of converting native programming language data types to a format suitable for transmission across a network.

Message

Information sent between applications (processes). Messages can include data, program instructions, or both.

Method

The object-oriented programming term for the behavior associated with an object. It is the Java equivalent of a C++ function or IDL operation.

Module

An IDL module contains, and provides a naming context for, all or part of an application's IDL definitions. Modules and interfaces form naming scopes.

Middleware

Software that interfaces both the operating system and applications that run on it, providing services that are common to a number of applications but independent of the operating system. Middleware approaches for implementing n-tier client/server applications include remote procedure call, distributed transaction processing, object components, and message-oriented middleware.

Multithreading

Multithreading is the creation of multiple threads in a single program. A thread is placeholder information associated with a single use of a program that can handle multiple concurrent users. From the program's point of view, a thread is the information needed to serve one individual user or a particular service request. If multiple users are using the program or concurrent requests from other programs occur, a thread is created and maintained for each of them. Each thread identifies for the program which user is being served as the program is re-entered on behalf of different users. For any system to scale it needs to be able to support multiple concurrent users. All versions of Orbix are multi threaded.

N

Naming context

An object in the naming service that you can use to resolve the names of application objects. Naming contexts and application objects are organized into naming graphs.

Naming graph

A hierarchy of naming contexts and application objects.

Naming scope

IDL modules and interfaces form naming scopes, so identifiers defined inside an interface need only be unique within that interface. To resolve a name, the IDL compiler conducts a search among the following scopes, in the order shown:

- The current interface.

- The base interfaces of the current interface, if any.
- The scopes that enclose the current interface.

Naming service

The Orbix implementation of the CORBA Interoperable Naming Service. It allows you to associate abstract names with CORBA objects in your application, thus allowing clients to locate the objects, by looking up the corresponding names.

N-Tier Client/Server

An application development approach that partitions application logic across three or more environments— the desktop computer, one or more application servers, and a database server. The main advantage of the n-tier client/server application development approach is that it extends the benefits of client/server applications to an enterprise level. Other advantages include added manageability, scalability, security, and higher performance.

O

Object

In object-oriented programming, a single software entity that consists of both data and procedures that manipulate that data. In CORBA, objects can be located anywhere in a network. The functionality of a CORBA object is accessed through interfaces defined in IDL. By encapsulating object interfaces within a common language, IDL facilitates interaction between objects, regardless of their actual implementation, thereby ensuring interoperability between different languages and platforms.

Object Interface

The interface to an object, as defined in an application's OMG IDL statements. The object interface identifies the set of operations and attributes that can be performed on an object. For example, the interface for a teller object identifies the types of operation that can be performed on that object, such as withdrawals, transfers, and deposits.

Object Management Architecture (OMA)

A definition of a standard object model from the Object Management Group that defines the behavior of objects in a distributed environment. The communications component of OMA is the Common Object Request Broker Architecture (CORBA).

Object Management Group (OMG)

A consortium that aims to define a standard framework for distributed, object-oriented programming. The OMG is responsible for the CORBA and OMA specifications.

Object Model

A model that describes the overall design of an application or system in terms of objects.

Object reference

When a server starts, it creates one or more objects and publishes their references, usually in a naming service (a file or a URL can also be used). When a client program makes a request on an object, it passes the name of the required object to the ORB, which passes it to the naming service. The naming service returns the corresponding object reference to the ORB, which uses the reference to pass the request to the server object.

Object Request Broker (ORB)

A middleware component that acts as an intermediary between a client and a distributed object. It is responsible for delivering messages between the client and object across a network. It hides the underlying complexity of the distributed system, such as differences in hardware, operating systems, and programming languages, from the application programmer. When a client invokes a member function on a remote CORBA object, the ORB intercepts the function call, formats it as a CORBA request, and redirects it across the network to the target object.

An ORB uses the CORBA Interface Repository to find out how to locate and communicate with a requested component. When creating a component, a programmer uses either CORBA’s Interface Definition Language (IDL) to declare the component’s public interfaces, or the compiler of the programming language translates the language statements into appropriate IDL statements. These statements are stored in the Interface Repository as metadata or definitions of how a component's interface works. Orbix and OrbixWeb are ORBs.

Operation

The IDL equivalent of a function in C++ or method in Java. Operations are defined in IDL interfaces and can be called on CORBA objects. When a client invokes a member function on a CORBA object, the ORB intercepts the function call, formats it as a CORBA request, and redirects it across the network to the target object whether the object is in the same address space as the client, in another address space in the same machine, or on a remote machine.

Orbix daemon

In an Orbix system there is an Orbix daemon called orbixd, which locates and activates CORBA servers and services. Orbix runtime environments The runtime environments contain the services (configuration domains, location domains, and naming service) needed to make Orbix applications work. They can exist as part of a complete built-in Orbix solution or as an independently manageable Orbix-ready application.

P

Portable Object Adapter (POA)

The adapter that delivers requests from the ORB (via the network) to the programming language objects that implement CORBA objects. POAs can be transient or persistent. The POA is an addition to the CORBA specification to unify and extend the capabilities of a CORBA server. It is a replacement for the Basic Object Adapter (BOA) and is a runtime library of routines that are built into the server application executable image. A POA creates and manages object references to all objects used by the application, manages object state, and provides the infrastructure to support persistent objects and the portability of object implementations between different ORB products. The POA is the only standard CORBA object adapter.

R

Request

A message sent by a client application that identifies an operation to be fulfilled. The message is sent to the ORB and is relayed to the appropriate server application, which fulfills the request.

S

Servant

The programming language object that implements the interface defined in an application's OMG IDL statements. A servant contains the method code that can perform operations on one or more CORBA objects.

Server

A program that provides services to clients. CORBA servers act as containers for CORBA objects, allowing clients to access those objects using IDL interfaces.

Server Object

The object that performs server application initialization functions, creates one or more servants, and performs server application shut-down and clean-up procedures.

Skeleton

The ORB component that is specific to the object interface and that assists a Portable Object Adapter in passing requests to particular methods. The skeleton is connected to both the server application and the ORB. It is produced by the IDL compiler and is used at runtime to invoke method code.

Superclass

A class from which a particular class is derived, perhaps with one or more classes in between. See also subclass, subtype.

T

Two-Tier Client/Server

An application development approach that splits an application into two parts and divides the processing between a desktop workstation and a server machine.

U

Unmarshalling

The conversion of data, received over a network, from its on-the-wire representation to data types appropriate to the receiving application.

V

Variable
A programming language object that can contain any one of a defined set of values, often of a particular data type. Contrast with constant.

W

X

Y

Z

Index