WILEY eBOOK

Programming with VisiBroker®

Programming with VisiBroker®

A Developer's Guide to VisiBroker for Java™

Second Edition

Vijaykumar Natarajan
Stefan Reich
Bhaskar Vasudevan

Wiley Computer Publishing

John Wiley & Sons, Inc.
NEW YORK • CHICHESTER • WEINHEIM • BRISBANE • SINGAPORE • TORONTO

Publisher: Robert Ipsen
Editor: Robert M. Elliott
Assistant Editor: Emilie Herman
Managing Editor: John Atkins
Associate New Media Editor: Brian Snapp
Text Design & Composition: North Market Street Graphics

This book is printed on acid-free paper. ♾

Published by John Wiley & Sons, Inc.

Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data
Natarajan, Vijaykumar, 1972–
Programming with VisiBroker / Vijaykumar Natarajan, Stefan Reich, Bhaskar Vasudevan.—2nd ed.
p. cm.
"Wiley Computer Publishing."
Includes index.
ISBN: 0-471-37682-5 (paper : alk. paper)
1. Java (Computer program language) 2. VisiBroker. I. Reich, Stefan, 1970– II. Vasudevan, Bhaskar, 1970– III. Title.

QA76.73.J38 N36 2000
005.2'762—dc21 00-063320

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To the VisiBroker team, past,
present and future . . .

Contents

Acknowledgments

Embarking on a project like this one can be quite exciting and daunting at the same time and despite the challenges along the way, I can say that this has been an awesome experience. I would like to thank Jonathan Weedon for giving me this great opportunity.

I would like to thank Bob Elliott at Wiley for persevering and driving this project to completion. My gratitude and extra special thanks to Emilie Herman for her guidance and patience during this project and for keeping us on track. Exasperating as I may have been at times, I really appreciate her maintaining a cheerful attitude through it all and lifting my spirits in turn.

My thanks to John Atkins and the other members of production crew at Wiley for their pushing the quality of this book up a few notches. Thanks also to the authors of the first edition for the great base that we could start our work with.

I would also like to thank the VisiBroker teams and all the great engineers I've worked with for all that I learned from them and for making VisiBroker a great product.

Special thanks to Ioana Pirvulescu for writing the chapter on Interceptors, Gopal Ananthraman for writing the chapter on dynamic VisiBroker, and Tomasz Mariusz Mojsa for writing the chapter on the gatekeeper.

I would like to also thank my dear wife, Chitra, for being so patient and supportive while I spent those evenings writing, even though we were newly wed. I thank my dear friend Kirthi for always pushing me to do more and for reviewing the book for me. And finally, I would like to thank my colleagues (past and present), friends, and my family for their support and guidance.

–VIJAYKUMAR NATARAJAN

Working with our two VisiBroker teams has been a great experience, both personal and professional. I would like to thank my family for their encouragement, my friends for all the great times, and my wife Tatjana for taking a bold step.

–STEFAN REICH

Thanks to Vijay Natarajan and Stefan Reich, without whom this book would not have been possible. I am deeply indebted to them for their hard work and for completing the book while I was away from the US. My sincere thanks to my ex-colleagues at Inprise Corporation for their support and encouragement. Thanks to Emilie Herman and Bob Elliott at Wiley for their patience with us during the course of writing this book.

–BHASKAR VASUDEVAN

Introduction

Over the last decade, the much-hyped benefits of software reuse have never been realized by most software organizations. Various excuses have been offered, from a lack of management commitment to the not-invented-here syndrome. But the real bane of software reuse has always been the great difficulty of interoperability between software written in different languages or on heterogeneous hardware platforms. That is beginning to change, thanks to the maturation and convergence of two independent technologies: CORBA (Common Object Request Broker Architecture) from the Object Management Group (OMG), and Java. Java's inherent platform independence conveniently facilitates software reuse in heterogeneous environments. CORBA's language-neutral approach to object interface specification allows objects to interoperate without respect to implementation language. Java and CORBA, taken together, provide a solid architectural bedrock for developing highly reusable and portable distributed software systems.

Component-level reuse has emerged from the soup of competing computing paradigms, languages, and development environments to become the dominant theme in software development today. Objects participating in a distributed system can access data, business logic, and functional behavior directly through distributed components with little or no concern for how those components are implemented or where they reside on a network. With Java objects distributed through CORBA, you have a highly productive environment for component-based software development. Write it, deploy it, and use it over and over again.

This book provides an in-depth look at one specific incarnation of CORBA and Java—namely, VisiBroker® for Java from Inprise Corporation. Although we are intentionally focusing on the implementation details of one tool in this book, there is considerable benefit to be gained within these pages by anyone developing CORBA-based applications.

The Ailments of Information Systems

With the acceleration of change in software technologies, information systems (IS) departments are forced into a corner: Stand pat with the current technology and risk being left behind by the industry and competitors, or frequently adopt new technology in an attempt to maintain a strategic advantage over competitors but risk exposing the company to technological churning. Churning is the detrimental side effect of changing languages, tools, and technology so often that development teams cannot be productive. With each change, developers face a new learning curve. Applications, class libraries, and business rules oftentimes cannot be reused, and therefore must be rewritten for the new technology. Enabling applications written in diverse languages on multiple platforms to interoperate is paramount to maximizing the investment in existing applications and in improving the short-term productivity of development teams.

The problems that information systems face are well explained in the following excerpt from the introduction to the Object Management Group's *Discussion of the Object Management Architecture:*

> The major hurdles in entering this new world are provided by software: the time to develop it, the ability to maintain and enhance it, the limits on how complex a given program can be in order to be profitably produced and sold, and the time it takes to learn and use it. This leads to the major issue facing corporate information systems today: the quality, cost, and lack of interoperability of software. While hardware costs are plummeting, software expenses are rising.
>
> As information systems attain strategic importance and represent the key competitive edge to the industry leaders, the cost of inaccuracies or delayed implementations is attenuating entire MIS departments. As systems departments require information among a diversity of inhouse, brought-in, supplier, customer, and commercial applications, those applications become increasingly difficult and complex.

What Is CORBA?

Common Object Request Broker Architecture (CORBA) is an industry-wide standard for creating distributed object systems. It is a standard that is accessible from many different languages and allows interoperability on various platforms. CORBA is the Object Management Group's solution to the ailments just described. CORBA, in many ways, is the next step in client/server computing utilizing a multi-tiered, distributed-systems architecture.

What Is Distributed Computing?

In many business sectors today, client/server systems have become the solutions architecture of choice. Client/server systems are an example of a two-tiered model of distributed computing. Client/server systems deliver significant advantages in system design over traditional mainframe development, including the following:

- Sophisticated graphical user interfaces made possible through utilization of increased processing power on the client computer
- A way to distribute business and application logic to a user's computer
- Increased performance potential, as processing is distributed between server and client machines

Typical two-tier client/server systems are implemented such that the server is used for storing and retrieving data via some database management system (DBMS) and the client does everything else. This design is functional in many business scenarios, but it unfortunately has some limitations (detailed in Chapter 1). Many of the limitations of two-tiered client/server systems are solved by using multi-tiered, distributed computing systems—and specifically CORBA.

What is distributed computing? In the context of this book, *distributed computing* is the concept of using multiple-networked computers in cooperation to complete a business process. A well-designed distributed computing system will attempt to place the more complex and processor-intensive operations on the faster systems within that network. The World Wide Web is a successful example of a distributed computing system intended largely for human operators. The next step is to create a distributed computing system that can interoperate and communicate intelligently without always needing a human to guide its actions.

What Are Distributed Objects?

There are some other existing types of distributed systems that you may be familiar with:

- Remote procedure calls (RPCs)
- Socket-level programming
- Message queuing

Each of these is useful in its own way, but none of these is completely object-oriented (OO).

The term *object-oriented* has been the source of a good deal of confusion in the computer industry. Almost every software development environment on the market today claims to be object-oriented, but what does that really mean? Object orientation, at the most basic level, means using objects as the base construction block in a system. An object is typically considered to encapsulate data (variable attributes) and behavior (through methods or operations).

Object-oriented technology does hold much promise. The following is taken from the white paper "Distributed Objects for Business" by Jim Clarke, Jim Stikeleather, and Peter Finger of Technical Resource Connection Inc., and explains why OO is a unique and compelling approach:

> Object-oriented technology is based on simulation and modeling. Although this may be interesting in and of itself, the use of models represents a breakthrough in the way business information systems are developed. Instead of deploying the traditional application development life cycle, models of a business or business area

are constructed. These models are shared by individual computer applications. Essentially a "computer application" becomes a unique use of the model, not a separate development activity resulting in stand-alone software constructed for "this application only."

The quality of the model is a key determinant of "reuse" and adaptability. The model itself must be designed for change. Business processes change and their change is based on using the business model to simulate proposed processes. Modelers can play "what if," run simulations of various process alternatives, and learn from the simulations.

The focus of IS shifts from applications development to the enhancement and maintenance of common business models. Business models and software models become one and the same. Applications become derivatives, alternate views and refinements of the business models.

The modeling approach to business innovation is not possible without a software approach suited to the task. For these business reasons, object-oriented technology has become of vital interest to both commerce and industry. Business and technology must be fused if corporations are to maintain the competitive advantage. Object-oriented technology can be the foundation for that fusion. With object-oriented technology, change and the management of complexity are first-class concepts. Object technology holds great promise as a means of designing and constructing the adaptive information systems needed for 21st century business.

Objects and object-oriented development represent a significant shift in systems design. One of the benefits of object-oriented design is that objects tend to model artifacts that exist in the real world. An object may be a user, another computer system, or part of a process, but it is something that has a meaning tied to the real world.

Another important eventuality that comes out of OO is this: As reuse increases, software costs will decrease and reliability will increase. With a strong collection of business objects in place, creating a new application will no longer mean building from the ground up; it will be more akin to linking objects together in a new and useful way. This is where components enter the discussion. A component is really a prepackaged collection of objects that a developer can treat as a single object. Components offer greater potential for reuse because they abstract many of the complexities of the lower-level constituent objects.

But what is a distributed object system? A distributed object is an object that can be accessed as if it were a local object, although its actual location may be local or remote. This powerful concept allows a system to leverage the power of any and all CPUs in its network. Today, in local object systems (client/server is an example of this), your application must contain all of the code for every operation that your application might someday perform. From data access to printing, it all must be coded in your system.

Here is a description of a system that should help to convey the powerful difference between a distributed object system and a traditional local object system (client/server).

Imagine your PC is currently connected to a network and is running a personal finance application. Imagine that you are late for a meeting, so you are trying to do many operations quickly. You are trying to complete the following: (1) get a current valuation of your investment portfolio; (2) compute your capital gains tax for the past year; and (3) print off your tax forms from last year.

In the client/server world, your application would have to know how to do all three of these functions. To get current stock prices, your app must connect directly to a pricing feed. To compute the capital gains tax for the year, your app must be programmed with the current tax law's formula. To print, your application must know how to talk to a print driver and send each piece of information to a printer.

In the distributed object world, your client does not have to know how to do any of this. It need only know how to call a distributed object that can perform this function for you. To get current stock prices, you can connect to an object server that we can query for current prices. This is more flexible because now if we need to switch pricing feeds to a new, more accurate and timely feed, we need only change our object server. We do not need to change a single thing in our distributed system client, as it calls and invokes the same operation.

Similarly, in trying to compute your capital gains tax for the past year, our distributed-system client need only invoke this operation in a tax object server. Our client needs to know nothing about how this value is computed. Again, this is more powerful since we can easily change this formula as the tax laws change, without having to modify our clients.

Printing is handled in much the same way. All our distributed-system client need do is send the documents to a print server to have them printed. All of the logic associated with negotiating with the printer resides in the distributed object and not in the client.

Of course, this does not release you from all the drudgery of having to write those distributed objects. But what this design does allow you to do is to create a solid separation between your client and the business logic used by your client. The business logic, when implemented in distributed objects, is now readily reusable by any application that may need similar functionality.

The print server mentioned previously is an obvious example of how distributed-object functionality could be used in many applications.

There is at least one other subtle difference that should be pointed out: Client/server applications tend to be very synchronous in nature. Therefore, you most likely would have had to invoke each of these operations in a sequence, waiting for each one to finish before moving along to the next. Distributed object systems are more easily architected to be asynchronous, which means you could have invoked all three of these operations immediately in any sequence, without waiting for any operation to complete.

One of the greatest promises in OO systems is reuse. Reuse does exist, but has eluded many IS shops. By and large, the IS shops that have seen reuse have seen it only within a single development environment. It is an unfortunate fact that the zealous pursuit of reuse has forced many IS shops to choose a single development environment and stick with it, even when it may have outlived its usefulness. CORBA offers a solution to this problem by offering language-independent object mappings. CORBA allows object-level communication with many of the most popular languages today. This language-independent approach facilitates reuse between tools. This feature is obviously a great benefit when properly utilized.

A Word about Standards . . .

One of the difficulties with any new technology is that standards take a while to develop after the technology has been released. Standards, when properly applied, achieve many of the goals of most software organizations, such as lower implementa-

tion costs, increased reuse, and so forth. CORBA represents an industry-wide standard for distributed object computing. As the CORBA standard (and others) is increasingly embraced by the software industry, greater reuse and connectivity between in-house and purchased systems will become possible.

CORBA: A Standard for Distributed Systems

CORBA is an industry-wide standard for creating distributed object systems. CORBA has been defined by the Object Management Group (OMG), a nonprofit consortium of companies whose only goal is to facilitate the definition of standards for interoperable software. This is truly a revolutionary approach to defining standards, as the primary arbiters of standards in the past have been the companies with the best marketing campaigns. The OMG writes no software; it only facilitates the definition of standards. Any vendor who implements these standards will have created software that, by definition, is interoperable with all software implemented according to the same standards.

The OMG was created in 1989 by eight charter members: 3Com Corporation, American Airlines, Canon, Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems, and Unisys Corporation. It now boasts the support of over 700 companies around the globe, the world's largest consortium of this type. Notably, Microsoft was initially at odds with CORBA and the OMG, but it seems that they have largely settled their differences and intend to peacefully coexist.

CORBA is a significant part of the OMG's Object Management Architecture. CORBA continues to be extended by the addition of CORBA Services and CORBA Facilities.

The Object Request Broker is significantly different from the two-tiered client/server paradigm. Client/server systems typically only pass data back and forth between server and client. An ORB system can pass functionality from client to client and server to server via objects.

As a distributed object standard, what advantages does CORBA provide?

- CORBA supports many programming language mappings. CORBA also allows for the mixing of these languages within a distributed system (i.e., a system could consist of C++, Java, and COBOL-implemented object servers, all communicating fluently).
- CORBA supports distributed computing *and* object orientation.
- CORBA is an industry standard backed by over 700 companies.
- CORBA provides interoperability between different vendors' ORBs.

So what does CORBA really do? Imagine an environment where, for example, you could use your desktop PC to interoperate with software objects located anywhere within your connected network (which can be pretty large if you include the Internet) as if they were local to your machine. This is a concept known as *location transparency.* These objects could be served up from any hardware platform that is running a CORBA server, which can be implemented in virtually any programming language. This concept is known as *programming language transparency.*

CORBA will allow you to leverage existing legacy code on any platforms that support the standard, which include several mainframes. This may be the first technology that will allow your company to move forward without having to rewrite every function in your standard library.

Why Was This Book Written?

If you survey the bookshelf at your local bookstore, you will see that several CORBA titles are available. Most are general surveys of CORBA. Others are books written about CORBA in general, but do not discuss any specific product. This book is one of the first books written about an ORB from a specific vendor: Inprise.

This book is designed to help any developer rapidly become proficient in the critical aspects of CORBA development. The knowledge passed along here is intended to help you design better distributed systems and implement them faster.

Why VisiBroker?

Although CORBA has been around for several years, its popularity has skyrocketed in the past 12 to 18 months. This is partly attributable to the popularity of the World Wide Web (an accessible distributed environment) and, to a lesser extent, Java, which has reduced the complexities in programming distributed object servers.

Since late in 1996, many companies have begun to build in support for CORBA. Many companies have licensed VisiBroker to integrate into their development and deployment environments, and, in some cases, their core products. It is extremely likely that you are using a product today that will shortly, if it doesn't already, have VisiBroker integrated into it. This book is intended to show you not only how to use VisiBroker as a stand-alone product, but also how to integrate it with products from other vendors.

What Industry Support Is There for CORBA and VisiBroker?

The following will give a short description of the type of CORBA integration each company is undertaking.

Netscape: Netscape has licensed VisiBroker for C++ and Java to use as, among other things, a powerful server-side scripting tool and embeds the client portion of the ORB in every browser.

Sun: Sun includes CORBA support in the Java JDK 1.2 as part of their "enterprise java beans" strategy. Sun also uses VisiBroker in its Solstice network management products.

Hewlett-Packard: HP integrates VisiBroker in its Smart Internet Usage product of ISPs.

Oracle: Oracle, which has also licensed VisiBroker, is using CORBA across the board at the very core of its database management system.

Cisco: Cisco uses VisiBroker in its enterprise network management solutions.

Telcordia: Telcordia integrates VisiBroker in its Next Generation Network solutions.

Hitachi: Hitachi builds many products including TPBroker® product suite and Cosminexus Application Server, which both use VisiBroker.

Sabre: Sabre will use VisiBroker as part of its application development and management framework.

Ericsson: VisiBroker will form the basis of the future architecture of Ericsson's OSS products for managing its GSM and broadband CDMA networks.

Sybase: Sybase has announced support for CORBA as part of its next-generation middleware product, called Jaguar. Sybase has also built CORBA-compliant support into its Powerbuilder Development Tools.

Inprise™: Inprise itself integrates VisiBroker with its IDE products such as Borland® JBuilder™, Borland C++ Builder™ and Delphi™. At the enterprise level, VisiBroker for Java forms the foundation of the Inprise Application Server™.

Why Java and CORBA?

We have elected to use Java as the core programming language throughout this book. There are several reasons for this choice:

Portability: As you well know, one of the most compelling reasons to use Java is its multiplatform support via the Java virtual machine. This allows us to write a book that will be applicable to all Java-supported platforms.

Extends development and deployment options for CORBA: By using the portability of Java, it becomes possible to develop your distributed object system on a given platform, say Windows NT, and deploy to a different platform, say SUN OS, with relative ease. This is not so easily accomplished with other programming languages.

Simplifies garbage collection: One of the biggest problems in ORBs (and other distributed systems generally) implemented in other languages is garbage collection. In an ORB, situations can arise where a once-used distributed object no longer has any clients referencing it. At this point the distributed object should be garbage collected and destroyed. In C++, you must write a separate process to perform the garbage collection. In Java, because garbage collection is built into the language, this process is greatly simplified.

Multithreading: Your systems will see a significant performance gain if you properly multithread your servers (and, in some cases, your clients). Since multithreading is nicely integrated into Java, it is more readily available for use.

Readability: Java code is relatively easy to read and understand, as it is not bogged down with the complexities of C++. This helps to make the code examples in this text easier to understand. However, all of the examples implemented in this book can be implemented in C++, and most examples could be implemented in any language for which CORBA mappings exist.

How This Book Is Organized

The book has been divided into three sections:

- **Part One. An Introduction to Distributed Systems in CORBA.** Section 1 includes all the information necessary for a developer to reach a basic competency level in CORBA. We've included a chapter on analysis and design of distributed systems (Chapter 5) with the intent of sharing information on analysis and design techniques that we've found to be effective in CORBA. All of the information in Section 1 is meant to be specific and brief, as we expect most readers will be already somewhat familiar with this content.
- **Part Two. Working with VisiBroker.** Section 2 includes the topics that are necessary for an intermediate to advanced CORBA developer. This section has extensive coverage of the most important aspects of implementing VisiBroker servers and clients.
- **Part Three. Advanced Topics in VisiBroker.** Section 3 includes more detailed descriptions of more complex features that will be important for specific advanced uses of VisiBroker. Topics will include CORBA Services, the POA and how to take advantage of it, the interface repository, and ORB interoperability.
- **Appendixes**
  - Appendix A: VisiBroker for Java Quick Reference: Interfaces and Properties
  - Appendix B: VisiBroker for Java Quick Reference: Commands and Utilities
- **The Companion Web site.** This book comes with a companion Web site hosted at www.wiley.com/compbooks/natarajan. This site contains:
  - Source code for all examples in this book
  - Pointers to newsgroups, specs, and other places where you can find out more about VisiBroker and CORBA
  - Links to useful resources
  - Errata for the book, if any

What's New in This Edition?

The second edition of this book covers all of the new features introduced in VisiBroker for Java 4.0:

- **Portable Object Adapter.** The Portable Object Adapter replaces the deprecated Basic Object Adapter as an Object Adapter. This adapter (as the name implies) provides users with an API to program servers portably across multiple ORBs. See Chapter 14 for details.
- **CORBA Valuetypes.** CORBA valuetypes allow the transfer of state much like struct but with a more flexible type system, with features such as inheritance, operations, and attributes. Chapter 15 describes valuetypes in detail.

- **Interoperable Naming Service.** The Interoperable Naming Service specification introduced many new features, such as the URL-based mechanism to define initial references to the naming service, and describes the more interesting features of the naming service that comes with the VisiBroker product, such as clustering. Chapter 19 describes the naming service and its features.
- **VisiBroker Console.** This is a graphical management tool for VisiBroker servers and services. This tool is introduced in Chapter 9, and details specific to each service are described in the appropriate chapters.
- **Protocol Engine.** This is a powerful new framework that allows servers to publish multiple endpoints for the object reference, with different properties. Details of the Protocol engine and how to configure it can be found in Chapter 16.
- **Interceptors and Object Wrappers.** Interceptors are a framework of callbacks that allow users to monitor the progress of different lifecycles within the ORB. This includes requests, object binding, object creation, POA creation, and IOR creation. Object Wrappers are like interceptors for requests but work at the application level, with the ability to affect the behavior of a request based on input parameters for the operation. The Interceptors and Object Wrappers are described in detail in Chapter 18.
- **Firewall support.** The gatekeeper and its enhanced support for firewall navigation allows very flexible and easy mechanisms to negotiate firewalls.

All the examples have been updated to work with the new POA model introduced in VisiBroker 4. The interceptor examples use the Visibroker 4 interceptor framework.

At the time of writing, the security service framework for VisiBroker was not shipping, so we do not discuss it in this edition. You can find updated information on the security service for VisiBroker on the Inprise Web site at www.inprise.com/security.

Compatibility

All source code examples work with VisiBroker for Java release 4.0 or later versions. For simplicity, however, we refer to the software throughout the book as VisiBroker for Java 4. If there is anything you need to know about a specific dot release of this software, we explain it in the text. *Programming with VisiBroker*, *Second Edition*, has been tested against JDK 1.2.2. Please see the companion Web site for this book for updates to the code.

Onward!

So let's get on with it! The pages that follow will give you a brief overview of CORBA and then enable you to build advanced distributed object systems in VisiBroker for Java. This book will become a valuable reference as you integrate VisiBroker for Java into your standard development environment.

About the Authors

Vijaykumar Natarajan is an architect for the VisiBroker products at Inprise Corporation, contributing to both the VisiBroker for Java and C++ products. Vijay came to Inprise through the Visigenic acquisition and worked at Informix prior to that. Vijay is also an active member of the OMG, helping to evolve the CORBA specifications as a representative of Inprise.

Stefan Reich is a senior software engineer at Inprise Corporation, contributing to the VisiBroker and AppServer product line. Stefan joined Inprise Corporation from the University of Hamburg, Germany, where he was conducting cutting-edge research on load balancing techniques using CORBA. While working on his diploma at Hamburg, he also consulted with various German companies to provide Internet and intranet e-commerce solutions.

Bhaskar Vasudevan currently works as an architect at @ztec software, building cutting edge e-commerce solutions for Bay Area companies. Prior to this, he was the Technical Lead for the VisiBroker for C++ product. Bhaskar also came to Inprise through the Visigenic acquisition. Prior to Visigenic, Bhaskar worked for Oracle Corporation.

Authors of the First Edition

Doug Pedrick is the Lead Technical Architect at Strong Capital Management in Milwaukee. He is responsible for the architecture and design of portfolio management and trading systems. Doug's current focus is on enterprise application integration and distributed object oriented systems using J2EE, EJB, JMS, CORBA, and XML.

Jonathan Weedon is Principal Engineer and Lead Architect of the Enterprise division of Inprise. During his five years at Inprise/Borland/Visigenic/Post Modern, he has been the principal developer of VisiBroker for Java (versions 1, 2 and 3), and the EJB Con-

tainer. He has played an architecture role on both Inprise Transaction Service (ITS) and Application Server (IAS). Jonathan and family split their time between Half Moon Bay and Penn Valley, California.

Jon Goldberg joined Tacit Knowledge Systems in 1999 as a Founding Member of Technical Staff. He is currently focused on Java and database development for the KnowledgeMail product line. Jon lives in San Francisco.

Erik Bleifield is currently the primary architect and development manager of middleware and integration solutions at a major banking and business processing outsourcing company.

The Contributors

Gopal Ananthraman is a Research and Development Engineer contributing to the VisiBroker products. Gopal would like to thank Ke Jin (architect) for his support and useful feedback and his management team. He would also like to thank Stefan for reviewing the chapter and providing useful comments.

Tomasz Mariusz Mojsa holds the position of Principal Consultant for Inprise Corporation, architecting, designing, and implementing strategic enterprise distributed systems for major US corporations. Tomasz would like to thank Vijay and Stefan, his close friends at Borland/Visigenic, whose book he was happy to contribute to, and for the great days (including some Saturdays and Sundays) spent working on distributed systems together. Without their hard work the book would never have come into existence.

Ioana Pirvulescu is the technical lead for the VisiBroker for C++ product, contributing to both the Java and C++ VisiBroker ORBs. Ioana would like to thank Vijay and Stefan for involving her in this project, Bhaskar for being a good friend and a great colleague, the VisiBroker team for the many hours of hard work together, and of course Mike, for being by her side and often covering for her on the wedding planning front!

Programming with VisiBroker®

Part

One

An Introduction to Distributed Systems in CORBA

CHAPTER

1

Introduction to CORBA

This chapter will discuss the essential parts of a CORBA (Common Object Request Broker Architecture) system and some design goals useful for a successful CORBA environment. This material is essential to understanding any CORBA system and will be referred to throughout the book. We have intentionally left out much of the theoretical background on which CORBA is based, believing that this material has been sufficiently covered elsewhere. For more information on CORBA, check out *Client/Server Programming with Java and CORBA* by Robert Orfali and Dan Harkey (Wiley, 1998). Our intent here is to give readers a quick description of the terminology and functional areas involved in any CORBA development effort so they can immediately begin to grasp the essentials in CORBA system design.

Distributed Object Computing

In many ways CORBA is the next evolution of the client/server paradigm. Client/server topologies came about in the 1980s in response to the need to share centralized data with a large number of end users, all employing the processing power of increasingly powerful desktop computers. Typical two-tier client/server software architecture partitions functionality in such a manner that the client application performs both business processing and user interface operations. The server is used as a data manager: a file server or database repository. The architectural limitations and development pitfalls of this approach have become apparent. Software maintenance in a two-tier architecture is tedious at best. Because the user interface makes direct calls to the database, changes to the database have a widespread impact. Domain-level reuse is nearly impossible

because the rules of the business are so tightly coupled with presentation and data storage logic.

The application of sound software engineering principles has resulted in the separation of responsibilities into three areas, or tiers: presentation, business logic, and data storage and retrieval. The logical separation and loose coupling of tiers isolates each tier from change in the other tiers. The third tier, encapsulating the business and application logic, has become known as the *middle tier*. Services to help support the implementation of the middle tier have become known as *middleware*.

Middleware has very different meanings to different users (and seemingly to many vendors). Unfortunately, it is too often associated solely with database access software, completely losing sight of what we believe is the most significant purpose of the middle tier. It is paramount that any software tool touted as middleware increase developer productivity and reduce (or abstract) system complexity. We believe that good middleware will be defined by several other important features: It must allow for clear separation of business logic; it must aid in supporting reuse; it must be standards-based; it must exhibit high reliability and availability; and, to be part of an enterprise's strategic architecture, it must be highly scalable.

Also, good middleware must support interoperable objects, implemented in various languages, living on different types of platforms, located on a network. A client and a server implemented using object-oriented languages that cannot directly invoke operations transparently, regardless of where those objects actually live, is not a distributed object system. Prior to the advent of CORBA, many Information Systems (IS) shops implemented client/server systems that used object-oriented languages on all nodes, but had to either define their own communication protocols or resort to sockets and remote procedure calls to carry out the desired behavior. Of course, remote invocations were not done directly on objects, and their targets were anything but transparent.

CORBA: Object-Oriented Middleware

The marriage of the object-oriented paradigm to a client/server topology, with the intention of facilitating the interaction of objects in a client/server relationship, has given rise to CORBA. An industry consortium, the Object Management Group (OMG), which now numbers over 700 members, created the CORBA standard as an answer to the need for distributed object interoperability. CORBA is the heart of the OMG's architectural framework, the Object Management Architecture (OMA), which will be discussed shortly.

CORBA is superior to other middleware products for many reasons, not the least of which is that it is a nonproprietary, industry-supported standard. Other benefits of CORBA include the following:

- It forces the separation of an object's interface and its implementation.
- It is scalable.
- Support for reuse is inherent.
- There is language and platform transparency.
- It provides vendor independence through interoperability.
- CORBA Services provide à la carte functionality.
- Network communication is abstracted from the developer.

CORBA clients and servers are developed against a common interface specification, written in the OMG's Interface Definition Language (IDL), which is essentially a contract between a server and potential clients. The IDL for a server specifies the interfaces, or objects, attributes, and operations available for that interface. The IDL file is compiled, and supporting files are created that map the IDL specification to a target implementation language. For instance, the VisiBroker® for Java IDL to Java compiler generates several Java source files for each IDL interface. Some of these source files are used by the object implementation (the server) and some by the client. The client and server need not be developed in the same language. One IDL file may be compiled into different implementation languages—a server written in Java doesn't know, or care, that a client was written in C++. The client and server communicate via an Object Request Broker (ORB), which is the core of the CORBA middleware architecture. The client does not have to be aware of the object's location, the network protocol used, the language used to implement the object, or the operating system hosting the server. The only aspects the client has knowledge of are those specified in the IDL interface.

So what is happening under the covers? The IDL compiler, ORB, and object adapters conspire to abstract the complexities of distributed object communication. The key players in this game follow.

Object Request Broker

The ORB is the heart of any CORBA implementation. It is responsible for enabling objects to transparently make requests and receive responses in a distributed environment—whether that environment is a heterogeneous or a homogeneous system of computers and networks. Because the ORB assumes responsibility for so much object management, and because the mechanisms of routing invocations to their target objects have been abstracted so completely, the client applications are relatively simple. To the client, it appears as though every object is always active, even though that's not the case. CORBA does not even provide a separate command for a client to start up an object implementation—the client just sends a request, and the ORB does everything else. With the exception of some ORB initialization requirements, a typical client views the distributed system solely through the IDL interface specifications, and the client is completely divorced from the implementation details.

Object Adapter

An Object Adapter (OA), in the CORBA sense, is a logical set of server-side facilities that serves both to extend the functionality of the ORB and to provide a mechanism for the ORB and the object implementation to communicate with each other. Rather than bundling this functionality into the ORB Core, adapters can be used to offer specialized services that have been optimized for a particular environment, platform, or object implementation. The OA is layered on top of the ORB Core to provide an interface between the ORB and the object implementation. A typical OA provides services such as the following:

- Registration of servers (implementations)
- Activation and deactivation of object implementations

- Instantiation of objects at run time and the generation and management of object references
- Mapping of object references to their implementations
- Dispatching of client requests to server objects via a static skeleton or DSI (Dynamic Skeleton Interface)

While many types of object adapters are possible for unique situations, the CORBA specification only requires implementations to provide a Portable Object Adapter (POA). The adoption of the Portability Specification called for the deprecation of an earlier specification for an object adapter called the Basic Object Adapter (BOA), which was woefully underspecified in favor of the POA.

Interface Definition Language (IDL)

The OMG Interface Definition Language (IDL) defines types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters that apply to those operations. IDL is the means by which a particular object implementation informs its potential clients of the operations available and the way to invoke them. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

An object implementation provides the semantics of the object, usually by defining data to represent the state of the object instance and code (methods) to implement the object's behavior. Each IDL interface is ultimately implemented in code, and that code is collectively called an *object implementation.* In Java, this typically corresponds to one (though it need not be limited to one) Java class. To fully implement the object, the class will often use other objects and classes and define variables and methods not part of the IDL interface.

Clients and Servers

Quite simply, a client is any entity that issues requests for the services of an object. An entity plays the role of a client relative to a particular object. An object that assumes the role of client in one invocation may in turn respond to requests for services from other objects. A client of an object must have an invocable reference—also known as an object reference—to that object.

Typically, the term server is used to describe an executable program or a specific process executing that program. The server contains one or more object implementations—that is, a Java class that implements the operations corresponding to an IDL interface.

Internet Inter-ORB Protocol (IIOP)

The goal of ORB interoperability is to allow communication between independent implementations of the CORBA standard. ORB interoperability allows a client of one vendor's ORB to invoke operations on an object in a different ORB. Invocations between client and server objects are independent of whether they are on the same or different ORBs. To

make this happen, all ORBs must communicate via a standard protocol. The OMG protocol for ORB interoperability, the General Inter-ORB Protocol (GIOP), defines the on-the-wire data representation and message formats for all inter-ORB communication. The OMG also defined a specialization of GIOP, called the Internet Inter-ORB Protocol (IIOP), that uses TCP/IP as the transport layer. Specialized protocols for other transports are expected to be defined in time. All compliant ORBs are required to at least provide support for IIOP.

Distributed Object Computing: The Object Management Architecture (OMA)

The OMA is a larger framework within which all OMG-adopted technology resides. It provides two basic models on which CORBA and other standard interfaces are based: the Core Object Model and the Reference Model.

The Core Object Model defines the concepts that allow distributed application development to be facilitated by an ORB. It describes the theoretical basis of CORBA. The Core Object Model is an abstract definition that does not attempt to detail the syntax of object interfaces or any other part of an ORB. It also defines a framework for refining the model into a more concrete form. The model provides the basis for CORBA, but is more relevant to ORB designers and implementers than to distributed object application developers. It is thoroughly described in the OMG's Object Management Architecture Guide and will not be dealt with at any level in this book.

The Reference Model places the ORB at the center of groupings of objects with standardized interfaces that provide support for application object developers. The following groups are identified: Object Services, which provide infrastructure; Domain Interfaces, which provide special support to applications from various industry domains; Common Facilities, which provide application-level services across domains; and Application Interfaces, which is the set of objects developed for a specific application.

The Reference Model is important to CORBA developers, because it presents a development model through which developers can create and use frameworks, components, and objects. This book will focus mostly on a subsection of the Reference Model (Figure 1.1), Application Objects, although we will spend some time talking about Object Services in Chapters 19 and 20.

Object Services

The Object Services offer fundamental services for use by the developers of implementation objects. Among other things, the object-level functionality specified by these interfaces provides services to store, manage, and locate objects, to enforce relationships between objects and groups of objects, and to provide frameworks for licensing and security. The Object Services implemented by VisiBroker for Java, Naming, and Events, are detailed fully in Chapters 19 and 20. The published services include:

Naming. The Naming Service provides the ability to bind a name to an object relative to a naming context. A naming context is an object that contains a set of name bindings, in which each name is unique. To resolve a name is to determine the object associated with the name in a given context. Through the use of a very

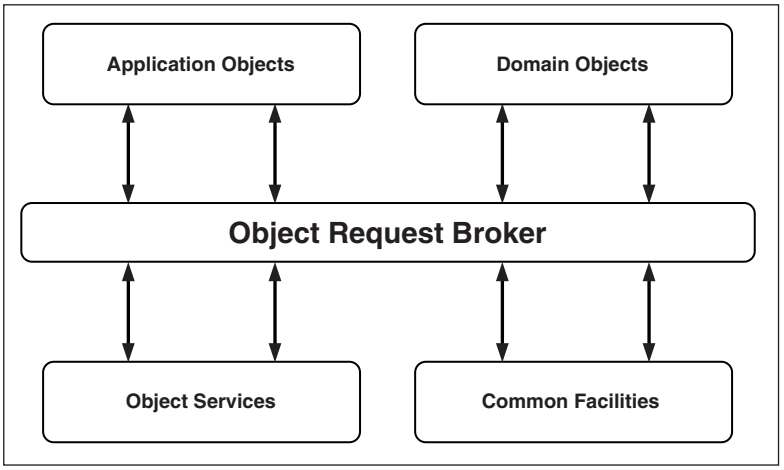

Figure 1.1 The OMA Reference Model.

general model and in dealing with names in their structural form, Naming Service implementations can be application-specific or based on a variety of naming systems currently available on system platforms.

Events. The Event Service provides basic capabilities that can be configured together flexibly and powerfully. The service supports asynchronous events (decoupled event suppliers and consumers), event fan-in, notification fan-out, and, through appropriate event channel implementations, reliable event delivery.

The Event Service design is scalable and is suitable for distributed environments. There is no requirement for a centralized server or dependency on any global service. Both push and pull event delivery models are supported; that is, consumers can either request events or be notified of events.

Persistent Object Service. The Persistent Object Service (POS) provides a set of common interfaces to the mechanisms used for retaining and managing the persistent state of objects. The object ultimately has the responsibility of managing its state, but it can use POS or delegate the actual work to it. A major feature of the Persistent Object Service (and the OMG architecture) is its openness. In this case, that means that there can be a variety of different clients and implementations of the Persistent Object Service, and they can work together. This is particularly important for storage, whereby mechanisms useful for documents may not be appropriate for employee databases, or mechanisms appropriate for mobile computers do not apply to mainframes.

Relationships. The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two new kinds of objects: relationships and roles. A role represents a CORBA object in a relationship. The Relationship interface can be extended to add relationship-specific attributes and operations. In addition, relationships of arbitrary degree can be defined. Similarly, the Role interface can be extended to

add role-specific attributes and operations. Type and cardinality constraints can be expressed and checked. Exceptions are raised when the constraints are violated.

Lifecycle. The Lifecycle Service defines operations that copy, move, and remove graphs of related objects, while the Relationship Service allows graphs of related objects to be traversed without activating the related objects. Distributed implementations of the Relationship Service can have navigation performance and availability similar to CORBA object references: Role objects can be located with their objects and need not depend on a centralized repository of relationship information. Therefore, navigating a relationship can be a local operation.

Externalization. The Externalization Service defines protocols and conventions for externalizing and internalizing objects. To externalize an object is to record the object state in a stream of data (in memory, on a disk file, across the network, and so forth); it can then be internalized into a new object in the same or a different process. The externalized object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and be internalized in a different, disconnected ORB. For portability, clients can request that externalized data be stored in a file whose format is defined with the Externalization Service Specification.

The Externalization Service is related to the Relationship Service and parallels the Lifecycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for facilities, directory services, and file services.

Transactions. The Transaction Service supports multiple transaction models, including the flat (mandatory in the specification) and nested (optional) models. The Transaction Service supports interoperability between different programming models. For instance, some users want to add object implementations to existing procedural applications and to augment object implementations with code that uses the procedural paradigm. To do so in a transaction environment requires the object and procedural code to share a single transaction. Network interoperability is also supported, since users need communication between different systems, including the ability to have one transaction service interoperate with a cooperating transaction service using different ORBs.

Concurrency Control. The Concurrency Control Service enables multiple clients to coordinate their access to shared resources. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.

Licensing. The Licensing Service provides a mechanism that allows producers to control the use of their intellectual property. Producers can implement the Licensing Service according to their own needs and the needs of their customers because the Licensing Service does not impose its own business policies or practices.

Query. The Query Service allows users and objects to invoke queries on collections of other objects. The queries are declarative statements with predicates and include the ability to specify values of attributes, to invoke arbitrary operations, and to invoke other Object Services.

Properties. The Property Service provides the ability to dynamically associate named values with objects outside the static IDL-type system. It defines opera-

tions to create and manipulate sets of name-value or name-value-mode tuples. The names are simple OMG IDL strings. The values are OMG IDL anys. The use of type Any is significant in that it allows a Property Service implementation to deal with any value that can be represented in the OMG IDL–type system.

Security. The Security Service comprises the following:

- Identification and authentication of principals (human users and objects that need to operate under their own rights) to verify that they are who they claim to be
- Authorization and access control—deciding whether a principal can access an object, normally using the identity and/or other privilege attributes of the principal (such as role, groups, security clearance) and the control attributes of the target object (stating which principals, or principals with which attributes can access it)
- Security auditing to make users accountable for their security-related actions. It is normally the human user who should be accountable. Auditing mechanisms should be able to identify the user correctly, even after a chain of calls through many objects.
- Security of communication between objects, which often involves insecure lower-layer communications. This requires trust to be established between the client and the target, which may require authentication of clients by targets and authentication of targets by clients. It also requires integrity protection and (optionally) confidentiality protection for messages in transit between objects.
- Nonrepudiation, which provides irrefutable evidence of actions such as proof of origin of data for the recipient or proof of receipt of data for the sender, to protect against subsequent attempts to falsely deny receiving or sending data.

Administration of security information (for example, security policy) is also needed.

Time. The Time Service enables the user to obtain the current time together with an error estimate associated with it. It ascertains the order in which events occurred and computes the interval between two events.

Collections. The Collections Service provides a uniform way to create and manipulate the most common collections. Collections are groups of objects that support some operations and exhibit specific behaviors that are related to the nature of the collection rather than to the type of object they contain. Examples of collections are sets, queues, stacks, lists, and binary trees.

Trading. The Trader Service provides a matchmaking service for objects. The service provider registers the availability of the service by invoking an export operation on the trader and passing as parameters information about the offered service. The export operation carries an object reference that can be used by a client to invoke operations on the advertised service, a description of the type of the offered ser-

vice (that is, the names of the operations to which it will respond, along with their parameters and result types), and information on the distinguishing attributes of the offered service.

Within the common object services, the following design guidelines have been employed:

- **Basic, flexible services.** Each service is designed to perform its main function well and therefore is only as complicated as it needs to be.
- **Generic services.** Services are designed to be generic in that they do not depend on the type of the client object or, in general, on the type of data passed in requests.
- **Allowance for local and remote implementations.** This guideline reinforces the concept of total location independence.
- **Quality of service as an implementation characteristic.** Service interfaces are designed to allow a wide range of implementation approaches, depending on the quality of service required in a particular environment. For example, in the implementation of a particular service, a channel could be implemented to be fast with unreliable delivery of information, or to be slow, with guaranteed delivery.
- **Use of callback interfaces.** Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to call back to it to invoke some operations. The callback may be, for example, to pass back data asynchronously to a client.

Common Facilities

Whereas the Object Services provide functionality for use by *objects*, Common Facilities provide standards for services aimed at *applications*. Also known as CORBA facilities, this is an in-progress effort that intends to create a set of interfaces to provide generic functions needed by many applications. Facilities such as printing, document management, and e-mail have been proposed.

Domain Interfaces

These interfaces will provide domain-specific objects for vertical application domains such as finance, healthcare, manufacturing, telecom, electronic commerce, and transportation. When this work is completed, it should provide a solid basis for building industry-wide interoperable software.

Application Objects

This part of the architecture represents those application objects that perform specific tasks for users. This is the area where developers will be doing most of their work, and this is, of course, the area of focus for most of this book.

Sending and Receiving Requests

A *request* refers to a client invoking an operation on an object residing on a server. In order to do this, the client's request is handled locally by a programming construct (a Java class in the case of Java) known as a *stub* (also called a *proxy*). Client stubs and server skeletons are generated by the IDL-to-language compiler. It appears to the client that the stub is the actual target object, when in reality the stub is used as a placeholder for the remote object. The stub and the ORB cooperate to marshal any parameters and transmit the request to the remote object. An instance of the skeleton is waiting on the remote system for the client's request. The skeleton (and the ORB) unmarshals the arguments, executes the requested operation, and creates a reply if necessary.

Stubs and Skeletons

There is a stub for every interface type. The stub presents access to the OMG IDL–defined operations on an object in a way that is easy for programmers to predict once they are familiar with OMG IDL and the language mapping for the particular programming language. The stubs make calls on the rest of the ORB using interfaces that are private to, and presumably optimized for, the particular ORB Core. If more than one ORB is available, there may be different stubs corresponding to the different ORBs. In this case, it is necessary for the ORB and language mapping to cooperate to associate the correct stubs with the particular object reference. Java provides a standard set of interfaces that allows ORB vendors to generate portable stubs, which will run on any ORB.

On the server side, for a particular language mapping, and possibly depending on the object adapter, there will be an interface to the methods that implement methods supported by each type of object. This is known as the skeleton. The interface will generally be an up-call interface, in that the object implementation comprises of routines that conform to the interface and the ORB calls them through the skeleton. The skeleton is the bridge between the ORB and the actual code that implements the methods associated with the object's interface.

The existence of a skeleton does not imply the existence of a corresponding client stub (clients can also make requests via the Dynamic Invocation Interface).

Dynamic CORBA

An interface is available that allows the dynamic construction and dispatch of object invocations. This functionality is called the Dynamic Invocation Interface (DII). Rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a sequence of calls. The client code must supply information about the operation to be performed and the types of the parameters being passed (perhaps obtaining it from an Interface Repository or other run-time source). The nature of the Dynamic Invocation Interface may vary substantially from one programming language mapping to another.

On the server side, an interface exists that allows dynamic handling of object invocations. This is called the Dynamic Skeleton Interface (DSI). The DSI allows an object's implementation to be reached through an interface that provides access to the operation name and

parameters in a manner analogous to the client side's Dynamic Invocation Interface, rather than relying on a specific skeleton to access an operation's implementation. Purely static knowledge of those parameters may be used, or dynamic knowledge (perhaps determined through an Interface Repository) may be used to determine the parameters.

The implementation code must provide descriptions of all the operation parameters to the ORB, and the ORB provides the values of any input parameters for use in performing the operation. The implementation code provides the values of any output parameters, or any exceptions, to the ORB after performing the operation. The nature of the dynamic skeleton interface may vary substantially from one programming language mapping or object adapter to another, but will typically be an up-call interface.

Dynamic skeletons may be invoked through both client stubs and the dynamic invocation interface; both styles of client-request construction interfaces provide identical results.

CORBA without IDL

Although IDL provides the conceptual framework for describing the objects offered by a particular server, the availability of IDL source code is not a necessity for a client to interoperate with remote objects. As long as the equivalent information is available in the form of stub routines or a run-time interface repository, a particular client may be able to function correctly. Inprise's java2idl compiler uses this characteristic of CORBA to generate client stubs and server skeletons directly from Java source code rather than IDL.

Interface and Implementation Repositories

The Interface Repository is a service that provides objects that represent the IDL information in a form available at run time. The Interface Repository information may be used by the ORB to perform requests. Moreover, by using the information in the Interface Repository, it is possible for a program to determine what operations are valid on an object whose interface was not known when the program was compiled, and to make an invocation on it.

In addition to its role in the functioning of the ORB, the Interface Repository is a common place to store additional information associated with interfaces to ORB objects. For example, debugging information, libraries of stubs or skeletons, or routines that can format or browse particular kinds of objects might be associated with the Interface Repository.

The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. Although most of the information in the Implementation Repository is specific to an ORB or operating environment, the Implementation Repository is the conventional place for recording such information. Ordinarily, installation of implementations and control of policies related to the activation and execution of object implementations are done through operations on the Implementation Repository.

In addition to its role in the functioning of the ORB, the Implementation Repository is a common place to store additional information associated with implementations of ORB objects. For example, information on debugging, administrative control, resource allocation, and security might be associated with the Implementation Repository.

CORBA System Design

Critical analysis and design issues for developing successful CORBA systems are mentioned throughout this book, and they are examined in detail in Chapter 5.

Perhaps the two most important issues have nothing to do with CORBA per se:

- Your IS group should create a development process and strive to continually improve it. A company without a development process is unlikely to achieve repeatable, successful development efforts. We strongly believe that the *only* place a true development-centric process can come from is the developers themselves. Your development team should certainly allow management to interact with and improve your process so it meets their needs as well, but, in our experience, management cannot sufficiently dictate a good development model. Creating a good development process is never an easy task, but it is always worth undertaking. Even small improvements will pay large dividends over time.
- Each CORBA project team should spend the necessary time up front to lay the foundation for good analysis and design work. A strong object model is not built by accident. It will take several revisions in order to discover an appropriate structure that will not only work in your current implementation, but will continue to work for years to come. We encourage the use of some form of use case analysis for object discovery and robustness analysis to solidify your long-term object model.

Summary

This chapter has given a high-level overview of CORBA, including the sum of its parts and how they work together. If you would like a more extensive look at the details of CORBA, browse the CORBA specification, which can be found at the OMG's Web page (www.omg.org) in the documentation section.

CHAPTER 2

The OMG Interface Definition Language

This chapter describes the Interface Definition Language, or IDL, in depth. It should be noted that the IDL is part of the CORBA standard; therefore, once you learn IDL, you can use it with any CORBA implementation.

IDL is the means by which a particular object implementation tells all potential clients what operations are available and how they should be invoked. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

The OMG IDL grammar is similar, in many ways, to the C++ grammar, with additional constructs, including support for the CORBA operation invocation mechanism. OMG IDL is a declarative language. It supports C++ like syntax for constants, type, and operation declarations.

IDL has constructs to describe an interface. Interfaces, obviously, are the most important constructs in IDL because they form the basis for describing the players in a distributed system. In addition, IDL allows you to define data structures that are used to represent the data that flows through the distributed system.

The Preprocessor

The IDL preprocessor is similar to the C++ preprocessor, so many of its functions and its syntax are similar. IDL provides preprocessing directions that allow for macro substitution, conditional compilation, and source file inclusion.

As with a C++ include file, the following directive should be used in an IDL file to prevent multiple inclusion errors.

```
//IDL
#ifndef <unique_name>
#define <unique_name>
<Body of IDL file.>
#endif
```

The other preprocessing directives available in IDL are as follows: #define, #undef, #include, #if, #ifdef, #indef, #elif, #else, #endif, #defined, #pragma.

Modules

An interface can be defined within a module; this allows interfaces and other IDL-type definitions to be grouped together in a useful fashion. Modules also create a naming scope; thus a type name used within one module will not conflict with the same name used in another module.

The following pseudo-IDL illustrates the use of a module:

```
//IDL
module holdings{
  interface account { . . . };
  interface security { . . . };
  interface security_factory { . . . };
  interface quote_service { . . . };
}
```

Note that the full or scoped name is specified as Module::Interface, or holdings::account, for example.

Interfaces

The IDL interface provides a description of the functionality that will be provided by an object. An interface definition provides all the information needed to develop a client that can use this defined interface to interact with the object. An interface definition typically specifies the attributes and operations belonging to that interface, as well as the parameters of each operation. Defining the interfaces between components is the most important aspect of distributed object design. Interfaces are the single most important feature of IDL.

The interface body can contain the following kinds of declarations:

- Constant declarations, which specify the constants that the interface exports
- Type declarations, which specify the type definitions that the interface exports
- Exception declarations, which specify the exception structures that the interface exports
- Attribute declarations, which specify the associated attributes exported by the interface

- Operation declarations, which specify the operations that the interface exports and the format of each, including the operation name, necessary parameters, and exceptions that might be returned

Here is a little snippet of IDL that will give you an idea of how it is used. Imagine that you are creating a stock market pricing server. Your IDL might look something like this:

```
interface security {
        //attributes
        attribute string ticker;
        attribute string issuing_company;
        //operations
        float get_current_price();
}
interface security_factory {
        //operations
        security get_security(in string ticker);
        void remove_security(in security aSecurity);
}
```

We have two interfaces here, a security and a security_factory. The value and use of the security object should be fairly obvious. Note that, as already defined, an object has attributes, and methods or operations that can be invoked upon an instance of that object. Attributes are defined using the following syntax:

```
'attribute' <attribute_type> <attribute_name>;
```

Operations or methods have this syntax:

```
<method_return_type> <method_name> (<parameter_direction>
        <parameter_type> <parameter_name>, (1-n))
```

Parameters that are used in methods can have one of the three direction adjectives:

1. in. The parameter is passed from the client to the called object.
2. out. The parameter is passed from the called object to the client.
3. inout. The parameter is passed in both directions.

The security interface has two attributes: its ticker and the name of its issuing company. It has one operation, which allows us to determine its current price.

The second interface is security_factory. This introduces the concept of an object factory. We use an object factory to control an object lifecycle within our system. Our security_factory has two operations to do exactly that: (1) get_security, whereby the factory would create a new security object and, presumably, do some sort of database lookup to populate the issuing_company attribute, then return the security object reference to the calling client, and (2) remove_security, which would destroy the object and remove it from use within the system. The factory pattern is one that will be used often in your CORBA systems.

Oneway Operations

Normally, operations will block the calling client until the called method returns. However, an IDL operation can be defined as oneway. A oneway operation will not block the calling client, so it can proceed with its functions once the request is dispatched to the server. Unfortunately, there are some limitations with the oneway operation: (1) It must be declared with a void return type, (2) it cannot have any out or inout parameters, and (3) it cannot have a raises clause.

Here is an example of how a oneway operation can be used:

```
interface user_display {
        oneway void alert (in string theAlertText);
}
```

Inheritance

An interface can be derived from another interface. A derived interface may be extended by adding new elements that are not supported in the base interface. In addition, unless redefined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator (**::**) may be used to refer to a base element explicitly. A derived interface may redefine any of the type, constant, and exception names that have been inherited.

An interface is called a direct base if it is mentioned in the inheritance declaration. An interface is called an indirect base if it is not a direct base but is still a base (direct or indirect) interface of one of the direct base interfaces mentioned in the inheritance specification.

An interface may be derived from any number of base interfaces. Such use of more than one direct base is called multiple inheritance. The order of derivation is not significant, functionally.

An interface may not be specified as a direct base interface of a derived interface more than once; it may be an indirect base more than once.

Figure 2.1 displays a legal path of inheritance.

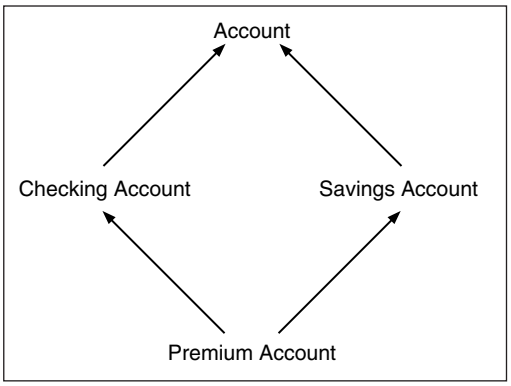

Figure 2.1 Multiple inheritance.

Reference to base interface elements must be unambiguous. Reference to a base interface element is ambiguous if the expression used refers to a constant, type, or exception in more than one base interface. It should be noted that it is currently illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute in the derived interface. Ambiguities can be resolved by qualifying a name with its interface name using the scoped name.

References to constants, types, and exceptions are bound to an interface when it is defined. This means that the names used are replaced with the fully scoped names upon definition.

Operation names are used at run time by both the stub and the dynamic interfaces. As a result, all operations that might apply to a particular object must have unique names. In other words, CORBA IDL does not support overloading of operation names.

Exceptions

The standard way of processing errors in CORBA is through exceptions. An IDL operation may raise an exception indicating that an error has occurred. Exceptions provide a clean way for an operation to raise an error to the caller. This is illustrated in the following:

```
interface security {
        exception invalid_ticker{ string reason; };
        //attributes
        attribute string ticker;
        attribute string issuing_company;
        //operations
        float get_current_price()
        raises(invalid_ticker);
}
```

The preceding exception, invalid_ticker, will be raised when the operation get_current_price is called with an invalid ticker.

In addition to user-defined exceptions, like the invalid_ticker exception above, a set of standard exceptions is defined in CORBA. These correspond to the standard run-time errors that may occur during the execution of a request.

Basic IDL Types

Table 2.1 lists the basic types supported in IDL.

Constructed Types

IDL supports three constructed types: structures, enumerated types, and discriminated unions.

Table 2.1 Basic Types

TYPE	IDL IDENTIFIER	DESCRIPTION
float point type	float	IEEE single-precision floating point numbers
	double	IEEE double-precision numbers
	long double	IEEE long double
integer type	long	32 bit
	short	16 bit
	unsigned long	32 bit
	unsigned short	16 bit
	long long	64 bit
	unsigned long long	64 bit
char type	char	An 8-bit quantity
	wchar	A wide character
boolean type	boolean	TRUE or FALSE
octet type	octet	An 8-bit quantity that is guaranteed not to undergo any conversion during transmission
any type	any	The any type allows the specification of values that can express an arbitrary IDL type
string type	string	ISO-Latin 1 string
	wstring	A wide character string

Structures

A struct data type allows related items to be grouped together in a useful fashion. For example,

```
//IDL
struct account_details {
        string name_of_owner;
        float total_value;
        string date_opened;
};
interface account {
        attribute account_details the_account_details;
};
```

Enumerated Types

Enumerated types consist of ordered lists of identifiers. Here is an example of one:

```
enum months (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
```

Discriminated Unions

The IDL union type is a space-saving type whereby the amount of storage required for a union is the amount necessary to store its largest element. The tag field is used to specify which member of a union instance is currently assigned a value.

```
//IDL
union token switch (long) {
        case 1 : long 1;
        case 2 : float f;
        default : string s;
};
```

OMG IDL unions are a cross between the C union and switch statements. IDL unions must be discriminated; that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call. Each expression that follows the case keyword must be compatible with the tag type. The type specified in parentheses after the switch keyword must be an integer, char, boolean, or enum type. A default case can appear at most once in a union declaration, and cannot appear if all legal values of the discriminator type appear in case labels.

Arrays

IDL provides multidimensional fixed-size arrays to hold lists of elements of the same type. The size of each dimension should be specified in the definition. Some examples are as follows:

```
//IDL
//A one dimensional array
Employee theEmployees[100];
//A two dimensional array
short grid[10] [10];
```

Template Types

IDL provides for only two template types: sequence and string.

Sequences

A sequence is a one-dimensional array with two characteristics: a maximum size (which is fixed at compile time) and a maximum length (which is determined at run time). A sequence is similar to a one-dimensional array, but a sequence is not a fixed length. Sequences are often preferred to arrays because of this feature. You can create a sequence of any type.

```
//IDL
typedef sequence<long, 10> theBoundedSequence;
```

This first example is of a bounded sequence.

```
//IDL
typedef<long> theUnboundedSequence;
```

This example is unbounded and can be of unlimited length.

A sequence that is used in an interface definition (i.e., in its operation or attribute definitions) must be named by a typedef declaration.

Strings

The string type is implemented in a fashion similar to a sequence of char. A string may be bounded or unbounded depending on whether your model calls for a fixed-length string or not.

```
//IDL
interface library {
        //misc details
        //A bounded string
        attribute string<12> dewey_decimal_code;
        //An unbounded string
        attribute string title;
};
```

Constants

Constants can be defined at any level: interface, module, global, or file level. A constant is defined as follows:

```
//IDL
interface account {
        const long maximum_holdings = 1000;
        //rest of the definition
};
```

Constants can be of type long, unsigned long, unsigned short, char, boolean, float, double, and string.

Typedef Declaration

A typedef declaration can be used to define a meaningful name for a basic or a user-defined type. For example,

```
//IDL
typedef long lsize;
```

will define lsize as a synonym for long.

Forward Declarations

An interface must be declared before it can be referenced. A forward declaration declares the name of an interface without defining it. This allows the definition of interfaces that mutually reference each other. The syntax is simply the keyword interface followed by the interface identifier. The actual definition must follow later in the specification.

```
//IDL
interface account;
```

Pseudotypes

CORBA defines a number of IDL pseudo interfaces, that is, interfaces defined in IDL but whose implementation does not necessarily follow the normal mapping from interfaces to a target programming language. In particular, the following pseudotypes are defined:

- NamedValue
- NVList
- Request
- ServerRequest
- Context
- TypeCode
- ORB
- Environment

Valuetypes

CORBA objects are represented in IDL by interfaces and are passed by reference. This means that when an object is passed as a parameter to an IDL operation, an object reference is created and sent over the wire to the client. No state information associated with the object is transferred. As a result, any invocation using the object reference is a remote operation and is handled by the remote implementation.

It is sometimes more desirable to pass objects by value, rather than by reference. If the primary purpose of the object is to encapsulate data or if the application wants to copy an object, it is preferable to pass the state information associated with an object, construct an instance of the object implementation locally, and populate the state infor-

mation, rather than passing a reference to the object. Such passing of state information is termed pass by value. This does match the pass by value semantics used commonly in programming languages. The IDL equivalent of representing an object passed by value is called a valuetype. An IDL valuetype looks like:

```
//IDL
valuetype StockQuote {
        // state members
        private string ticker;
        private float stockprice;

        // an attribute
        attribute float price;
        //A method
        float calculate();

        // a factory method
        factory create (in string sym, in float cur_price);
};
```

The valuetype StockQuote defined above contains two private state members, one attribute, one operation, and one factory method. In some sense, an IDL valuetype is midway between an IDL struct and an IDL interface. An IDL valuetype can be imagined to be an IDL struct with operations and inheritance. It can contain state members who can be either private or public operations and attributes. Note that attributes are only shorthand for two methods—a mutator and an accessor, and do not really represent state. Understanding the difference between attributes and state is key to understanding the difference between an IDL interface and an IDL valuetype. Factory declarations provide portable interfaces to create IDL valuetypes. The important characteristics of IDL valuetypes are listed below:

- IDL valuetypes are local. They are implemented locally and reside locally. Their state information is marshaled across from one entity to another (client or server). They cannot be invoked upon remotely like a CORBA Object.
- Valuetypes impose constraints on the implementation in that all entities (clients and servers) dealing with a given valuetype need to have a local implementation of that valuetype. With IDL interfaces, there is no constraint at all on the implementation of an interface. The receiving side of a parameter that is passed by value needs to know the implementation of the valuetype so that it can instantiate an instance of the valuetype and fill in with the state information passed to it. In other words, the implementation of a valuetype at the receiving side should be structurally similar to the implementation of the valuetype on the sending side.
- Valuetypes can singly inherit from another concrete valuetype and multiply inherit from zero or more abstract valuetypes. Valuetypes can also support one

or more IDL interfaces, with at most one being nonabstract, using the supports IDL keyword.

- **Sharing semantics.** Valuetypes allow sharing. Valuetype instances can be shared between other valuetype instances. If the same valuetype is passed multiple times as arguments to a method invocation, the single valuetype is received on the receiving end instead of multiple copies. Similarly, if a valuetype is referred at multiple points in a graph, the same relationship is maintained in a receiving context. This is not possible with IDL structs, unions, or sequences. This allows arbitrary graphs to be passed and to have that graph structure maintained in the receiving context.
- **Null semantics.** Valuetypes allow passing NULL as a valid value for valuetype parameters to IDL operations as opposed to IDL strings, structs, or sequences. Since this is possible, valuetypes can act as wrappers to wrap any non-value IDL type to give these types value semantics.
- Valuetypes can be of two types: concrete or stateful valuetypes and abstract valuetypes.

Concrete Valuetypes

Concrete or stateful valuetypes contain state members that could be either public or private. Concrete valuetypes can be used to describe complex state, such as an arbitrary graph. Using stateful values, it is possible to describe recursive data structures such as a cyclic graph, or a linked list, which was previously not possible. The following IDL shows a concrete valuetype:

```
//IDL
valuetype Vertex;
typedef sequence<Vertex> Vertices;
valuetype Vertex {
        private string label;
        private Vertices branches;
        Vertices traverse();
        void add (in Vertex vert);
        void remove (in Vertex ver);
};
valuetype Graph {
        // private state member
        private Vertex root;
        //A method
        Vertex traverse();
};
```

Abstract Valuetypes

Abstract valuetypes contain operations only and no state data. They cannot be instantiated. Only concrete or stateful valuetypes derived from abstract valuetypes can be instan-

tiated. Abstract valuetypes can multiply inherit from abstract valuetypes. Concrete valuetypes may also inherit from multiple abstract valuetypes. This is possible because abstract valuetypes have no state. Abstract valuetypes can also support multiple interfaces with at most one of them being non abstract. They are represented in IDL using the prefix abstract. The following IDL modifies the concrete valuetype example to show an abstract valuetype in IDL:

```
//IDL
abstract valuetype traversable {
        Vertices traverse();
};
valuetype Vertex : traversable {
        private string label;
        private Vertices branches;
        void add (in Vertex vert);
        void remove (in Vertex ver);
};
```

Boxed Valuetypes

It is sometimes convenient to define a valuetype with a single state member with no inheritance or methods. Such valuetypes can be defined in IDL using a short hand IDL notation called a boxed valuetype. A boxed valuetype can be used to support sharing semantics and null semantics described above. The following IDL shows a boxed valuetype:

```
//IDL
valuetype Label string;
```

It is a shorthand representation of the following valuetype:

```
//IDL
valuetype Label {
        // state member
        private string value
};
```

Custom-Marshaling

If users want to provide their own implementations of marshaling routines for valuetypes, they should use the custom prefix when declaring the valuetype in IDL. The following IDL represents a valuetype that is custom-marshaled. The code to read and write the state of this valuetype needs to be implemented by the user:

```
//IDL
custom valuetype Date {
        // private state member
```

```
        private unsigned long day;
        private string month;
        private unsigned long year;
        // method
        String findTomorrow();
};
```

Abstract Interfaces

If, at compile time, we cannot determine whether an object is to be passed by reference or by value, it is possible to declare such a parameter's type as an abstract interface in IDL. IDL abstract interfaces allow the decision to be deferred until runtime. An IDL abstract interface can be thought of as a super class to an IDL interface as well as an IDL valuetype. Therefore, by themselves, abstract interfaces do not support any standard CORBA::Object operations. If an IDL abstract interface can be successfully narrowed to be an IDL interface, then CORBA::Object operations can be invoked on the narrowed object reference. The following IDL defines an abstract interface Person:

```
Typedef sequence<string> Details;

abstract interface Person {
    Details getDetails();
};
interface Employee : Person {
        // attributes
        attribute unsigned long identity;
        attribute string identity;
};
valuetype PersonValue supports Person {
        // state members
        private string name;
        private unsigned long birthYear;
        // method
        unsigned long calculateAge();
};

interface Company {
        // operation
        Details printDetails( in Person person );
};
```

When an IDL abstract interface is used as a parameter to an IDL operation, if at runtime:

- the actual parameter passed can be determined to be of a regular interface type, or a subtype of the regular interface type
- that regular interface type is a subtype of the signature abstract interface