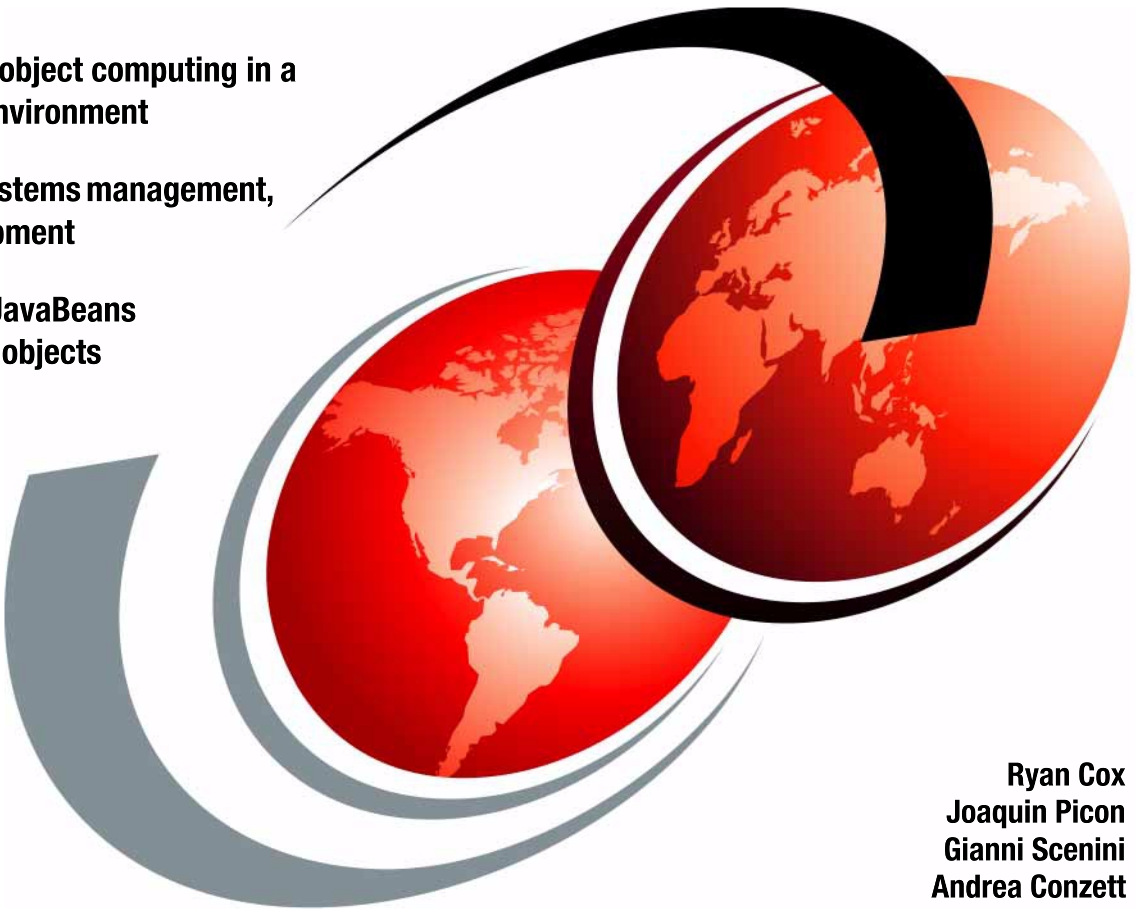
IBM

IBM WebSphere Application Server Enterprise Edition

Component Broker 3.0: First Steps



Distributed object computing in a multi-tier environment

Runtime, systems management, and development

Enterprise JavaBeans and CORBA objects support

Ryan Cox
Joaquin Picon
Gianni Scenini
Andrea Conzett

Redbooks

ibm.com/redbooks

SG24-2033-03

International Technical Support Organization

IBM WebSphere Application Server Enterprise Edition Component Broker 3.0 First Steps

August 2000

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix E, "Special notices" on page 419.

Fourth Edition (August 2000)

This edition applies to IBM WebSphere Application Server Enterprise Edition 3.0, for use with Windows NT.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

Contents

Figures

Preface

This IBM Redbook introduces you to Component Broker object-oriented (OO) application development in the multi-tier environment, and is part of the International Technical Support Organization (ITSO) Component Broker Redbook series.

Component Broker is a new member of the IBM WebSphere Application Server Enterprise Edition product family that supports distributed object computing in a multi-tier environment. Component Broker provides a middle-tier application environment that allows Business Objects to be highly managed and integrated with back-end databases and transactional systems. Different types of first-tier clients can access the middle-tier Business Objects. The middle-tier essentially includes middleware that provides clients with an object-oriented rendering of other middleware. In this regard, Component Broker is not a stand-alone product, but instead, is designed to work with existing resource managers that provide persistence, concurrency control, and other services needed in commercial computing environments.

Component Broker is composed of an industry-leading set of technologies that facilitate distributed object applications. As the only solution of its kind on the market today, it combines three critical dimensions: runtime, systems management, and development.

Component Broker consists of two parts that support these three dimensions. The Component Broker (CBConnector) provides the runtime environment and supports systems management. The Component Broker Toolkit (CBToolkit) contains the tools that application developers use to define and implement the objects running on the middle-tier.

Component Broker's unique value lies in its ability to integrate and therefore leverage existing enterprise transactions and data. It will most often be used to extend existing business models.

This IBM Redbook gives you an opportunity to become familiar with the Component Broker Toolkit and with the development process itself. We describe, and give you the opportunity to study, the development process by using incremental samples. You can execute and develop your own applications with the guidance this document provides.

Together with the *IBM Component Broker – Programming Guide*, G04L-2376, and the *IBM Component Broker – Quick Beginnings Guide*, G04L-2375, this redbook helps you make rapid progress in understanding and gaining practical experience with Component Broker. The descriptions and examples in this redbook apply to Component Broker 3.0. The redbook in general applies to WebSphere Application Server Enterprise Edition 3.0, for use with Windows NT.

We recommend that you first explore the Component Broker architecture by reading the Redbook, *IBM Component Broker Connector Overview*, SG24-2022.

The ITSO has added to the *IBM Component Broker Cookbook Collection* a sample banking application written in the three-tier model which re-uses legacy system applications and data by accessing CICS on MVS. The application is delivered and documented in two redbook publications, the *IBM CBConnector Cookbook Collection: CBConnector Bank User Guide*, SG24-5121, which is accompanied by a CD-ROM, and the *IBM CBConnector Cookbook Collection: CBConnector Bank Implementation,* SG24-5119.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

The project to create the original version of the book was designed and managed by:

Alex Gregor

The project to update the book for version 3.0 of IBM WebSphere Application Server Enterprise Edition was managed by:

Joaquin Picon and **Andrea Conzett**

The team that wrote the original book:

Tomas Carlsson is a Senior Technical Analyst at Industri-Matematik International (IMI). IMI develops and supports System ESS, a high-volume, mission-critical product for Demand Chain Management. Tomas has worked at IMI for six years in areas such as application development tools, operating systems, database technology, object-oriented technologies, Java, C++, and CORBA.

Alex Gregor is an IBM Senior Software Engineer working at the ITSO Austin in the OO/AD group. His responsibilities include technical support for IBM application frameworks.

Michael Hofstaetter is a systems engineer at Spardat in Austria. Spardat is the central information systems center for a major section of the country's banking sector. It develops and supports all kinds of banking software. Prior to specializing in distributed computing, Michael invented and implemented Paladin - a part of the company's middleware for automatic document creation and printing. Before joining Spardat, he developed compilers and an object-oriented GUI framework during his time at Vienna's Technical University.

Nils Joensson is a Senior Systems Engineer at Industri-Matematik International (IMI) in Stockholm, Sweden. IMI develops and supports System ESS, a high-volume, mission-critical product for Demand Chain Management. Nils has worked at IMI for ten years, nine years with implementation and maintenance of System ESS at customer sites and the last year with object-oriented applications.

Henri Jubin currently works for the International Technical Support Organization (ITSO) in Austin, where he covers the area of Object Oriented Technology and in particular the JavaBeans and Java Enterprise arena. Henri has previously worked in various support and consulting positions within IBM France. He has dealt with topics such as Object-Oriented Technology, OS/2 Windows NT and OpenDoc.

Joaquin Picon is a consultant at the International Technical Support Organization, San Jose Center. He writes extensively and teaches IBM classes worldwide on application development, object technology, CORBA and Enterprise JavaBeans. Before joining the ITSO, Joaquin worked at the IBM Application Enabling Center Of Competency in France. Joaquin holds a degree in telecommunications from the Institut National de Telecommunications (`http://www.int-evry.fr`).

Ivo Plath is an IBM IT architect working for IBM Global Services in Germany. He has five years of experience in object-oriented application development. Ivo had the technical lead of several object-oriented projects. He is now supporting large customers in the development, integration and operating of distributed, object-oriented applications.

Pasi Salminen is a project manager at Profit Ltd in Finland. He has six years of experience in object technology and three years in distributed computing. Profit Ltd is building large-scale insurance systems using object technology. Its main product is Once&Done, a system managing the complete insurance process from point-of-sale to contract administration. Profit's goal is to provide insurance companies a path from the current "system jungle" to a component-based system architecture conforming industry standards and running on all major platforms.

Takeo Shibazaki is an Information Technology Specialist in IBM Japan/Systems Laboratory. He has worked for five years at IBM and his responsibility is technical support to field system engineers. He worked with AIF (Application Integrated Feature), Lotus Notes and Object-Oriented technology.

Heinz Stoellinger is a consultant at IBM Austria. He has written on advanced object technology in leading trade magazines and has over seven years of experience in object-oriented application development. Prior to that, he worked mainly as an IBM systems engineer supporting large customers in areas such as operating systems, networking, application development, and database technology.

The team that updated and expanded the original book:

Hiroshi Arai is a Consulting Information Technology Specialist at Technical Support Center in IBM Japan. Currently he is doing a consulting support for NLS related problems for Japanese customers including Object technology, application conversion and year 2000 paradigm. He has over fifteen years of experience in application system development including client server application and MVS/VM host application.

Andrea Conzett is an Advisory Information Technology Specialist at IBM's International Technical Support Organization, San Jose Center. He has 14 years of experience in application development. He writes extensively, and he teaches IBM classes worldwide on high-level language application development.

Ryan Cox is an Object Technology/Application Development specialist working in IBM Advanced Technical Support. He has worked on projects in many areas including Internet/Intranet development, Lotus Notes/Domino, Java, distributed computing, and CORBA. He currently supports application development tools in the IBM VisualAge family including VisualAge for Java, and Component Broker in the IBM Transaction Series.

Alex Gregor is an IBM Senior Software Engineer working at the ITSO Austin in OO/AD group. His responsibilities include technical support for IBM application frameworks.

Hanne Rygg Johnsen is a Systems Engineer in the Nordic Object Technology Practice (OTP), IBM Norway. She has four years of experience in developing object-oriented systems with Smalltalk, and most of them have involved interfacing to legacy systems. As a member of the OTP, she is currently dedicated to the reuse of application frameworks and to provide customer consulting services in object-oriented analysis and design.

Stefan Lindblad is currently working for A-Ware Ltd, an independent Finnish specialist consultancy and software company devoted to the exploitation of Internet and Java technologies. Stefan has fifteen years of experience in programming on various platforms and languages. A-Ware Ltd develops applications and products to satisfy client requirements for security-critical solutions. A-Ware Ltd manages and takes parts in projects for banks, insurance companies, the Finnish state and various other major national industries (incl. telecommunication).

Gianni Scenini works at IBM Italy and he has eight years of strong knowledge in computer science. His area of expertise includes object oriented technologies, distributed objects, framework and component design, n-tier architectures, middlewares, network and application security, and emerging technologies. He has written several articles on EJB, CORBA, DCOM , Java/C++ languages, security , application servers, IT architectures for enterprise applications, object oriented design, framework and component design, and network protocols.

Terje Storstein has worked as a Systems Engineer in IBM Norway for almost thirty years. He started in the mainframe area, then via distributed systems of the 70's to the client server scenarios of the 80's and 90's. Currently he is working in the e-business department of the Norwegian Global Services.

Thanks to the following people for their invaluable contributions to this project:

IBM International Technical Support Organization Center in Austin, Texas:

Marcus Brewer
Steve Gardner

IBM International Technical Support Organization Center in San Jose, California:

Evgeny Deborin
Andrea Conzett
Hanspeter Nagel
Joaquin Picon

AD Center of Competency in La Gaude, France:

Bruno Georges
Bertrand Gruson
Richard Mills
Muriel Viale

IBM Advanced Technical Support Center in Dallas, Texas:

David Johnson

IBM Development Laboratory in Austin, Texas:

Virgil A. Albaugh
Eleni Barton
Roger Cundiff
Steve Gardner
Robert H. High Jr
Henri Jubin
Ron D. Martin
Cynthia A. McFall
Kim Rochat
Kathy B. Sitar

IBM Development Laboratory in Rochester, Minnesota:

Eric N. Herness
Eric H. Jenney
Randy Schnier
Peter Schommer
Michael Shupert

IBM Santa Teresa Development Laboratory in San Jose, California:

Robert Garnero
Paula Rutherford

David Wisneski
Peiyuan Yan

IBM Development Laboratory in Toronto, Canada:

Alan S. Boxall
Frank J. Budinsky
Tim Francis
Karla J. Johnson
Suman Kalia
Eric Labadie
Christina P. Lau
Harm Sluiman
Vito Spatafora
Joe Wigglesworth

IBM Japan Systems Engineering Company:

Kanako Michimoto

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Please send us your comments about this or other Redbooks in one of the following ways:

- Fax the evaluation form found in "IBM Redbooks review" on page 441 to the fax number shown on the form.
- Use the online evaluation form found at **ibm.com**/redbooks
- Send your comments in an Internet note to redbook@us.ibm.com

Part 1. Ready

Part 1 concentrates on setting the stage for the development of the incremental samples. It explains how this document is organized and how to use it. As you may have noticed, this redbook is accompanied with a CD-ROM. The main objective of our "cookbook" format is to introduce you to the CBConnector development environment through interaction between reading and using the sample code from the CD-ROM.

In addition, we paint the "Big Picture" to illustrate why IBM CBConnector is an excellent tool for developing distributed applications. A description of the needed prerequisites is provided, and you can test your current development environment using the sample Ping application.

Good luck with your First Steps!

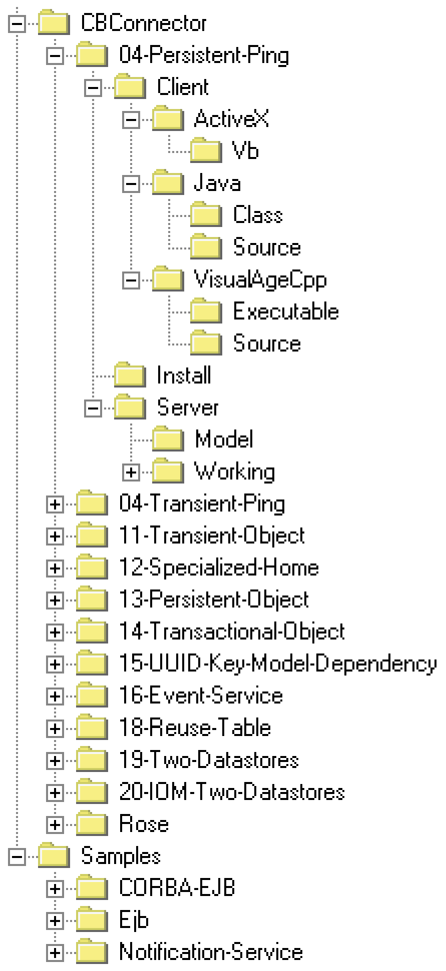
Chapter 1. How to read and use our book

This introductory chapter concentrates on how this document is organized and provides you guidance on how to use it.

As you will notice when reading this book, Chapters 9 through 23 are identically structured. First, we give you a hint of the major issues examined in the chapter. We show you the CBConnector componentry involved and let you build the server and client applications. Then we provide you a short description of how to deploy and execute the created application. You can always refer back to Chapter 4, “Test your development system” on page 29 and to *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - Getting Started with Component Broker*, SC09-4433, in matters of System Management and deployment of applications. At the very end of the chapters, we summarize what you learned.

1.1 How this book is organized

The directory layout on the CD-ROM mirrors the chapters in the book. You will find a directory subtree underneath the CBConnector directory, as well as the Samples directory, for each sample. An overview of the directory structure is shown in Figure 1.



```
CBConnector
    04-Persistent-Ping
        Client
            ActiveX
                Vb
            Java
                Class
                Source
            VisualAgeCpp
                Executable
                Source
        Install
        Server
            Model
            Working
    04-Transient-Ping
    11-Transient-Object
    12-Specialized-Home
    13-Persistent-Object
    14-Transactional-Object
    15-UUID-Key-Model-Dependency
    16-Event-Service
    18-Reuse-Table
    19-Two-Datastores
    20-IOM-Two-Datastores
    Rose
Samples
    CORBA-EJB
    Ejb
    Notification-Service
```

Figure 1. CD-ROM directory structure (04-PersistentPing example expanded)

Each sample has two main directories: Client and Server. The Client directory is divided into the three subdirectories holding the ActiveX, VisualAge C++, and Java clients. In the Install directory, you can find the installation executable. The server subdirectory holds the Object model and the working subdirectory with Data and Disk1 subdirectories.

In some samples, you can also find a subdirectory called *\client\simplecppclient*. In this subdirectory, we keep very simple C++ code and executables showing the proof of concepts exercised in this chapter.

As you can see in Figure 1 above, you can find a directory unrelated to the samples in the chapters.

- *Rose* - This subdirectory keeps the Rose Object Builder Bridge-related files you need.

All the" root" directories for the samples are listed below. They are either in the CBConnector directory or the Samples directory.

Sample	**Directory you can find it in**
Chapter 4, Test your development system	04-Transient-Ping 04-Persistent-Ping
Chapter 11, Transient object sample	11-Transient-Object
Chapter 12, Specialized home sample	12-Specialized-Home
Chapter 13, Persistent object sample	13-Persistent-Object
Chapter 14, Transactional object sample	14-Transactional-Object
Chapter 15, Object with UUID key and model dependency	15-UUID-Key-Model-Dependency
Chapter 16, Event service	16-Event-Service
Chapter 17, Notification service	Notification-Service
Chapter 18, Deploying Enterprise JavaBeans	Ejb
Chapter 19, Mixing CORBA Objects and Enterprise Java Beans	CORBA-EJB
Chapter 21, Query service - reuse of an existing table	18-Reuse-Table
Chapter 22, Meet-in-the-middle paradigm with two datastores	19-Two-Datastores
Chapter 23, IOM with two datastores	20-IOM-Two-Datastores

1.2 The best way to use this book

Now that you know how the book and CD-ROM are organized, we suggest the best way for you to interact with them.

Starting with Chapter 8, “The Account scenario” on page 101, our samples are built pedagogically, meaning from the simple to more complex. Our suggestion is to study the book from beginning to the end. Here are the recommended steps for using the CD-ROM samples:

1. Create a CBConnector directory on your hard drive.
2. Create a Samples directory on the same hard drive.
3. Map that hard drive to **O**.
4. Copy the subdirectory from the CD-ROM that maps the chapter you are reading to **O** under the corresponding subdirectory (CBConnector or Samples).
5. Execute the `ATTRIB -R` command on your new directory in order to remove read-only flag from the copied files.
6. Start the CBConnector Object Builder.
7. Load the Object Model from the level of Server directory.

When you reach this point, you are ready to start working with the samples. You can follow the three-step model while reading the book chapter by chapter:

- **Step 1**
  - Install and run the sample by following the instructions in the sections titled "Deploy and Run It".
- **Step 2**
  - Load and study the model.
  - Generate all the necessary files described in the chapters as you are building the application.
  - Build the application-family batch file.
  - Deploy and run the sample.
- **Step 3**
  - Use your own naming convention and build the described applications with new interfaces (keep the implementations) from scratch.
  - Verify the correctness with the working sample.
  - Deploy and run your samples.

We hope the methodology presented in this book, together with the other Component Broker publications, the *IBM Component Broker Connector Overview*, G*etting Started with IBM Component Broker*, SC09-4433, the *IBM Component Broker – Programming Guide*, SC09-4442, and the online documentation, will make it easier for you to start developing applications in the Component Broker environment.

Again, we wish you good luck!

Chapter 2. Component Broker introduction

In this chapter, you will find a quick introduction into what IBM's new Component Broker is all about. This is supplied just in case you have not seen the companion redbook, *IBM Component Broker Connector Overview*, SG24-2022, in particular, Part 1 of that document. For those of you who have done so, you may find some of the following text familiar. This is no surprise. For the most part, it is just a more concise version of the material in the redbook, *IBM Component Broker Connector Overview*, SG24-2022.

However, since this time we expect our audience to be developers, we will keep the introduction short and sweet. On the other hand, reuse of somebody else's work is a major paradigm in object technology. So, why not join the bandwagon?

We start you off by going into what Component Broker is; then we address the areas where we expect this new technology mainly to be applied. Finally, we position the product with respect to related IBM packages.

2.1 What is CBConnector?

IBM's Component Broker is a new, integrated package that allows you to develop, deploy, run, and manage a new generation of distributed, multi-tier, object-oriented applications.

Its major components are shown in Figure 2.

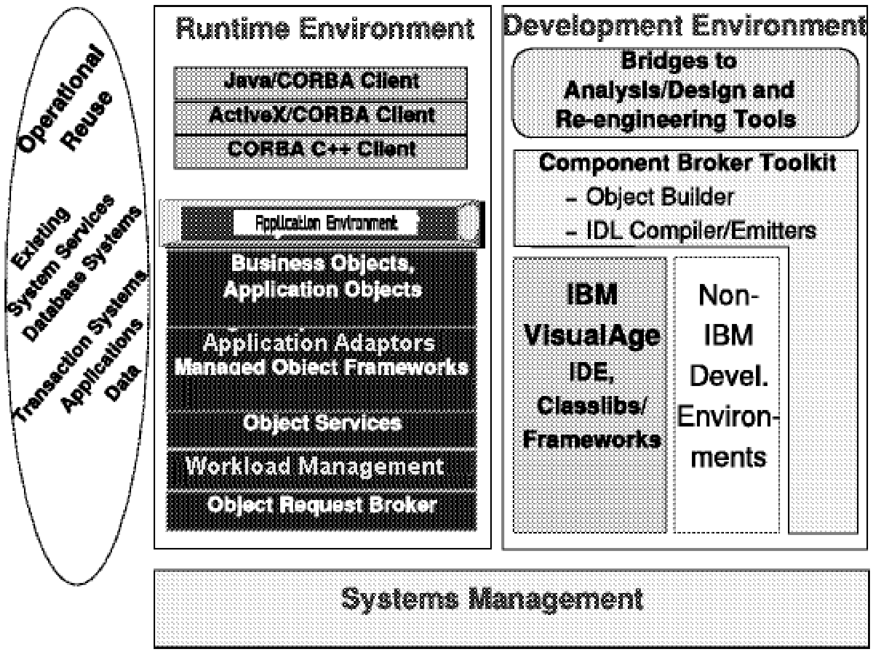

Operational Reuse
Existing System Services Database Systems Transaction Systems Applications Data
Runtime Environment
Java/CORBA Client
ActiveX/CORBA Client
CORBA C++ Client
Application Environment
Business Objects, Application Objects
Application Adaptors Managed Object Frameworks
Object Services
Workload Management
Object Request Broker
Development Environment
Bridges to Analysis/Design and Re-engineering Tools
Component Broker Toolkit
- Object Builder
- IDL Compiler/Emitters
IBM VisualAge IDE, Classlibs/ Frameworks
Non-IBM Devel. Environments
Systems Management

Figure 2. Component Broker building blocks

Leaving aside the somewhat amorphous, elliptical "blob" on the left for a moment, let's go over the figure from left to right and from top to bottom.

2.2 Component Broker dimensions

We briefly describe here the Component Broker's three dimensions shown in the figure above:

- Runtime
- Development
- System Management

2.2.1 Runtime dimension

The box labeled *Runtime Environment* shows that CBConnector connects to most of today's popular client environments. For the time being, we support clients written as Java applets and applications, clients written in VisualAge for C++, and clients developed for Microsoft's ActiveX platform. While you certainly can continue to run "fat-client" topologies, CBConnector is ideally suited for the thinner client variety. CBConnector does not restrict you in any way from using your favorite GUI frameworks or tools.

You code the user interface and any local business logic as usual. In addition, the developers of your CBConnector server code will provide you with the constructs needed to connect to your server. What you get depends on your client platform. In any case, the appropriate source code usage bindings are available for all the above environments. Java code can be downloaded or cached locally on the client, as usual. In the other environments, you will need to install the necessary DLLs to build and run your application. All clients connect to the application server through a CORBA-compliant Object Request Broker (ORB) and its Internet Inter-ORB (IIOP) protocol.

However, the core of CBConnector lies in its middle-tier application server. The server allows you to separate the traditionally extremely *volatile* client software environment from the code that implements your business logic. It also isolates that business logic from back-end technology specifics, such as databases or transaction monitors. This way, it provides you more opportunity to concentrate on the demands of your business, leaving the idiosyncrasies of the other tiers to the specialists in those areas.

The main components housed within the middle-tier application server are:

- **Application Adapters, Managed Objects, and frameworks**

  These provide the core infrastructure and scaffolding for your business logic to enable it to run in a mission-critical environment. They take care of such aspects as transactional behavior, connecting to back-end datastores and transaction monitors for persistence or legacy reuse, or interfacing to Object Management Group (OMG) object services.

  Application Adapters and the Managed Object framework *administer* what is called a *Managed Object Assembly*. Of that assembly, what is *visible* to a developer are the following objects:

  – **Business Objects**

    These contain your actual business domain code (that is, your business logic).

  – **Data Objects**

    These isolate the Business Objects from the specifics of back-end databases and transaction monitors. The Data Objects themselves use the services of other, framework-provided objects to implement the connection to back-end datastores. They can make use of extensive caching services to provide the kind of performance you would expect in today's online environments.

- **Managed Objects**

  These *wrap* the above two. The framework interfaces primarily with Managed Objects. When business logic operations are invoked on a Managed Object, the implementation is delegated to your Business Object. CBConnector-required methods are either implemented by the Managed Object itself or delegated to other parts of the runtime environment. Examples for the latter are setting up the transactional environment and implementing object services, such as the identity methods used by clients to compare object references.

There are a number of additional objects, such as:

- **Homes**

  In *object speak*, these are both factories and collections. They provide lifecycle functionality for objects of one particular type. They are also used to implement the query facility as well as iterability over the objects they contain.

- **Containers**

  These shield client programmers from hairy details, such as memory management for your Business Objects. They take care of activating and passivating objects according to policies you can specify through systems management facilities.

  In general, containers provide you with an administrative boundary for implementing policies on your Business Objects. These policies are set up at install and are then used at runtime to set up the *quality of service* required in the case at hand.

The frameworks are extensible, in that the implementation of framework interfaces can be adapted to your specific needs. One good example would be to extend them to run with your own application frameworks. Another one might be the need to support a back-end database environment that is not yet supported by CBConnector per se.

• **Object Services**

Just to give you an overview, here is what is included:

- **Naming Service**

  The CBConnector Naming Service is compliant with the OMG specification with some extensions to make working with the naming tree easier.

– **Security Service**

  At the time of writing, an implementation based on the Open Software Foundation/Distributed Computing Environment (OSF/DCE) is available. It supports authentication.

– **LifeCycle Service**

  CBConnector extends the OMG standard by giving you a measure of control over where you want to place your objects within the network of servers. This is accomplished through a concept called a *location*. Locations embody the notion of proximity, which may be interpreted in a geographical, structural, or even temporal sense.

– **Event Service**

  The CBConnector Event Service, at the Release 1 level, implements OMG's transient, non-typed events.

– **Externalization Service**

  Externalization Service has two main purposes:

  - Moving or copying the state of objects to different memory locations, between hosts, or simply to create a new object with the state of another already existing object
  - Moving the state of objects in and out of a persistent store

– **Query Service**

  You can access CBConnector server objects using keys. You can then follow links using references to get to related objects. CBConnector also allows you to execute queries using the query engine and query evaluators it provides. One way to do so is to specify your query using OO-SQL.

  CBConnector cooperates with back-end RDBMS systems, in particular with DB2, to optimize query performance. In particular, it uses a query push-down technology to evaluate as much as possible on the back-end. This way, indices present in the database can be used efficiently, and unnecessary data movement is avoided.

– **Transaction Service**

  The OMG Object Transaction Service is implemented mostly under the cover by Application Adapter Frameworks that support this requirement. CBConnector can act as a transaction coordinator, managing two-phase commit operations with back-end database systems. It works with the XA protocols, as defined in the X/Open Distributed Transaction Processing specification (XA). These can be connected easily to other XA-compliant resource managers.

  - **Concurrency Service**

    CBConnector's implementation of the OMG Concurrency Service does the job you would expect from it when you want to serialize access to resources. All of the well-known types of locks are supported.

- **Object Server runtime**

  The Object Server manages the collaboration among most of the other parts of the CBConnector platform. With respect to user code, it coordinates system resources such as execution threads or context information related to transactions or security, for example.

- **Workload management**

  This essential scalability feature brings load balancing into the picture. It allows clients to talk to a single, logical server image. In reality, operations that clients invoke may well be dispatched to different physical server processes. Introducing a new dimension in distributed object processing, Server Groups bring increased scalability and reliability to the picture.

- **Language interoperability**

  Essentially, the CBConnector object model is based on CORBA 2.0. Moreover, CBConnector includes a set of runtime libraries that supports inter-language calls between CORBA 2.0 objects within the same process.

  In the initial release, CBConnector-managed server-side Business Object implementations will be written in C++. On the other hand–considering developments such as Just-in-Time (JIT) or native platform compilation of Java source code–Java is certainly a high priority item on the list of server languages to support. The object model is a key component in supporting Business Objects written in Java.

  As you should expect from a distributed objects system worth its name, CBConnector client code does not have to be coded in the same language used for your server objects. At the time of writing, it supports Java, VisualAge for C++, or the languages supported within the Microsoft ActiveX platform.

2.2.2 Development dimension

Some of the prominent components of the CBConnector development environment are depicted on the right side of Figure 2 on page 8. Of the tools within that environment, the *Object Builder* is the major tool you use to define the structure and CBConnector-specific interfaces of your objects. Using a set of wizards, it guides you through defining and building all the constructs necessary in the CBConnector environment. Object Builder generates and helps you build the scaffolding needed right up to creating a deployable install image for servers and clients. This way, it saves you a lot of drudgery you would otherwise have to go through.

The Object Builder also has a bridging facility to outside analysis and design tools. This allows you to bring your analysis models into the CBConnector world and then extend them for CBConnector as appropriate. You can also import Interface Definition Language (IDL) files or RDBMS Data Definition Language (DDL) files to begin re-engineering your existing applications and databases.

As far as complementary development environments are concerned, the IBM VisualAge family is an obvious choice for Business Object development. For clients, you can continue to use your favorite platforms.

2.2.3 System management dimension

To start with, you use CBConnector's system-management facilities to define the network of host machines, and their configurations of servers and application software to parameterize the runtime environment.

At install time, you use its facilities to introduce your applications into the picture, deploying them within those configurations, as needed. At run time, you control the environment, bringing up servers, enabling or disabling applications, and so forth. Rather than being grafted on top as an afterthought, systems management has been designed into CBConnector from the start. For example, this has been done by instrumenting the code with the appropriate hooks for controlling the system and collecting management information from it. There is a short overview of the facilities built into CBConnector Release 1 in the redbook, *IBM Component Broker Connector Overview*, SG24-2022.

2.3 CBConnector's focus areas

We said we would talk later about the "amorphous, elliptical blob" on the left side of Figure 2 on page 8. Well, the time has come!

Essentially, CBConnector is a platform for developing a new generation of flexible software, built using distributed objects technology. It is an ideal base for implementing thin-client topologies, cleanly separating the client-side concerns of ever changing user interface technologies from the very different characteristics of the code needed to support business logic. The Internet is, of course, the most prominent example. CBConnector is very well suited to that dynamic environment.

However, one main focus area is that of the *operational reuse* of existing code and databases. Talking about CBConnector to one large customer in the insurance industry lately, one of the main things they were enthusiastic about was CBConnector's potential for reusing their legacy code. Specifically, they estimate their investment in domain-specific, home-grown software to be approximately ten times their present manpower capacity. In their view, it is ludicrous to even think about throwing all that away and starting any major system from scratch. IBM research has shown this situation to be more the norm than the exception with Fortune 500 companies.

Another typical requirement today is that of integrating separately developed application subsystems. This situation is aggravated by the surge in mergers and acquisitions we are experiencing. Through its middle-tier Business Objects application server, Component Broker offers an integrating platform that can bring together hardware and software systems that were developed without too much concern for interoperation.

2.4 Component Broker – a member of the Transaction Series family

Component Broker does not stand alone, but rather cooperates with existing transaction and resource managers. It functions as an object server by providing an application environment that lets clients work with mature back-end systems through object-oriented middleware. It cooperates fully with IBM's industry-leading family of transaction servers.

CBConnector incorporates proven technology from other IBM middleware products such as CICS/ESA, Transaction Series (Distributed CICS and Encina) as well as OSF/DCE. Over time, IBM intends to add Component Broker technology to Transaction Series so that these existing, proven platforms will also benefit from the new computing model.

Future offerings will expand support for other enterprise systems (for instance, IMS) and clients (such as Smalltalk). Further integration with systems-management environments, such as the Tivoli Management Environment (TME), is planned. Extensions to Component Broker and corresponding extensions to the *Shareable Frameworks Project* will enable Component Broker and Java/AS400 business frameworks to work together.

Component Broker development spans IBM teams and laboratories. This ensures that the whole product will take advantage of the diverse experiences and many disciplines required to build an integrated solution. We feel strongly that Component Broker is a third-generation object-technology offering. Component Broker's design itself is object-oriented and CORBA-compliant.

The ultimate goal is for it to work seamlessly with other CORBA-compliant products in the marketplace. It does so, recognizing that–given today's complex de-facto situation–the best way forward is one of evolution. And we invite you to bring your baggage of legacy along with you on the way.

Chapter 3. What do you need?

This chapter provides information you may find useful in planning your Component Broker installation. Please refer to the *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - Getting Started with Component Broker*, SC09-4433 as a complete and accurate source of information.

3.1 Planning for the installation

The following information is included:

1. Packaging
2. System requirements
3. Prerequisites
4. Component Broker packages
5. Installation considerations
6. Installation options and required prerequisites

3.2 Packaging

Component Broker is packaged as follows:

IBM Component Broker Toolkit Version 3.0 for Windows NT 4.0

This package includes:

- Component Broker compact disc
- CBToolkit compact disc
- Component Broker Supplemental Programs disc
- IBM VisualAge C++ for Windows, Version 3.5.3
- IBM VisualAge Java Enterprise 2.0 or higher
- Associated documentation

IBM Component Broker Connector Version 3.0 for Windows NT 4.0

This package includes:

- Component Broker compact disc
- Component Broker Supplemental Programs disc
- Associated documentation.

Notes

The IBM Database Server and DCE Security and Cell Directory Servers are separately licensed IBM and DCE packages. You can read more detailed information in 3.4, “Prerequisites” on page 18.

3.3 System requirements

This section lists the optimal system requirements for Component Broker with its prerequisite software.

IBM Component Broker Server (runtime or development)

- 200 MHz processor or better
- At least 128 MB of memory
- At least 5 GB of disk space
- 800x600 SVGA display
- At least 300 MB of paging space (500 MB optimal for Object Builder performance)

IBM Component Broker C++ or ActiveX client (runtime)

- 60 MHz processor or better
- At least 32 MB of memory
- Base operating system plus 150 MB of disk space

IBM Component Broker Java Client

- Processor, memory, and disk space as required by the native operating system

3.4 Prerequisites

The following products are prerequisites for Component Broker:

- Microsoft Windows NT 4.0 with Service Pack 4 applied
- IBM DCE 2.01 client (with FixPak DCENT11CE03 applied) with access to a DCE Security Server and a DCE Cell Directory Server
- IBM DB2 Client CAE (Client Access Enablement for Windows NT, Version 4.0.1 or 5.0) with access to an IBM Database Server for Windows NT, Version 4.0.1 or Version 5.0
- IBM Database Server for Windows NT, Version 4.0.1 or version 5.0

- IBM VisualAge C++ for Windows, Version 3.5.3 with FixPak WTC354 applied.
- Microsoft Visual C++, Version 5.0 (for ActiveX client development environments only).
- Microsoft Internet Explorer, Version 3.0 (for ActiveX clients only).
- IBM JDK 1.1.7B

Notes

1. The DCE client and FixPak are included on the Component Broker Supplemental Programs compact disc. The DCE Security and Cell Directory Server compact discs are part of a separately licensed DCE package.
2. DCE is required for security-enabled clients only.
3. The DB2 client is included on the Component Broker Supplemental Programs compact disc. The Database Server is part of a separately licensed DB2 package.
4. To develop Component Broker applications using DB2, the DB2 SDK is required. The DB2 SDK can be installed from the IBM Database Server compact disc to any system with Component Broker installed. The DB2 SDK can be installed on any system with Component Broker, whether it is a DB2 server or a DB2 client.
5. TCP/IP is a prerequisite for all Component Broker components. TCP/IP is installed as part of Microsoft Windows NT.
6. VisualAge C++ for Windows is a prerequisite for server application and CORBA C++ client development environments only and is included in the CBToolkit package. Install VisualAge C++ for Windows and FixPak WTC354.
7. Microsoft Visual C++ is a prerequisite for the ActiveX client development environment only.
8. IBM jdk 1.1.7B is a prerequisite for the Java client development environment only.
9. Microsoft Internet Explorer 3.0 and above is a prerequisite for the ActiveX client.

3.5 Component Broker packages

There are several separately installable components on the Component Broker and CBToolkit compact discs. The Component Broker compact disc includes runtime services. The CBToolkit compact disc includes application development tools and Software Development Kits (SDKs) consisting of header and library files for use in server or client development.

3.6 Component Broker compact disc contents

The separately installable components on the Component Broker compact disc include:

Component Broker Server

Install this component on a system that you want to function as an object/component server.

C++ Client

Install this component if you want to run Component Broker C++ CORBA client applications developed using VisualAge C++ for Windows on a system.

Java Client

This is similar to the C++ client, but is implemented in Java and supports Java client applications. The Java client is a CORBA client.

ActiveX Client

This component enables the development of client applications that use ActiveX/COM interfaces to Component Broker server objects. This component contains a C++ client ORB. The generated parts can be used from any ActiveX-enabled language, including VisualBasic.

System Manager

Component Broker products and applications deployed on a host can be managed by the System Manager if the System Management Agent component is installed on the host. Any host can serve as a System Manager platform if the System Manager is installed. The System Manager does not need to reside on each host and is typically executed on a limited number of hosts in an enterprise. Either the System Manager or the System Management Agent can be installed on a host, but not both.

System Management User Interface

The System Management User Interface can be installed on any host and does not need to co-reside with a System Management Agent or System Manager. There can be multiple instances of the System Management User Interface installed and active at any point in time. Each System Management User Interface is connected to one System Manager.

System Management Agent

Component Broker products and applications deployed on a host can be managed by the System Manager if the System Management Agent component is installed on the host. Either the System Manager or the System Management Agent can be installed on a host, but not both.

Application Adaptor

The DB2 Application Adaptor provides a location for instances of Managed Objects. It is responsible for providing system capabilities for its Managed Object instances. You must select the DB2 Application Adapter if the server is to be installed on your system.

Documentation

The complete set of online documentation in HTML 3.2 format is included on the CBToolkit compact disc.

Runtime Samples

A complete set of sample applications that accompany Component Broker is available to install from either the Component Broker compact disc or from the CBToolkit compact disc.

IBM VisualAge C++ for Windows FixPak WTC354

The required FixPak for IBM VisualAge C++ for Windows is Version 3.5.3. VisualAge C++ for Windows is required only for application development.

3.7 Component Broker Supplemental Programs compact disc contents

The Component Broker Supplemental Programs compact disc contains software for the Windows NT platform.

DB2 CAE (Client Access Enablement), Version 2.1

If you do not have a DB2 Server installed on the system, you need to install this component to support Component Broker. This compact disc contains the DB2 CAE for Windows 95 and Windows NT for version 2.1 in directory *db2c*.

DB2 CAE (Client Access Enablement), Version 5.0

If you do not have a DB2 Server installed on the system, you need to install this component to support Component Broker. This compact disc contains the DB2 CAE for Windows 95 and Windows NT for version 5.0 in directory *db2v5c*.

DCE Client

These DCE client files contain the application developers kit, the runtime environment, and tutorials for DCE. This compact disc contains the DCE Client for Windows 95 and Windows NT.

DCE FixPak DCENT11CE03 for Windows NT

The required FixPak for DCE for Windows NT.

3.8 CBToolkit compact disc contents

The application development tools can be installed on any computer with a configuration that includes VisualAge C++, access to DCE and DB2, and a Component Broker server or client. Using the application development tools, you can develop Component Broker applications on one system (for example, a client) and deploy the applications on another. The CBToolkit compact disc includes:

Application Development Tools

A set of visual tools used to define, implement, deploy, and debug distributed Component Broker applications.

Server SDK

Header and library files for developing applications on the server.

Client SDKs

Header and library files for developing applications on the Component Broker clients. The ActiveX Client SDK, CORBA Client SDK, and Java Client SDK are included.

If you install the ActiveX client SDK on the same system on which a C++ client is installed, you need to create a two-compiler environment. For more information, see Appendix B, “Tips and hints” on page 407.

CICS and IMS Application Adaptor SDK

Header and library files for developing applications with the Component Broker CICS and IMS Application Adaptor.

TKSamples

A set of buildable (source) sample applications.

3.9 Installation considerations

The installation considerations and the installation process is fully documented in the *IBM Component Broker – Quick Beginnings Guide*, G04L-2375. This section includes considerations for planning your installation. Considerations include:

- **Installation**

  Component Broker must be installed on a local drive. However, the install image can be accessed through a network drive. Before you install Component Broker, you need to plan your site to determine which Component Broker components to install on which computer. Because of software interactions, you must install the Component Broker software and prerequisites in the following order:

  1. Microsoft Windows NT 4.0
  2. DB2
  3. VisualAge C++ for Windows (if installing the CBToolkit package)
  4. NT Service Pack 4
  5. DCE
  6. DCE FixPak DCENT11CE03
  7. Component Broker package
  8. CBToolkit package

- **DCE**

  All Component Broker servers must have access to DCE 1.1c (also known as DCE 1.1.1) for object naming purposes. You can choose to install DCE on the same host as the System Manager, or you can install DCE on a remote host. You must determine which systems will be DCE Security and Name servers. You can install DCE on a single computer, or you can break it up into its component parts and install each server on separate computers. The DCE runtime must be installed on each Component Broker computer. Dynamic IP addressing (DHCP) is not supported.

- **DB2**

  All Component Broker servers must have access to DB2. You can choose to install DB2 on the same host as the System Manager, or you can install DB2 on a remote host.

- **VisualAge C++ for Windows**

  VisualAge C++ for Windows must be installed on each computer intended for application development. After installing VisualAge C++ for Windows, you must install the Windows NT Service Pack 3 again.

- **Component Broker**

  Please read the *IBM Component Broker – Quick Beginnings Guide*, G04L-2375 in order to designate a bootstrap host. A bootstrap host is a host from which remote client computers can bootstrap into the system name space. Every host computer that has the Component Broker server installed can be a bootstrap host. Decide which host computer each Component Broker computer will use for its bootstrap host. Not all client computers need to use the same bootstrap host. You should limit the number of computers that share the same bootstrap host to reduce the impact to your system if one of the bootstrap hosts should fail. Assign a port ID to use for each bootstrap host.

  The port ID you supply during Component Broker installation must match the port ID configured with the corresponding bootstrap host in its Component Broker TCP/IP Protocol Image. The TCP/IP Protocol Image for a host can use the default System Management port ID (900). If you do not know what port ID to use, accept the default value and change it later if that port ID is already in use. Designate one or more computers as Component Broker System Manager hosts. Each System Manager host manages one or more Component Broker clients and servers to form a Management Zone of systems. Each managed Component Broker client or server belongs to one Management Zone. During Component Broker installation, you need to provide the host name for the System Manager host system.

  All host names defined during installation and management of Component Broker should be fully-qualified names. Determine locations for the System Management User Interface. The System Management User Interface can be installed on a computer other than the computer on which the System Manager is installed. Determine how you want applications to be developed and deployed. Applications can be deployed only from a server, but can be developed on either a server or a client. If you have a large site, you might not want a computer to share deployment and System Manager responsibilities, and you might want to mirror the applications to more than one server. Consider installing the System Manager as part of your first Component Broker installation. The System Manager is used to manage the other Component Broker servers and clients.

Note:

If you are installing Component Broker in a development environment, your first Component Broker installation must be a development server with VisualAge C++ from Windows installed as a prerequisite.

Consider performing the Server Package installation option on other computers that you want to be CBConnector servers. During the installation, you must provide the host name and port ID used for System Management services. Provide the information you specified during the Typical Install installation.

Consider installing the typical clients on hosts that you want to run client applications that use CBConnector applications. You must supply the System Manager host name and port ID and information about boostrap hosts. Provide the information you specified during the Typical Install installation. A frames-capable, HTML 3.2-compatible browser is required to view the CBConnector online documentation.

- **Web Server**

  A Web server is required if Java Applets that are developed to enable Java clients are going to be downloaded from the World Wide Web.

- **CBConnector Runtime**

  CBConnector Runtime (server and client) comes with an imbedded Java Runtime Environment (JRE 1.1.4). You only need the JDK if you want to develop client or server applications in Java, using either the Java client or using the CBConnector Object Model (SOM4) functions for developing C++ or Java applications. The Java 1.1.4 JDK for Windows NT can be obtained over the World Wide Web from the JavaSoft home page.

  Component Broker does not work with the Microsoft Java Runtime Environment (JRE).

3.10 Installation options and required prerequisites

Component Broker components installed from the CBConnector compact disc include:

Typical Install

This installation option allows you to define and manage servers as well as develop and deploy applications on your desktop. You should consider using this installation procedure for your first installation.

Typical Clients

This installation option installs the C++ and Java clients, allowing the user to run existing Component Broker applications.

Server Package

This installation option includes the Component Broker server, the C++ client, and the System Management Agent. This package is useful for a deployment system for applications that are to be developed and managed from other hosts.

System Management Workstation

This installation option includes the System Manager and the System Manager User Interface and defines a host that can be used to manage other Component Broker hosts.

Custom Install All Available Packages

This installation option enables you to select any or all of the Component Broker components for installation. Component Broker components installed from the CBToolkit compact disc include:

Adding the Development Environment

This installation option installs the Component Broker application development environment. You can specify to install development tools and software.

Development Kits

These include:

- CBToolkit
- Server SDK
- CORBA Client SDK
- ActiveX Client SDK
- Java Client SDK
- CICS and IMS Application Adaptor SDK
- TKSamples

Component Broker supports ERwin/ERX, Version 3.0 as a database design tool and Rational Rose/C++, Version 4.0.5 as an application design tool.

The following tables list the Component Broker components you can choose to install on your system from the CBConnector compact disc, and the CBToolkit compact disc and the prerequisites that apply to each component. Components and related prerequisites that you can install in a development environment are shown below.

Component	**Windows NT 4.0**	**VisualAge C++ for Windows**	**Access to DCE**		**Access to DB2**
			Security	**Directory**	
Server	X	X	X	X	See Note 3
C++ Client	X	X	X	X	
ActiveX Client	X	See Note 1	x		
Java Client	See Note 2	x	x	x	
System manager	x				
System Management User Interface	x				
System Management Agent	x				
Application Adaptor	x				
Documentation	x				
TKSamples	x	x			
Server SDK	x	x			
C++ Client SDK	x	x			
ActiveX Client SDK	x	See Note 1			
Java Client JDK	See Note 2				
CBToolkit	x	x			See Note 3
CICS and IMS Application Adpator	x	See Note 4			See Note 3

Notes:

1. Basic samples provided with ActiveX client require Microsoft Visual Basic and Visual C++ Version 5.0
2. IBM jdk 1.1.7B
3. The DB2 SDK is required for developing Component Broker applications.

The table below lists components and related prerequisites that you can install in a runtime environment.

Component	**Windows NT 4.0**	**Access to DCE**		**Access to DB2**
		Security	**Directory**	
Server	X	X	X	x
C++ Client	X	X		
ActiveX Client	X	x		
Java Client	See Note 1	x		
System Manager	x			
System Management User Interface	x			
System Management Agent	x			
Application Adaptor	x			
Documentation	x			
1.Note: IBM jdk 1.1.7B				

Chapter 4. Test your development system

Component Broker Connector is a framework with many components. Before you start your own application development, it is a good idea to make sure that your development environment is installed and functioning correctly. For CBConnector installation, refer to *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - Getting Started with Component Broker*, SC09-4433.

We developed two small Ping applications serving this purpose. You can find them on the CD-ROM in the following subdirectories:

- *04-Transient-Ping*
- *04-Persistent-Ping*

They include the CBConnector servers and client programs. CBConnector Release 1.2 supports server Business Objects created in C++. and Java. On the client side, we developed these samples for all currently supported clients. That means C++, CORBA, Java, and ActiveX clients.

One server application supports only transient Managed Objects. The other server application uses the DB2 Application Adaptor to store the essential state of data in the DB2 database.

These test applications use all the important basic components of Component Broker Connector, such as:

- Object Request Broker
- Naming Service
- LifeCycle Service
- Transaction Service
- Server Runtime
- DB2 Application Adaptor
- System Management

At this point, we do not want to explain how the Ping applications were created, but we do recommend that you follow fully the instructions in this chapter to install and execute the test Ping applications. It will help you to become familiar with the CBConnector System Management User Interface and the distributed, multi-tier application environment.

4.1 Get ready

As me mentioned before, Component Broker Connector has a multi-tier, distributed infrastructure; so you can execute both the server and client applications directly on the CBConnector server or execute them separately on two different computers. We would like to advise you that you would typically install and run the Ping applications first on your newly installed CBConnector server. Afterwards, you can install the Ping test clients on the CBConnector client machine and test the remote environment.

To install the test Ping application server and client programs, prepare your environment in the following way:

- Install CBConnector Runtime, CBConnector client and CBToolkit on your CBConnector server and client as described in *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - Getting Started with Component Broker*, SC09-4433. In the case of ActiveX and CORBA clients, you should be aware of the two-compiler environment as described in this book in B.1, “Two-compiler environment” on page 407.

  No part of the CBToolkit is necessary to run the tests, but we expect that you are testing your CBConnector development environment; so your CBToolkit is installed. You can use the Object Builder to explore the model data of the samples, which are also on the CD-ROM in above mentioned directories.
- Install the DB2 server with development toolkit. The samples using a relational database will require certain header files available only in the toolkit.
- If you installed IBM jdk 1.1.7B, include the current path (.\) in the `CLASSPATH`.
- If you are using more than one computer, assure that there is a TCP/IP protocol properly configured between them.
- Because we are using different implementations for the same interface, we have to install and configure single Location Objects. The Location Objects are used to limit the search scope of the factory finder and guarantee that the appropriate implementation is found. Chapter 18, “LifeCycle Service” on page 287 covers more details about name scopes.

  For each server application, the Systems Management Application automatically creates location objects with these naming conventions:
  - <servername>-server-scope
  - <servername>-server-scope-widened

4.2 Install and run the transient Ping test application

When you complete all the mandatory steps we described above in 4.1, “Get ready” on page 30, you are ready to install the transient Ping test application. Follow the instructions in the subsequent sections for both server and client installations. In case you are testing the Ping test applications from the CBConnector client, please repeat the installation instructions under sections "Install and Run the Client Programs".

4.2.1 Install the transient Ping application server

This section describes the installation of the server application. The installation includes the following steps:

- Installation of the server application
- Configuring the application server
- Activation of the configuration
- Starting the configured server

If you encounter any problems during these steps, control carefully the execution of all steps and refer to 4.2.1.3, “Troubleshooting” on page 34, and to the troubleshooting chapter in the CBConnector online documentation.

4.2.1.1 Install the server application

In this step, you will install the server application with all its libraries. The installation procedure also includes the update of the Interface Repository and the System Management data.

When you develop an application, the object builder generates in a directory called "PRODUCTION" the files you need to copy to the target server in order to install and run your application.

This is what we do to install an application.

Start the Systems Management Application and ensure that the Name Server is running:

1. From the **Host images -> your current host,** click mouse button 2 and select **Load Application** from the pop-up menu.
2. In order to pick up the DDL file, click **Browse** from the Load Application window. Navigate to the CD-ROM directory:

   <drive>:\CBConnector\04-Transient-Ping\Server\Working\Nt\PRODUCTION\PingTRAppFam

Select **PingTRAppFam.ddl** and click **OK.**

From the Action Console window, you can check that the DDL file was processed successfully. At the end of this process, the required files to run your application have been copied in:

<drive>:\cbroker\data\ntApps3.0\PingTRAppFam

Now you are ready to configure the server Ping application.

4.2.1.2 Configure the Ping application server

Once you have finished installing the application on your system, you have to configure it with System Management.

The general procedure always contains the following steps and is done by using the System Management User Interface:

- Create a model configuration by configuring the installed object to an application server model. The installed objects are created by the previous installation step.
- Register the server model with a host model. This tells the system on which host the server should be executed.
- Activate the model configuration. This includes an intensive checking of the consistency of the model data and the generation of a server image.

For more information about the System Management, refer to *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - System Administration Guide*, SC09-4445 and to the System Management online documentation.

To configure the application in the CBConnector System Management, follow these steps:

1. Start the System Management User Interface. Set the user level to **Super User**.
2. Create a new Management Zone and a new configuration for this test:
   1. Find the Management Zones folder.
   2. From the pop-up (right click) menu of Management Zones, click **Insert**. A dialog box appears.
   3. Type `Ping TR Management Zone` as the name for the new Management Zone.
   4. Expand the Management Zone folder and select the newly created Ping Transient Management Zone.
   5. From the pop-up menu of that Management Zone, click **New -> Configuration**. A dialog box appears.

    6. Type `Ping TR Configuration` as the name for the configuration.

3. Configure the application with a Management Zone. Note that the user level must be set to **Super User** for the following options to appear:

    1. Scroll to the bottom of the window.
    2. Find the **Host Images** folder and expand it.
    3. Under Host Images, find the host image for your current system and expand it.
    4. Under the host image, find the **Application Family Installs** folder and expand it.
       You can now see the entry for the PingTRAppFam application family. These installation objects were loaded into System Management by the setup in the previous task.
    5. Expand the PingTRAppFam folder; then expand the Application Installs folder. The PingTRS application appears below it.
    6. From the pop-up menu of the PingTRS application, click **Copy**.
    7. Scroll back to the top of the System Management window. Find the configuration you just created in the Management Zones folder, and expand it.
    8. From the pop-up menu of Ping TR Configuration, click **Add Application**.
    9. Under Ping TR Configuration, expand Applications. The PingTRS application now appears in the Applications folder.

4. Configure a server:

    1. Find Management Zones **-> Ping TR Management Zone -> Ping TR Configuration**.
    2. From the pop-up menu of Ping TR Configuration, click **New -> Server (free standing)**. A dialog box appears.
    3. Type `PingTRServer` as the name for the free-standing server.
       Note: Normally, you are free to choose any name you like as a server name. For this test, you must use exactly this name to enable the execution of the client programs because the Location Objects refer to it.
    4. Click **OK**. The PingTRServer now appears under Server (free standing).

5. Associate the configured application with the server:

    1. Find your configured PingTRS application under Management Zones **-> Ping TR Management Zone -> Ping TR Configuration**.
    2. From the pop-up menu of PingTRS, click **Copy**.
    3. Find the PingTRServer server you defined under Management Zones, select **-> Ping TR Management Zone -> Ping TR Configuration -> Server (free standing)**.

   4. From the pop-up menu of PingTRServer, click **Configure Application** A Configured Applications folder now appears under PingTRServer. You can expand the folder to display the entry for PingTRS.

6. Configure the server with the host:

   1. From the pop-up menu of PingTRServer, click **Copy**.
   2. Expand the Hosts folder and find the host for your current system.
   3. From the pop-up menu of the host, click **Configure Server (free standing)**. Under the host, there is now a folder called Configured Servers, that contains an entry for the PingTRServer server.

7. Finally, activate this new configuration:

   1. Find Management Zones and select **-> Ping TR Management Zone -> Ping TR Configuration**.
   2. From the pop-up menu of Ping TR Configuration, click **Activate**. This process may take a while, as CBConnector implements the configuration. When the activation is finished. You should see a message in the activation window that the activation completed successfully on your host. Also, when the activation is completed, the System Manager status line should show the following information (or similar):

```
PingTRServer: run immediate running 0 0 excellent
```

Your server is now up and running.

In case you want to stop and restart your server, do the following:

1. Find your server image, PingTRServer, under **Host images -> your current host -> server images**.
2. From the pop-up menu of PingTRServer, click **Stop Immediate**. CBConnector stops the application server.
   Restarting the server is done in a similar fashion:
3. In **Host images -> your current host -> server images**. do **Run Immediate**. An action console window appears. This process may take a while as CBConnector starts the application server. Wait until the messages indicate a successful start of the application server. Then close the action console window.

4.2.1.3 Troubleshooting

In the normal case, the installation and activation should work without any problems. If you encounter any problems during these steps, refer to the troubleshooting information in the CBConnector online documentation.

The most detailed information you get is from the activity log in the *\CBroker\service* directory. You can dump and format the CBConnector activity log by using the command:

```
showlog activity.log -debug > <filename of your choice>
```

Search after the last events connected to your application server.

It is also helpful to take a look at the CDS and DCE user directory. In the DCE user directory, there should be the following system-generated user:

- CBConnectorSM
- A user for the Name Server
- PingTRServer

In the DCE CDS, you should see the following entries:

- CBC-local-root
- CBC-root
- CBC-workgroup

If any entry or user is missing, check your installation.

4.2.2 Run the client programs

In this section, we describe the execution of the different client programs.

4.2.2.1 C++ Client

Run the application

To run the client application, follow these steps:

1. Open a command shell window.
2. Change to the directory: *<drive>:\CBConnector\04-Transient-Ping\Client\VisualAgeCpp\Executable*
3. Enter following command to run the Ping test client application:

```
PingTRC <PingId> <number of Pings>

For example:

PingTRC 1 100
```

The application log should look like this:

```
*** Start of C++ transient Ping Client Program for CBC v. 3.0 ***
1) About to initialize the ORB
         The ORB initialized successfully

2) Now attempt to resolve to root naming context
3) About to resolve the factory finder
         FactoryFinder resolved
4) Now attempt to get a Ping Home Object.
         Ping Home found
5) Now attempt to create a Ping Key Object.
         About to create a PingKey
         Primary key created successfully
6) About to find Ping Object with PingHome.findByPrimaryKeyString
         Ping Object found with PingHome.findByPrimaryKeyString
         About to narrow MO to PingMO
         Managed object narrowed to PingMO
7) Now executing the Ping method 100 times.
8) Now calculate the average response time and set the Ping attributes
         Number of Pings: 100 average response time: 6 ms

Program ran SUCCESSFULLY!
************* End of Sample Program ***********
```

Once the application finished running, a new transient Ping Object was instantiated in memory with values for its attribute's Ping ID, counter and avgResponseTime.

If you encounter any error messages, refer to the troubleshooting 4.2.2.4, “Troubleshooting” on page 40.

4.2.2.2 Java client

For our test scenario, we only provide a simple Java client program that can be run without any browser or applet-viewer. But if you are interested in a applet version of the Java client, you can easily generate this by using the Java code we supply on the CD-ROM in the directory: *<drive>:\CBConnector\04-Transient-Ping\Client\Java\Source*.

Run the Java client program

To run the Java client application, follow these steps:

1. Open a command shell window.
2. Check that there is a reference to the local path in the `classpath` by typing `set classpath` and look for a dot (.) in the path. Check also that the necessary Java class archives are in the `classpath`. These are named *somojor.zip* and *IBM jdk 1.1.7B/lib/classes.zip*.

   You need to add the jar file containing the Ping classes to the classpath. You can do it by entering from the command prompt the following command:

   SET CLASSPATH=.\jcbPingTRC.jar;%CLASSPATH%

   The jcbPingTRC.jar file was generated by Object Builder and put in the directory:

   <drive>:\CBConnector\04-Transient-Ping\Server\Working\Nt\PRODUCTION\Jcb

   You can also adapt the runit.cmd file by changing the hostname "atlantic.almaden.ibm.com" by your fully qualified hostname and run it!

   Also check the instructions for Java clients in 4.1, “Get ready” on page 30.
3. Change to the directory:

   <drive>:\CBConnector\04-Transient-Ping\Client\Java\Class
4. Enter the following command to run the Ping test client application:

```
java PingTRJ <PingId> <number of Pings> <hostname.domain.company.com>
<port number>

For example:

java PingTRJ 2 100 cbcsrva.itsc.austin.ibm.com 900
```

5. The application should show a log as follows:

```
*** Start of Java transient Ping Client Program for CBC v. 3.0***

1) About to call ORB.init passing Bootstrap information as a property list:
          ORBInitialHost = The host name of the initial CBConnector server
          ORBInitialPort = The bootstrapPing port number to connect to
2) Now attempt to resolve to root naming context using IExtendedNaming.
          orb.resolve_initial_references("NameService")
          NamingContextHelper.narrow(obj)
3) About to resolve the factory finder.

iExtendedNC.resolve_with_string("host/resources/factory-finders/PingTRScope")
          FactoryFinderHelper.narrow(obj)
          FactoryFinder resolved
4) Now attempt to get a Ping Home Object.
          factoryFinder.find_factory_from_string("Ping.object interface")
          IHomeHelper.narrow(obj)
5) Now attempt to create a PingKey Object.
          About to create a PingKey
         Primary key created successfully
          PingKey created
6) About to find Ping Object with PingHome.findByPrimaryKeyString
          Ping Object found with PingHome.findByPrimaryKeyString
7) Now executing the Ping method 100 times
8) Now calculate the average response time and set the Ping attributes
          number of Pings: 100 average response time: 13 ms

Program ran SUCCESSFULLY
************* End of Sample Program ***********
```

Once the application finished running, a new transient Ping Object was instantiated in memory with values for its attribute's Ping ID, counter and avgResponseTime in the memory.

If you encounter any error messages, refer to Section 4.2.2.4, “Troubleshooting” on page 40.

4.2.2.3 ActiveX Client

Run the ActiveX application

To run the ActiveX client application, follow these steps:

1. Move to the CD-ROM directory:

   <drive>:\CBConnector\04-Transient-Ping\Client\ActiveX\Vb

2. Run the register.bat command file in order to register the Ping DLLs to the Windows registry. You can check that the DLLs have been correctly registered by running the OLE/COM Object Viewer tool provided with VisualC++ from Microsoft. By expanding the Automation Objects tree node, you should find:
   - CPing
   - CPingCopy
   - CpingKey
3. Start the PingTRX.exe application.
4. After a while, a GUI window with entry fields pops up. Fill in the numbers and press the **StartPing**-button.
5. The application generates the cbc_vb_out.log file, which should show a log as follows:

```
Starting Ping Sample
About to initialize the ORB
  Initialization successful
Attempt to resolve root naming context
  narrow the NameService-Object
  Got the root naming context
About to resolve the FactoryFinder
  FactoryFinder resolved
Attempt to get a Home Object.
  Home found
Attempt to find object by Primary Key
  About to narrow to PingMO
  narrowed
  Object found
Pinging...
Average Time for one Ping: 10 msec
```

Once the application finished running, a new transient Ping Object was instantiated in memory with values for its attribute's PingID, counter, and avgResponseTime.

If you encounter any error messages, refer to 4.2.2.4, “Troubleshooting” on page 40.

4.2.2.4 Troubleshooting

All errors caught during the execution of the application are shown on the standard output console in a format such as this:

ERROR: <the error message>

In the following paragraphs, possible problems during each execution step of the client application are listed, together with some hints to solve each problem.

1. Step 1 of the client program fails. The client is not able to initialize the ORB.
   - Check the `SOMCBENV` environment variable by typing `Set SOMCBENV` on the command line. This variable should be set to:

```
SOMCBENV=C:<your host name> Default Client
```

   - If you are using more than one computer, make sure that there is an TCP/IP connection between the client and server system. Use the `Ping` command to check this:
     Ping <your server system IP name>
   - If you get the following error message using the Java client:

```
----------------------------------------
1) About to call ORB.init passing Bootstrap information as a property list:
         ORBInitialHost = The host name of the initial CBConnector server
         ORBInitialPort = The bootstrapPing port number to connect to
org.omg.CORBA.NO_IMPLEMENT:   minor code: 0  completed: No
        at COM.ibm.som.corba.rt.DelegateImpl.getPseudoOrb(DelegateImpl.java:67)
        at COM.ibm.som.corba.rt.ORB.set_parameters(ORB.java:86)
        at org.omg.CORBA.ORB.init(ORB.java:50)
        at PingTRJ.start(PingTRJ.java:140)
        at PingTRJ.main(PingTRJ.java:108)
```

     You should check whether you have installed the necessary Java class or class archives files for Component Broker Connector

     Note: For executing a Java program, you need the class archives, somojor.zip and IBM jdk 1.1.7B\lib\classes.zip, in your `classpath`.

2. Step 2 of the client program fails.
   The system is not able to resolve the root naming context.
   - Check whether the Name Server is running.

3. Step 3 of the client program fails.
    - The system is not able to find a factory finder.
4. Step 6 of the client program fails.
   The client is not able to find or create a Ping Object.
    - ERROR: Something went wrong during find.
    - Make sure that your application server is running.
5. If you get following exception during any step of the program:
   CORBA::SystemException CORBA::no_RESPONSE
    - Check your time-out interval. You can increase it with the command:

```
set SOMDREQUESTTIMEOUT=<your timeout value in seconds>
```

4.3 Install and run the persistent test program

Now we extend the test scenario from the last chapter with the access to a relational database. The application logic and the structure of our Ping Object do not change. We only add a new service to the Managed Object which stores the essential data of the object in a row of a database table.

4.3.1 Install the persistent Ping application server

This chapter describes the installation of the server application. The installation includes the following steps:

1. Installation of the server application
2. Creation of the database and database table
3. Configuring the application server
4. Activation of the configuration
5. Starting the configured server

If you encounter any problems during these steps, carefully control the execution of all steps, and refer to 4.3.1.4, “Troubleshooting” on page 47 and 4.2.2.4, “Troubleshooting” on page 40.

4.3.1.1 Install the application server

In this step, the server application with all its libraries is installed. The installation procedure includes also the update of the Interface Repository and the System Management data.

When you develop an application, the object builder generates in a directory called "PRODUCTION" the files you need to copy to the target server in order to install and run your application.

This is what we do to install an application.

Start the Systems Management Application and ensure that the Name Server is running.

1. From the **Host images -> your current host**, click mouse button 2 and select **Load Application** from the pop-up menu.
2. In order to pick up the DDL file, click **Browse** from the Load Application window and navigate to the CD-ROM directory:

   <drive>:\CBConnector\04-Persistent-Ping\Server\Working\Nt\PRODUCTION\PingDBAppFam

   Select **PingDBAppFam.ddl** and click **OK**.

   From the Action Console window you can check that the DDL file was processed successfully. At the end of this process, the required files to run your application have been copied in:

   <drive>:\cbroker\data\ntApps3.0\PingDBAppFam

Now you are ready to configure the Ping server application. Prior to that, we let you install the PingDB database with the Ping table that was previously created.

4.3.1.2 Create the database

The following steps describe how to create the Ping database and bind the access package to the database.

1. Open a command shell window.
2. Change the directory to *<drive>:\CBConnector\04-Persistent\Install*
3. Run following command where `userid` and `password` must belong to a user with DBADMIN authority:

```
db2cmd makePingDB <userid> <password>
```

4. Next you have to bind your database. Move to *<drive>:\CBConnector\04-Persistent\Server\Working\Nt\PRODUCTION* and execute the following command:

```
db2cmd db2 connect to PingDB user <userid> using <password>
db2 bind PingPO.bnd
```

You can cross-check the success of this operation using the DB2 database director tool, or from a DB2 command line connected to the PingDB database, by issuing a `SELECT * FROM Ping` statement querying the Ping table. You should not see any entries in the Ping table at this point.

4.3.1.3 Configure the Ping application server

Once you have finished installing the application on your system, you have to configure it with System Management. For the persistent test program, you must not only configure the Ping application, but also the DB2 Application Adaptor services (iDB2IMService application) that are needed to access the relational database.

To configure the application with System Management, follow these steps:

1. Start the System Management User Interface. Set the user level to **Super User**.
2. Create a new Management Zone and a new configuration for this test:
   - Find the Management Zones folder
   - From the pop-up menu of Management Zones, click **Insert**. A dialog box appears.
   - Type `Ping DB Management Zone` as the name for the new Management Zone.
   - Expand the Management Zone folder and select the newly created Ping DB Management Zone.
   - From the pop-up menu of that Management Zone, click **New -> Configuration**. A dialog box appears.
   - Type `Ping DB Configuration` as the name for the configuration.
3. Configure the application with a Management Zone. Note that the user level must be set to **Super User** for the following options to appear:
   - Scroll to the bottom of the window.
   - Find the Host Images folder and expand it.

- Under Host Images, find the host image for your current system and expand it.
- Under the host image, find the Application Family Installs folder and expand it.
  You can now see the entry for the PingDBAppFam and Specific PingDBAppFam application family. These installation objects were loaded into System Management by the setup in the previous task.
- Expand the PingDBAppFam folder; then expand the Application Installs folder. The PingDBS application appears below it.
- From the pop-up menu of the PingDBS application, click **Copy**.
- Scroll back to the top of the System Management window.
- Find the configuration you created for the Persistent Ping Object in the Management Zones folder, Ping DB Management Zone, and expand it.
- From the pop-up menu of Ping DB Configuration, click **Add Application**.
- Do the same for the Specific PingDBS application in the Specific PingDBAppFam application family.
- Under Ping Configuration, expand Applications. The PingDBS and Specific PingDBS applications now appear in the Applications folder.

4. Configure the iDB2IMService application with the Ping DB Management Zone:
   - Scroll to the bottom of the window.
   - Find the Host Images folder and expand it.
   - Under Host Images, find the host image for your current system and expand it.
   - Under the host image, find the Application Family Installs folder and expand it.
     You can now see the entry for the iDB2IMApplication application family. The DLL file for this application was loaded into System Management during the installation of the Component Broker Runtime.
   - Expand the iDB2IMApplication folder; then expand the Application Installs folder. The iDB2IMService application appears below it.
   - From the pop-up menu of the iDB2IMService application, click **Copy**.
   - Scroll back to the top of the System Management window.
   - Find the configuration you just configured the PingDBS.
   - From the pop-up menu of Ping Configuration, click **Add Application**.

- Under Ping DB Configuration, expand Applications. The iDB2IMService application now appears in the Applications folder.

5. Configure a server:
   - Find Management Zones and select **-> Ping DB Management Zone -> Ping DB Configuration**.
   - From the pop-up menu of Ping DB Configuration, click **New -> Server (free-standing)**. A dialog box appears.
   - Type `PingDBServer` as the name for the free-standing server. Normally, you are free to choose any name you like at this point. But in our case, you must use exactly this name to enable the execution of the client programs because the Location Objects refer to it.
   - Click OK. The PingDBServer now appears under Server (free standing).
6. Associate the configured application with the server:
   - Find your configured PingDBS application under Management Zones and select **Ping DB Management Zone -> Ping DB Configuration**.
   - From the pop-up menu of PingDBS, click **Copy**.
   - Find the PingDBServer server you defined under Management Zones and select **Ping DB Management Zone -> Ping DB Configuration -> Server (free-standing)**.
   - From the pop-up menu of PingDBServer, click **Configure Application**. A Configured Applications folder now appears under PingTRServer. You can expand the folder to display the entry for PingDBS.
   - Proceed in a similar way with the Specific PingDBS and iDB2IMService application. You should now have all three applications in the Configured Applications folder under PingTRServer.
7. Configure the server with the host:
   - From the pop-up menu of PingDBServer, click **Copy**.
   - Expand the Hosts folder and find the host for your current system.
   - From the pop-up menu of the host, click **Configure Server (free-standing)**. Under the host, there is now a folder called Configured Servers that contains an entry for the PingDBServer server.
8. Finally, activate this new configuration:
   - Find Management Zones and select **-> Ping DB Management Zone -> Ping DB Configuration**.

- From the pop-up menu of Ping DB Configuration, click **Activate**. This process may take a while, as CBConnector implements the configuration. When the activation is finished. You should see a message in the activation window that the activation completed successfully on your host. Also, when the activation is completed, the System Manager status line should show the following information (or similar):

  ```
  PingDBServer: run immediate running 0 0 excellent
  ```

Your server is now up and running...almost. This server application will use the DB2 Application Adaptor. To make this possible, you have to first configure the XA Resource Manager. Here are the steps:

1. Stop the server by finding your server image, PingDBServer, under **Host images -> your current host -> server images**.
2. From the pop-up menu of PingDBServer, click **Stop Immediate**. CBConnector stops the application server.
3. From the System Management User Interface, open your Host Images folder and expand your host image. Select **Host Images -> (your host image) -> Server Images -> PingDBServer -> XA Resource Manager Images -> PingDB**.
4. From the pop-up menu, click **Edit**.
5. Select the **Main** tab, and edit the **open string** attribute box. Currently, this field should only contain the name of the Ping database, PingDB. Modify this field to also include the user ID and password for accessing this database. Change `PingDB` to `PingDB, <user ID>, <password>`.
6. Click on **Validate -> Apply -> OK** to save these changes.
7. Follow the same steps for all other entries under the XA Resource Manager Images-folder.
8. Close the PingDBServer image.
9. From the pop-up menu of PingDBServer, click **Run Immediate**. An action console window will appear. This process may take a while as CBConnector starts the server process. Wait until the messages indicate a successful start of the application server. Then close the action console window.

   You can check the run status of your server by selecting the server image and looking at the status line of the System Management User Interface window. The status line should show a message like this:

   ```
   PingDBServer run immediate running 4 4 excellent
   ```

4.3.1.4 Troubleshooting

In this troubleshooting section, only the problem situations specific to the persistent application server are described. If you have problems activating or starting the application server, you may also refer to 4.2.1.3, “Troubleshooting” on page 34.

The server fails to Run Immediate.

Check the following points:

- Was the PingDB database successfully created as described in 4.3.1.2, “Create the database” on page 42?
- Was the bind of the bind file PingPO.bnd successfully executed?
- Check whether you have set the user ID and password in the XA Resource Manager Images to access the database.

4.3.2 Run the client programs

In this section, we describe the execution of the different client programs.

4.3.2.1 C++ Client

Run the application

To run the client application, follow these steps:

1. Open a command shell window.
2. Change to the directory *<drive>:\CBConnector\04-Persistent-Ping\Client\VisualAgeCpp\ Executable*
3. Enter the following command to run the Ping test client application:

```
PingDBC <PingId> <number of Pings>

For example:

PingDBC 1 100
```

4. The application should show a log as follows:

```
*** Start of C++ persistent Ping Client Program for CBC v. 3.0***
1) About to initialize the ORB
         The ORB initialized successfully

2) Now attempt to resolve to root naming context
3) About to resolve the factory finder
         FactoryFinder resolved
4) Now attempt to get a Ping Home Object.
         Ping Home found
5) Now attempt to create a Ping Key Object.
         About to create a PingKey
         Primary key created successfully
6) About to find Ping Object with PingHome.findByPrimaryKeyString
         Ping Object found with PingHome.findByPrimaryKeyString
         About to narrow MO to PingMO
         Managed Object narrowed to PingMO
7) Now executing the Ping method 100 times.
8) Now calculate the average response time and set the Ping attributes
         Number of Pings: 100 average response time: 59 ms

Program ran SUCCESSFULLY!
************* End of Sample Program ***********
```

Once the application has finished running, a new Ping Object was instantiated with values for its attributes Ping ID, counter, and avgResponseTime. In this case, the Ping Object is a persistent one, and it stores its essential state of data to a datastore.

Check the contents of the datastore with the commands described in 4.3.3, “Check the datastore” on page 52.

If you encounter any error messages, refer to 4.2.2.4, “Troubleshooting” on page 40 and 4.3.2.4, “Troubleshooting” on page 52.

4.3.2.2 Java client

Now you can run the Java version of the test Ping application.

Run the Java client program

To run the Java client application, follow these steps:

1. Open a command shell window.
2. Check that there is a reference to the local path in the `classpath` by typing `set classpath` and look for a dot (.) in the path. You need to add the jar file containing the Ping classes to the classpath. You can do it by entering from the command prompt the following command:

   SET CLASSPATH=.\jcbPingDBC.jar;%CLASSPATH%

   The jcbPingDBC.jar file was generated by Object Builder and put in the directory:

   <drive>:\CBConnector\04-Persistent-Ping\Server\Working\Nt\PRODUCTION\Jcb

   You can also adapt the runit.cmd file by changing the hostname "atlantic.almaden.ibm.com" by your fully qualified hostname and run it! Please also follow the instructions for Java clients mentioned in the 4.1, "Get ready" on page 30.
3. Change to the directory:
   <drive>:\CBConnector\04-Persistent-Ping\Client\Java\Class
4. Enter following command to run the Ping test client application:

```
java PingDBJ <PingId> <number of Pings> <hostname.domain.company.com>
<port>

For example:

java PingDBJ 2 100 cbcsrva.itsc.austin.ibm.com 900
```

5. The application should show a log as follows:

```
*** Start of Java persistent Ping Client Program for CBC v. 3.0***

1) About to call ORB.init passing Bootstrap information as a property list:
          ORBInitialHost = The host name of the initial CBConnector server
          ORBInitialPort = The bootstrapPing port number to connect to
2) Now attempt to resolve to root naming context using IExtendedNaming.
          orb.resolve_initial_references("NameService")
          NamingContextHelper.narrow(obj)
3) About to resolve the factory finder.

iExtendedNC.resolve_with_string("host/resources/factory-finders/PingDBScope")
          FactoryFinderHelper.narrow(obj)
          FactoryFinder resolved
4) Now attempt to get a Ping Home Object.
          factoryFinder.find_factory_from_string("Ping.object interface")
          IHomeHelper.narrow(obj)
5) Now attempt to create a Ping Key Object.
          About to create a PingKey
          Primary key created successfully
          PingKey created
6) About to find Ping Object with PingHome.findByPrimaryKeyString
          Ping Object found with PingHome.findByPrimaryKeyString
7) Now executing the Ping method 100 times
8) Now calculate the average response time and set the Ping attributes
          number of Pings: 100 average response time: 66 ms

Program ran SUCCESSFULLY!
************* End of Sample Program ***********
```

Once the application has finished running, a new Ping Object was instantiated with values for its attribute's Ping ID, counter, and avgResponseTime. In this case, the Ping Object is a persistent one, and it stores its essential state of data to a datastore.

Check the contents of the datastore with the commands described in 4.3.3, “Check the datastore” on page 52.

If you encounter any error messages, refer to 4.2.2.4, “Troubleshooting” on page 40 and 4.3.2.4, “Troubleshooting” on page 52.

4.3.2.3 ActiveX client

The ActiveX client has, in contrast to the other client test programs, a graphical user interface. But the application logic is the same.

Run the ActiveX application

To run the ActiveX client application, follow these steps:

1. Move to the CD-ROM directory: *<drive>:\CBConnector\04-Persistent-Ping\Client\ActiveX\Vb*.
2. Run register.bat command file in order to register the Ping DLLs to the Windows registry. You can check that the DLLs have been correctly registered by running the OLE/COM Object Viewer tool provided with VisualC++ from Microsoft. By expanding Automation Objects tree node, you should find:
   - CPing
   - CPingCopy
   - CpingKey
3. Start the PingDBX.exe application.
4. After a while a GUI window with entry fields pops up. Fill in the numbers and press the **StartPing**-button.
5. The application generates the cbc_vb_out.log file, which should show a log as follows:

```
Starting Ping Sample
About to initialize the ORB
  Initialization successful
Attempt to resolve root naming context
  narrow the NameService-Object
  Got the root naming context
About to resolve the FactoryFinder
  FactoryFinder resolved
Attempt to get a Home Object.
  Home found
Attempt to find object by Primary Key
  About to narrow to PingMO
  narrowed
  Object found
Pinging...
Average Time for one Ping: 198 msec
```

Once the application has finished running, a new Ping object is instantiated with values for its attribute's Ping ID, counter and avgResponseTime. In this case, the Ping Object is a persistent one, and it stores its essential state of data to a datastore.

Check the contents of the datastore with the commands described in 4.3.3, “Check the datastore” on page 52.

If you encounter any error messages, refer to 4.2.2.4, “Troubleshooting” on page 40 and 4.3.2.4, “Troubleshooting” on page 52.

4.3.2.4 Troubleshooting

In this troubleshooting section, only the problem situations specific to the persistent test applications are described. If you have problems running one of the client applications, refer also to the 4.2.2.4, “Troubleshooting” on page 40.

- Step 6 of the client program fails. The client is not able to find or create a Ping Object.

  ```
  ERROR: Something went wrong during find
  ```

  Check whether the bind of the bind file, *PingPO.bnd*, was successfully executed.

- If you get the error message:

  ```
  CORBA::persist_STORE minor code 0
  ```

  Check the user ID and password setting in the XA Manager `open string` attribute.

4.3.3 Check the datastore

To check whether the data have been written to the database table, open a DB2 command window and execute the following steps:

1. DB2 connect to PingDB.
2. Select * from Ping
   This should show you the contents of the Ping table. There should be a row containing the essential data of your Ping Object.
3. DB2 connect reset.

4.4 What was the purpose?

The purpose of this exercise was to give you a chance to test your CBConnector server with two different Ping server and client applications. You executed client applications either locally or from a CBConnector client.

During this time, you became familiar with the CBConnector System Manager and the application installation process in the CBConnector distributed, multi-tier environment. During the tests, you tried out the CBConnector Runtime architecture with the DB2 Application Adaptor.

We suppose that all the steps were completed successfully and you have a functional multi-tier system. Now, we move on to the next chapters, where you can study the development process steps using the CBToolkit. After that, you will be ready for your first development steps in the CBConnector environment in Chapter 7, “Working with systems management” on page 89 through Chapter 14, “Transactional Object sample” on page 217.

Part 2. Set

This part of our cookbook focuses on the development process and working with the development tools, especially with the Object Builder tool. We also describe the major steps for installation and configuration of the CBConnector Server applications using the System Management GUI.

Chapter 5. Developing for CBConnector

Having followed us up to this point, you should by now:

- Have a broad understanding of the architecture of CBConnector and the role it plays in establishing a solid platform for distributed object computing
- Have installed, configured, and tested your CBConnector development system

Since this book is all about developing applications to run on CBConnector, it is now time to introduce you to the process of building software for this environment.

In Part I of the *IBM Component Broker Connector Overview*, SG24-2022, you can find a high-level overview of the development process and the tools provided with the CBConnector package.

Recapping what is said in this *Overview* redbook, let's mention some essential characteristics of developing for CBConnector-based environments.

To start with, the software you build is meant to be deployed over a multi-tier infrastructure. Moreover, the environment will usually be cross-language and include more than one operating system. Most likely, it will also include a variety of database or transaction processing systems.

On one end of the picture, you have the thin clients, as typified by the devices attached to the ubiquitous Internet. In such an environment software must be dynamically downloadable from servers to any hardware and software infrastructure that might be present on the client. Also, given the "chaotic" state of the Net, you simply cannot assume a particular operating system or language environment in a given situation. Similar topologies are, of course, being established by businesses within their intranets with the primary goal of simplifying administration and reducing total cost of ownership. Java, with its "write-once, run-anywhere" characteristics, is naturally an ideal development platform for such an environment.

In general, the parts of client applications that deal with aspects such as a graphical user interface or local business logic are essentially unaffected by the presence of a CBConnector-based application server.

However, in a CBConnector environment, clients communicate with their application servers using the Internet Inter-ORB Protocol (IIOP), as defined by OMG's Common Object Request Broker Architecture. They do so with the help of *Proxies*, which hide the lower-level details of communication protocols. Proxies are "stand-in" objects for their real counterparts on the other end of the wire. They provide client programmers with the interfaces to call for application functionality.

The essence of developing software for the CBConnector environment lies in building reusable, object-oriented components that execute on middle-tier CBConnector application servers within the CBConnector runtime infrastructure.

These servers come with a full set of CORBA services. You are free to use them at your discretion. Sometimes you definitely have to use them, such as when you want to locate a well-known object (such as a Factory or an Event Channel) through the Naming Service.

However, you normally write your server-based software using the higher-level frameworks, as described in Chapter 2, “Component Broker introduction” on page 7, and in the *Overview* redbook. CBConnector induces you to structure your components such that business logic is well separated from both the user interface and the back-end data storage technology. This way, it supports you in establishing development roles (such as business domain specialists, application assemblers, database specialists, and so forth), separating the often diverging concerns in these areas.

Figure 3 introduces the process of developing in a CBConnector environment.

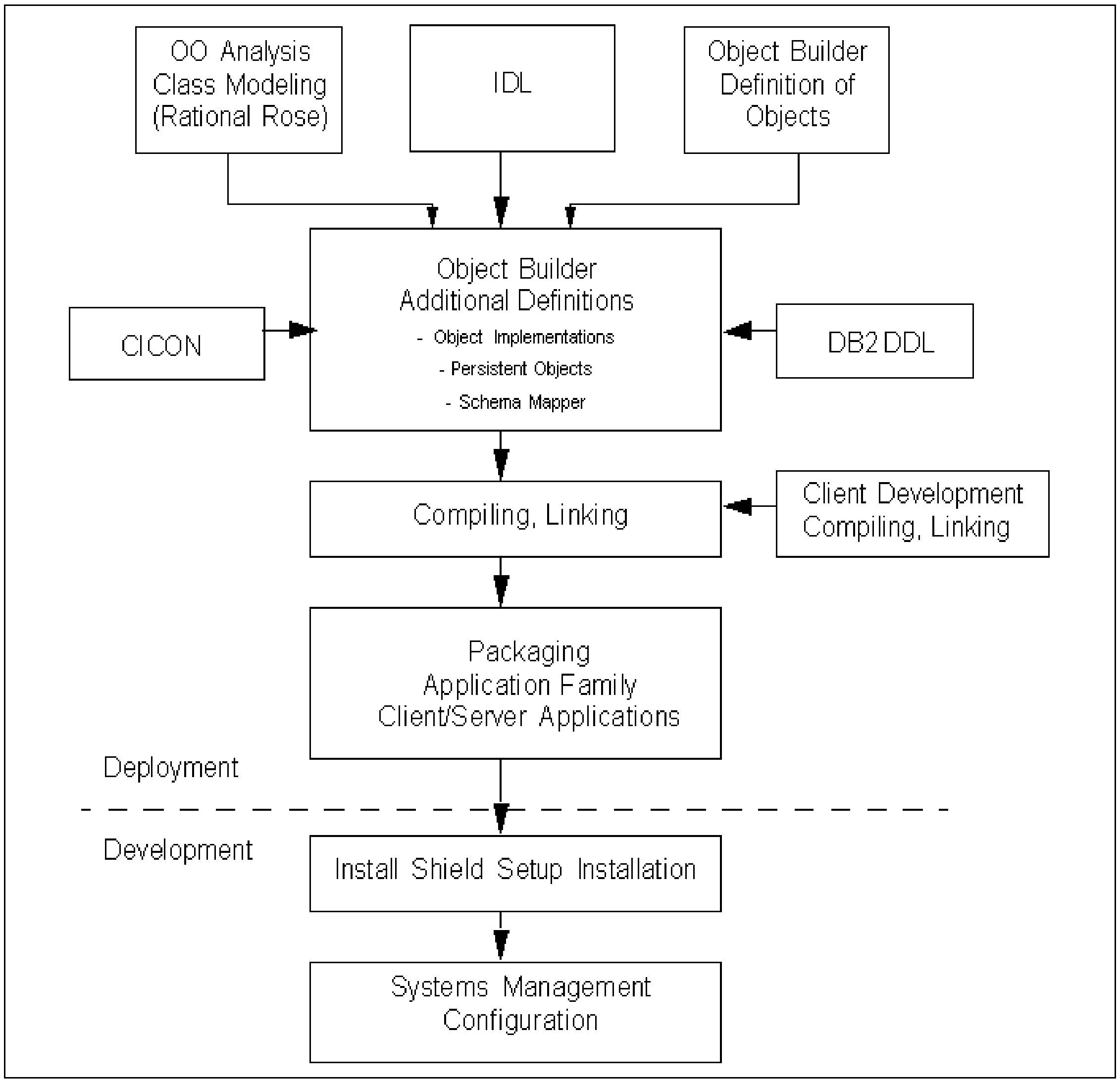

OO Analysis Class Modeling (Rational Rose)
IDL
Object Builder Definition of Objects
Object Builder Additional Definitions
- Object Implementations
- Persistent Objects
- Schema Mapper
CICON
DB2 DDL
Compiling, Linking
Client Development Compiling, Linking
Packaging Application Family Client/Server Applications
Deployment
Development
Install Shield Setup Installation
Systems Management Configuration

Figure 3. Development phases overview

The development process can be summarized in these steps:

1. Import a Business Model into Object Builder.
2. Define Business Objects and map them to persistent stores using Object Builder.
3. Compile and link the Business Objects to produce client and server applications using Object Builder.

4. Package the applications using Object Builder.
5. Install the application on the server using InstallShield.
6. Configure the application on a server using the Systems Management User Interface.

These steps are described in more detail in the following paragraphs.

1. Import a Business Model into Object Builder.

   The Business Object model is developed using Object Builder. You can build your objects from scratch in Object Builder, but most probably the Business Model Objects will be imported via a CASE modeling tool bridge, IDL files, or XML files. You will find some introductory information about the bridge to the Rational Rose modeling tool in Appendix A, “Rational Rose” on page 403.

2. Define Business Objects and map them to adapters.

   The Object Model maps the state of Business Objects to persistence stores by means of adapters. The two adapters developed for CBConnector version 1.2 is for DB2 and CICS. The DB2 DB2 adapter uses a DB2 data definition language file as input to Object builder in order to make the mapping between the Business Object and the database tables. The CICS adapter is a Java bean developed with the CICON supplement to VisualAge for Java. For additional information about using the CICS adapter see the Component Broker documentation: *WebSphere Application Server Enterprise Edition Component Broker*, — *CICS and IMS Application Adaptor Quick Beginnings Version 3.0*, SC09-4439.

3. Compile and link the Business Objects.

   When the definition of the Business Objects is done, they need to be compiled into an executable form. One part will reside on the client, acting as the interface to the Business Object proxies, and the other part is the actual server application.

4. Package the applications using Object Builder.

   The server application needs to be packaged in order to be successfully installed on the server. The resulting setup program handles the registration of the program with the naming services, and the correct placement of the files. Object Builder also provides functionality to package the client application. This is the last operation performed using the Object Builder.

5. Install the application on a server.

   The server application files are installed on the server by running the setup program created in the packaging process.

6. Configure the application on a server.

   The final step is to configure the application onto a server using the Systems Management User Interface. This will activate a server that you can access from the clients you have developed.

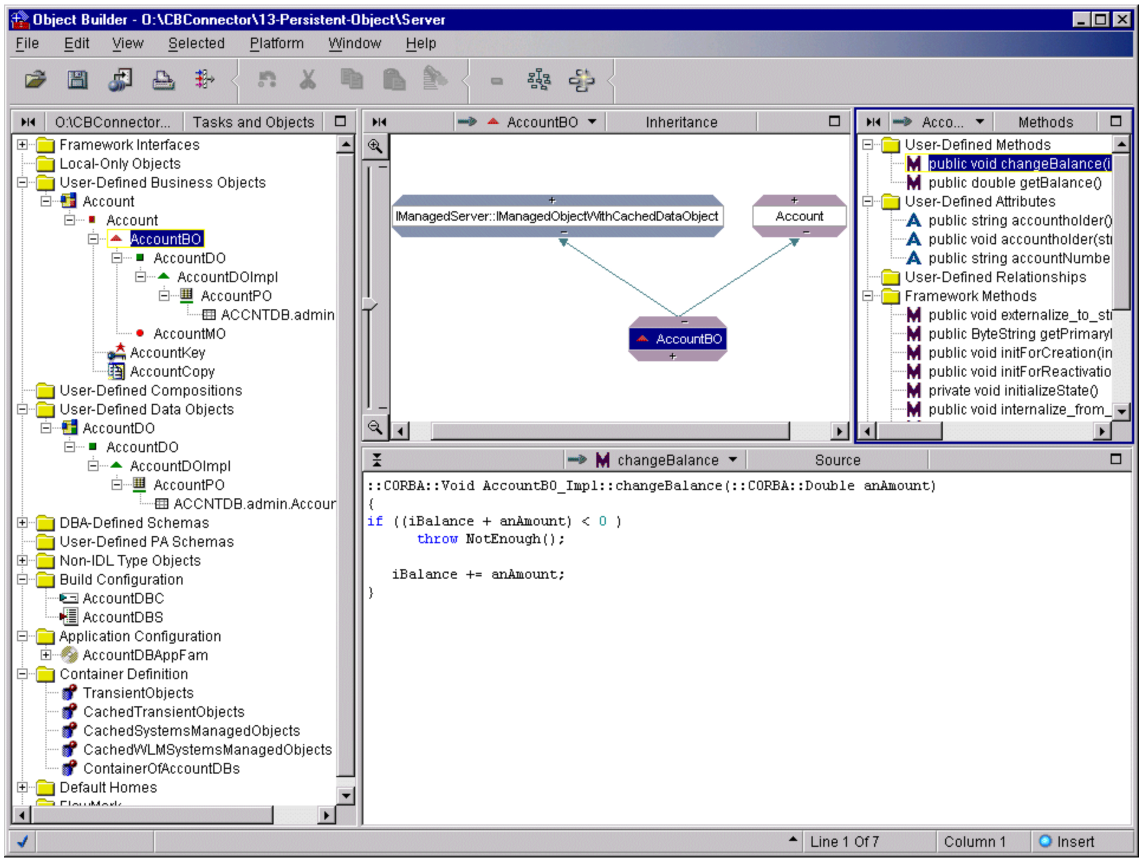
The following chapters will take you through the CBConnector development process by first introducing the Object Builder; and finally, explaining how an application is installed and configured using Systems Management.

Chapter 6. Working with the Object Builder

Object Builder is central to the CBConnector development process. Since it "understands" the CBConnector framework architecture, it can generate most of the scaffolding needed to make the application code you produce work within the CBConnector frameworks.

6.1 Introducing the Object Builder

No matter how you accomplish the first three steps we described in the preceding chapter, you will continue with step 4 to define the objects that are specific for the development with CBConnector. Figure 4 shows the Object Builder graphical user interface with its four interrelated panes.



```
::CORBA::Void AccountBO_Impl::changeBalance(::CORBA::Double anAmount)
{
if ((iBalance + anAmount) < 0 )
      throw NotEnough();

   iBalance += anAmount;
}
```

Figure 4. Object Builder graphical user interface

As you can see, we have selected an *AccountBO* Business Object implementation node in the left pane. The hierarchy screens show that *AccountBO* inherits its interface from **Account** (which is what you work with on the client. Remember the Proxies?).

AccountBO uses an *AccountDO* Data Object to implement its essential persistent state, using either the delegating or caching patterns. Your choice may be influenced by your application characteristics. For example, you might choose the caching pattern if you activate the Business Object once and then use it for a prolonged period, continually accessing its state. This way, state variables are synchronized once at activation, and frequent delegation calls are avoided. In the implementation of the Data Object, you select aspects such as whether or not to use the CBConnector caching service. This has nothing to do with the patterns employed between Business Objects and Data Objects. It relates to the way CBConnector accesses back-end data stores. You **must** use the caching service (as well as the delegating design pattern between Business Objects and Data Objects) when implementing an optimistic locking scheme, for example.

While you do have to add an *AccountMO* Managed Object to the *AccountBO* (essentially so as to specify its quality of service at runtime), Object Builder takes care of producing the necessary boilerplate code for you.

The rest of the nodes in the pane on the left are concerned with the following:

- Working with back-end database schemas (**DBA-Defined Schemas**)
- Defining the contents of your client-side and server-side DLLs (**Build Configuration**)
- You also generate the makefiles and build the application from here.
- **Application Configuration** is the place where you specify your application families as well as their constituent parts for clients and servers
- **Containers** and **Homes** show you what's available in your environment. For example, you can add a new Container, specifying its transaction policies, caching behavior, and storage-management characteristics.

The middle pane on the top displays either the interface or implementation inheritance for the object selected in the left pane. As with the other panes, you can maximize the pane so it takes up the complete Object Builder GUI client area. In the bottom pane, you specify the implementation (write your code) of your methods. The format of the pane should look familiar to you, especially if you have been using the VisualAge for C++ program editor.

Using Object Builder's graphical user interface and ample set of wizards, you are guided in a sequential way all the way from defining new IDL files to building the install images for your servers and clients.

Object Builder also helps with the re-use of your existing application code and database resources. For example, it can build the Persistent Objects for you from existing DB2 SQL data definition (DDL) files and link to existing object models to use as a base upon which to build larger, more complex scenarios (a good feature for teamwork, for instance). Because of these capabilities, Object Builder is the focal point for bottom-up or meet-in-the-middle development, thus supporting one of the major goals of CBConnector — that of operational reuse of existing code and database resources. You will find a sample for meet-in-the-middle development in Chapter 20, “Meet-in-the-middle paradigm with two datastores” on page 323.

In the following sections, we take you on a scenic tour of Object Builder. The procedure shown includes the whole process, from defining an object's interface right up to building the install image ready for deploying on your servers and clients. Our direction will be "top-down" and generic. This means that we can keep the material that follows in the later chapters more concise, allowing those chapters to focus only on the specific aspects of the samples presented.

The overview diagram shown in Figure 3 on page 59 shows the process you have to go through in order to get ready to produce components to be installed on your server. However, you only need one of the first three phases. In this cookbook, we start the process with phase 2 and define all basic objects with the Object Builder.

If you use an OO-modeling-tool to define the components, you may continue with phase 4 to define the objects that are specific for the development with Component Broker.

Another usual starting point is an existing IDL file that contains the interface definition. Thus, you can import the file and do not need to define it with the Object Builder.

Conventions:
In this chapter, a special syntax is used to provide a shorter explanation of the processes for the user input to the Object Builder.

For example, **Options -> Set Working Directory** means select **Options** from the main menu and click on **Set Working Directory** in the pull-down menu that appears.

The same syntax is used for the Tasks and Objects folders. For example, **User-Defined Business Object -> Add File** means right-click on the **User-Defined Business Objects** folder and select **Add File** from the pop-up menu.

6.2 Setting up Object Builder

Object Builder uses a model directory (*Model*) and a working directory (*Working*). The internal representation of the objects you define and configure in Object Builder are stored in the model directory as a series of model files. The generated code for these objects are stored in the working directory.

When you start the Object Builder, it will ask you for the project directory. Specify a directory for the project and move on. The next screen in the SmartGuide lets you specify which existing models you want to include (depend on). By linking to other models, you can easily use objects already defined in those other models. If you link to other models, their objects will be included in your current project as read-only. This is shown in the Object Builder object tree as figures with a thick frame around them. Also, when selected, the text "Read only" appears at the bottom of the Object Builder window. Leave the model linking blank for now and finish. Object Builder creates a new project directory (if it doesn't exist) and makes a *Model* directory underneath it for the object model. Once you start generating code, a *Working* directory will also be created.

You might need to provide Object Builder with the location of the InstallShield development kit so it can call the program to build the installation scripts.

To set the location of the InstallShield root directory, click on **File -> Preferences...** and select **Tasks and Objects** from the Preferences list. Change the InstallShield path if needed, and select an implementation language. The implementation language you select here will be the default language for all objects. You can override this selection for each of your Business Objects separately in case you want them to be generated in different languages (shown later).

6.3 Importing and exporting Object Builder models

This is how to import and export Object Builder models:

- To create a new IDL file, select **User-Defined Business Objects -> Add File...**. Specify the name for the IDL file. The SmartGuide also allows you to add constructs (`constants`, `enums`, `exceptions`, `typedefs`, `structs`, or `unions`) and additional include files (such as those containing other objects you use in the definition of the interface).
- To import a model from an existing IDL file, select **User-Defined Business Objects -> Import IDL...**. and follow the instructions of the SmartGuide to add your file to the IDL file list.
- Your model is always saved in the IDL format. You can, however, also export a model to an XML file format. Select **User-Defined Business Objects -> Export...**. Object Builder exports the Business Object model (*udbo.xml*) into an *Export* directory in your *Working* directory. You can export other model definitions similarly from the main folders in the "Tasks and Objects" pane. To export the whole project, select **File -> Export Model**.
- To import a model from an XML file, select **User-Defined Business Objects -> Import...**. Specify the file to import (*udbo.xml*).

6.4 Create Business Objects with Object Builder

In this section, you will learn how to add Business Objects to your object model.

6.4.1 Add a Business Object file

The first step will be to define a file name that will keep the interface definitions for your Business Object.

6.4.1.1 User-Defined Business Objects -> Add File

1. Name page.
   Type the name of the IDL file that should contain the interface definition for your Business Object, for instance **Account**. Then go to the next page.
2. Constructs page.
   Here, you have nothing to do and can proceed to the next page.
3. Files to Include page.
   Here, additional include files are specified. Only **IManagedClient** is needed; so you do not have to make a change.

4. Comments page.
   Anything you type in this edit box is copied to the IDL file and helps readers of the file to understand its purpose.
5. Finish.
   Now the IDL file is defined in Object Builder, but it is still an empty shell that must be filled with definitions of attributes and methods.

6.4.2 Add a Business Object interface

Now you can define the interface for your Business Object.

6.4.2.1 Business Object File (Account) -> Add Interface

1. Name page.
   Type the name of the interface for instance **Account** in the Name field. Leave the interface is queryable unchecked. (Select this option if you want to use SQL calls from the Query Service.)
2. Constructs page.
   Here, you can add `exceptions`, `constants`, `structs`, and so forth by right-clicking on the **Constructs** folder.
3. Interface Inheritance page.
   Again, the defaults are acceptable. If **Account** were a subclass of some other class you had defined, you would change the inheritance to reflect that. However, because **Account** is a base class in your design, it only needs to inherit interfaces from the CBConnector Managed Object Frameworks. The appropriate interface is already selected by default.
4. Attributes page.
   Here, you specify the type and name of the attribute along with the initial value and implementation properties. Right-click on the **Attributes** tab and select **Add** Type in an attribute name (AccountName) and select a type.
5. Methods page.
   Add methods with parameters and exceptions.
6. Object Relationships page.
   The objects defined in this book do not have relationships with other objects; thus you do not need to enter anything on this page.
7. Comments page.
   As with the file object you added earlier, any comments you type here are written to the generated IDL file.
8. Finish.
   Now the interface is created in the User-Defined Business Objects Folder.

Click on the interface to see the methods and attributes in the Method List pane.

6.5 Additional Definitions in Object Builder

The following steps guide you through the process of adding a Key.

6.5.1 Adding a Key

Define the Key Object prior to adding the implementation to the Business Object. You need to provide a key in order to uniquely identify the instances of the Business Object.

6.5.1.1 Business Object Interface (Account) -> Add Key

1. The Key Object is given a default name based on the name of the interface and a default file name based on the Business Object file name. You should not change these names unless you have a reason for it.
2. Select the attributes that uniquely identify an instance and move them to the Key Attributes list.
3. Implementation Inheritance page.
   For samples in this book, use **IManagedLocal IManagedLocal::iprimarykey** as the parent class.
4. Summary of Framework Methods page.
   This page shows a list of the framework methods that will automatically be implemented for you. Select a method to get a short description of its purpose.
5. Optional Framework Methods page.
   On this page you can select optional methods to implement. Right now nothing needs to be selected.
6. Finish.
   You can now click on a method in the Method List pane and take a look at the implementation which Object Builder has created.

6.5.2 Adding a Copy Helper

Since a Copy Helper is an optional class, we do not need to implement one and it is not difficult to create. The Copy Helper is useful for components with many attributes that have to be initialized by the client.

6.5.2.1 Business Object Interface (Account) -> Add Copy Helper

1. Name and Attributes page.
   The key object is given a default name based on the name of the interface and a default file name based on the Business Object file name.
   Select the attributes to be set by the Copy Helper.
2. Implementation Inheritance page.
   Do not change the defaults unless you have a good reason for doing this (for instance, you have created a Copy Helper class and want to re-use this definition).
3. Summary of Framework Methods page.
   This page shows a list of framework methods for the Copy Helper object that will automatically be implemented for you. Select a method to get a short description of its purpose.
4. Finish.
   You can now click on a method in the Method List pane and take a look at the implementation Object Builder has created.

6.5.3 Adding a Business Object Implementation

The next step is to create the server object. All the other objects we have defined so far are client objects (Business Object Interface, Key, and Copy Helper). The SmartGuide also creates a Data Object Interface that is used to access the data of the Business Object.

6.5.3.1 Business Object Interface (Account)->Add Implementation

1. Name and Data Access Pattern page.
   A default file name and name for the Business Object is provided.
   For the sake of convention, you should not change these names.
   The Pattern for Handling State Data in our samples is caching with lazy evaluation. Thus, the essential state data of the Business Object is cached, and the attributes are copied to the cache when required, not at startup.
   By default, the **Create a new Data Object Interface** is selected. If you want to re-use a Data Object Interface, check the other option to select an existing one later.
2. Implementation Inheritance page.
   There is nothing to do here.
3. Implementation Language page.
   On this page, you select the Business Object. The selection you make here overrides the default selection done for the project. Thus, if you have many Business Objects, you can easily generate and implement them in

different languages. Some files will be generated in both Java and C++ regardless of the language specified. The client language is totally independent on the Business Object language; so you can mix any type of Business Object language with any type of client language.

4. Key Selection page.
   Your key should appear in the combobox.
5. Handle Selection page.
   No handle is needed.
6. Data Object Interface page.
   Select the Business Object attributes that make up the state data. Move all the attributes in the Business Object Attributes list to the State Data list.
7. Attributes to Override page.
   Do not select anything. We'll leave the inherited attributes as they are.
8. Summary of Framework Methods page.
   Note the framework methods, **syncToDataObject** and **syncFromDataObject**, that have been included in order to keep the cached copy of attributes synchronized with the Data Object attributes.
9. Finish.
   Nodes for the Business Object and the Data Object are created.

6.5.4 Add your code

Click on the node for the Business Object Implementation in the Tasks and Objects pane at the bottom. By selecting a method, you should see it in the Method List pane. Click on one after the other and type your C++ Code in the Method Implementation pane. Alternatively, you can also save the implementation to an external file and use the properties SmartGuide of the method to set a link to the file.

For the first samples, this is the only code you need to type to create a fully functional Business Object! Isn't that great?

6.5.5 Adding a Data Object Implementation

As with the Business Object Implementation, you can define more than one Data Object Implementation. Each of our samples are created from scratch (except the IDL file). However, you can also use one implementation for transient or one for embedded SQL.

Regarding the samples in this book, you have two possibilities for Data Objects: transient and embedded SQL.

6.5.5.1 Data Object Interface (AccountDO) -> Data Object Implementation

1. Data Object Implementation page.
   Under Environment, select **BOIM with UUID Key** for transient samples and **BOIM with any key** for persistent samples.
   Under Form of Persistent Behavior and Implementation, select **transient** or **embedded SQL**.
   Set the Data Access Pattern for the persistent sample as delegating. (Local copy would cache the essential state).
   Under Handle for Storing Pointers, click **Home name and key**.
2. Implementation Inheritance page.
   For the transient, the default selection for the parent class is **IBOIMExtLocalToServer IBOIMExtLocalToServer::idataobjectbase**.
   For the Persistent Object, however, the default selection is **IRDBIMExtLocalToServer IRDBIMExtLocalToServer::idataobject**
3. Key and Copy Helper page.
   The previously defined objects are already selected. (If you had defined more than one Key and Copy Helper for the component, you could make an alternate selection here.)
4. Following pages.
   The following pages handle the Persistent Objects and the mapping of the attributes to the Persistent Object. We use the default setting for our samples; so you can skip these pages.
5. Finish.

6.5.6 Adding a Managed Object

Now you are ready to define the Managed Object. It represents the component to the client application and handles all calls from the client to the component on the server.

6.5.6.1 Business Object Implementation (AccountBO) -> Add Managed Object

1. Name and Application Adaptor page.
   As with earlier objects, a default name and file name are provided (*AccountMO*,*AccountMO*).
2. Implementation Inheritance page.
   For the samples in this book, no parents are necessary.
3. Finish.

6.5.7 Generating the code

This is a step where you can relax, because Object Builder will create all the IDL, header, and code files you need. Everything that is generated or compiled is placed in the *Working* directory.

If you have not saved your work, this is a good time to do so. Select **File -> Save**.

6.5.7.1 Business Object File (Account) -> Generate -> All

What is created?

Node	Files
AccountDOImpl(Data Object Implementation)	AccountDOImple.idl, AccountDOImpl.ih, AccountDOImpl_I.cpp
AccountDO (Data Object Interface)	AccountDO.idl
AccountMO (Managed Object)	AccountMO.idl, AccountMO.ih, AccountMO_I.cpp
AccountBO (Business Object Implementation)	AccountBO.idl, AccountBO.ih, AccountBO_I.cpp
AccountKey (Key)	AccountKey.idl, AccountKey.ih, AccountKey_I.cpp
AccountCopy (Copy Helper)	AccountCopy.idl, AccountCopy.ih, AccountCopy_I.cpp
Account (Business Object Interface)	Account.idl

A few files are still missing: the makefiles to generate the target DLLs.

6.5.8 Creating client and server DLLs

Now create the two DLLs for accessing the objects from the client and the server. One runs on the client and provides access to the Business Object Interface, Key, and Copy Helper, and one runs on the server and provides access to the Business Object, Data Object and Managed Object.

6.5.8.1 Defining a client DLL

You must define the client DLL first so that when you define the server DLL, it can link with the client DLL's library file. Select **Build Configuration -> Add Client DLL**.

1. Type a name for the client DLL in the Name field, for instance **AccountC**.The other fields (description and options for the makefile) are optional. You can leave the fields blank.
2. Client Source Files page.
   The **Items available** listbox shows the client objects you have defined (*Account*, *AccountCopy* and *Account Key*). Move all to the **Items chosen** listbox.
3. Libraries to Link With page.
   For the client DLL, there are no Items available. Thus, there is nothing to do on this page.
4. Finish.

6.5.8.2 Defining a server DLL

Select **Build Configuration -> Add Server DLL**.

1. Type a name for the client DLL in the Name field, for instance **AccountS**.As with the client DLL, you do not need to specify any other options on this page.
2. Server Source Files page.
   The **Items available** listbox shows the server objects you have defined (*AccountBO*, *AccountDO*, *AccountDOImpl*, and *AccountMO*). Move all to the **Items chosen** listbox.
3. Libraries to Link With page.
   You need to link with the AccountC client DLL.
4. Finish.

6.6 Generating and building the DLLs

This action generates makefiles for all the DLLs defined in the folder (in this case, for the AccountC and AccountS DLLs) and also generates an *all.mak* file that calls the DLL makefiles.

Select **Build Configuration -> Generate -> All -> All Targets**. Note that you can also generate only specific files.

6.6.0.1 Build Configuration -> Build -> All Targets

This builds the *all.mak* file and displays the progress of the build in a Command Window. Be sure that the build finished without errors. Note that in order to save time after the first build, you can also select to build only **Out-of-Date Targets** for a specific language.

6.7 Writing a client application

On the following pages, you can read about the client programming models.

6.7.1 C++ client programming model

Writing a client program to access CBConnector Managed Objects is described extensively in the *Component Broker for Windows NT Programmer"s Guide and Application Development Tools*, BOOKNUMBER. The client-specific development process builds on the following artifacts:

- Client runtime DLLs and library files (for example, AccountTRC.dll and *AccountTRC.lib* for the transient account scenario)
- Client C++ usage bindings.

These are the header files, like *Account.hh*, used for accessing account server operations. The bindings also include the header files for any local objects, such as key helpers and Copy Helpers (*AccountKey.hh* and *AccountCopy.hh* in our case) through an **Account** proxy.

Files such as *Account.hh* also include the declarations and definitions of a number of classes that are part of the C++ bindings, as defined by OMG and implemented in CBConnector. These are of no real concern to you as a client programmer. However, they tend to convolute the header files somewhat. If you want to get a quick understanding of the interfaces you are going to use, take a look at the corresponding IDL files (for example *Account.idl*) that should also be available to you.

All of the above are built using the Object Builder, which puts them into the *Working* directory.

Essentially, including the above constructs in your application should be business as usual to you. You make the header files accessible in your makefiles, such that the compiler can include the various declarations and definitions. The linker will need access to the *.lib* file, so that it can resolve its external references. And, of course, before you can run your program, don't forget to place the DLL provided through the server development process into the correct directory.

Having gone into all this detail, Object Builder helps you get your client application running in the following ways:

- When defining a client application within an application family, include both the client runtime DLL and your executable (for example, AccountTRAppC.exe) in that application.
- Make the client application part of an install image.
- Install the image using the InstallShield.
- Make sure the client runtime DLL (for example AccountTRC.dll) is in a directory accessible when executing your program.

6.7.2 Java client programming model

This section deals with the specifics of using a Java client to access Managed Objects on a Component Broker Connector server.

6.7.3 Java proxies

A Java application needs stubs, Key, and Copy local objects to create, find and use remote objects. Object Builder generates them for you and puts all the required files in a file named: jcb<client_dll_name>.jar. For example, if you named your client DLL AccountTRC, then the jar file will be jcbAccountTRC.jar.

This file is located in the directory:
<drive>:<your_directory>\Working\Nt\PRODUCTION\Jcb

At last, our application uses one additional Java file named *AccountApp.java* which is your own written client code. It encapsulates most of the logic interfacing the proxy objects.

When all the *Java* files above have been made, you can compile the client with javac.

6.7.4 ActiveX client programming model

We have tried to make the development of new client programs as easy as possible. For Visual Basic, most steps are automated, and you can concentrate on programming the specifics.

With an ActiveX component, we can develop our client side to access the business object interfaces using Visual Basic (in our example), C++, or Java languages. We describe all the steps to create the COM components using as example the Ping, PingKey and PingCopy interfaces.

The following are all the steps needed to wrap all the business object interfaces in COM components accessed using Visual Basic:

- Create or reuse the IDL files
- Create the client-side bindings
- Generate the GUIDs
- Generate the COM wrappers
- Compile and Link our classes
- Register the OLEs
- Access the OLE with Visual Basic

Create or reuse the IDL files

Our business object interfaces are defined using the IDL language. We can create these IDL files manually, or we can use the files generated by the Object Builder during the developing process.

Create the client-side bindings

The IBM idlc emitter generates all the client-side bindings needed to use the Business Object interfaces by the client. Process Ping.idl, PingKey, and finally PingCopy using the following command:

```
idlc -mcpponly -suc:hh Ping.idl
idlc -mcpponly -suc:hh PingKey.idl
idlc -mcpponly -suc:hh PingCopy.idl
```

The flag cpponly tells the emitter to generate only the standard CORBA C++ ORB binding. This process generates all *_C.cpp and *.hh files for our interfaces.

If our object is a local object only (that is, PingCopy or PingKey), we also need to the implementation headers. You can either use those generated by Object Builder or generate them using the following command:

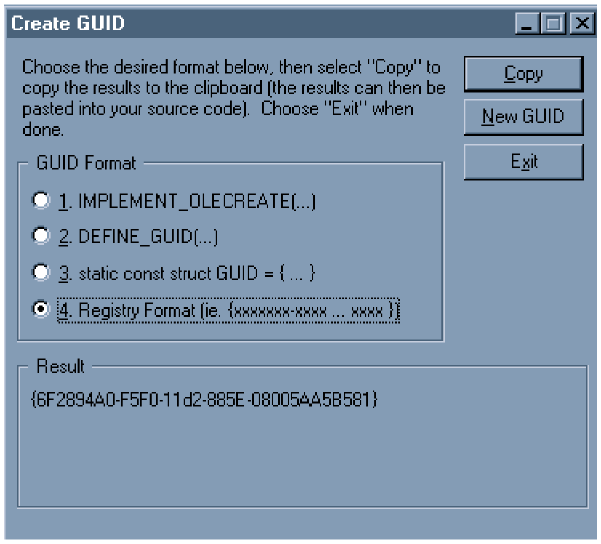
```
idlc -mcpponly -sih:ic PingKey.idl
idlc -mcpponly -sih:ic PingCopy.idl
```

As a result, you get *_I.cpp and *.ih files. If we check out the content of the local only implementation headers, we see that there is no implementation for these files.

We could develop the implementation, but it is easier to reuse the files generated by Object Builder. Thus, copy the files *_I.cpp and *.ih for the Key and Copy interface from the Object Builder working\nt directory to our ActiveX developing directory.

Generate the GUIDs

The Microsoft Visual C++ guidgen.exe tool allows us to generate the Global Unique ID (GUID) for the COM components. Generate the GUID for each interface, choosing from the GUID panel the Registry Format option as shown in Figure 5.



Choose the desired format below, then select "Copy" to copy the results to the clipboard (the results can then be pasted into your source code). Choose "Exit" when done.

GUID Format

- ○ 1. IMPLEMENT_OLECREATE(...)
- ○ 2. DEFINE_GUID(...)
- ○ 3. static const struct GUID = { ... }
- ● 4. Registry Format (ie. {xxxxxxxx-xxxx ... xxxx })

Result

```
{6F2894A0-F5F0-11d2-885E-08005AA5B581}
```

Figure 5. GUID panel

Warning

The guidgen.exe tool generates a Global Unique ID (GUID) that we use with the idl2com compiler in order to register our interfaces in the system as Automation controls. If we let guidgen generate the GUID sequentially with the new GUID option without restarting it each time, it generates the GUID by adding 1 to the previous GUID generated. Due to this behavior, if we have defined in our idl file more than one interface or declared some CORBA exceptions, it might happen that two or more interfaces end up with the same GUID IID_DI. With GUID collision, we will not be able to register the controls correctly. Thus, we need to stop/restart the guidgen tool each time to get the right 128-bit GUID with no potential collisions between each of them.

Create the COM wrappers

We need to process our idl files with the idl2com compiler to create all the wrapper and needed classes:

```
idl2com -g 006EE600-F5F1-11d2-885E-08005AA5B581 -i C:\cbroker\include
Ping.idl
idl2com -g 006EE601-F5F1-11d2-885E-08005AA5B581 -i C:\cbroker\include
PingKey.idl
idl2com -g 006EE602-F5F1-11d2-885E-08005AA5B581 -i C:\cbroker\include
PingCopy.idl
```

Use the value produced by guidgen tool with the -g flag, and the include directory where you have installed the CB where all IDL files are located with the -i flag. If we use other user-defined interfaces in our idl file, we have to process them as well through the idl2com compiler in order to have all the required libs and headers available during the compiling and linking time. We have all the C++ classes needed to create our COM component. The idl2com compiler also creates the makefile named <interface>.mak to compile and link your code to create the DLL. In our example, the mak files are named Ping.mak, PingKey.mak, and PingCopy.mak.

Warning

Idl2com is a Java program using the class ComCompile from the somaxic.zip archive located in the \Cbroker\lib directory on your drive. Due to the NT environment variable length problem, it might happen that the JVM is not able to find it, even though somaxic.zip has been correctly included in the classpath variable. In that case, we need to shorten the classpath by removing unused paths in order to it keep under the imposed Windows operating system limit.

Compile and link our classes

Process the makefiles with the utility nmake.

```
nmake -f Ping.mak
nmake -f PingKey.mak
nmake -f PingCopy.mak
```

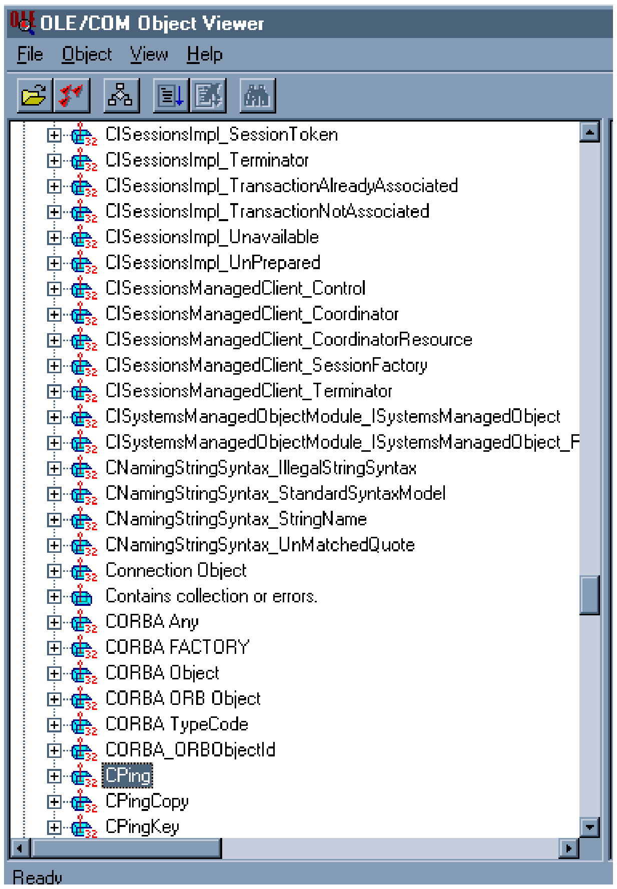
Refer to Appendix C, “Client development with VisualAge for C++” on page 413 for developing with two-compiler environments.

Register the OLEs

If we have followed all the steps described above, we should have the OLE in process server Ping.dll PingKey.dll and PingCopy.dll in our ActiveX developing directory. Now we use the utility regsvr32 provided by Microsoft to register the OLEs:

```
regsvr32 /s Ping.dll
regsvr32 /s PingKey.dll
regsvr32 /s PingCopy.dll
```

We use the utility oleview.exe to check out if the components registration has been done successfully. In the Automation Objects folder, we should have the three components as shown in Figure 6.



```
CISessionsImpl_SessionToken
CISessionsImpl_Terminator
CISessionsImpl_TransactionAlreadyAssociated
CISessionsImpl_TransactionNotAssociated
CISessionsImpl_Unavailable
CISessionsImpl_UnPrepared
CISessionsManagedClient_Control
CISessionsManagedClient_Coordinator
CISessionsManagedClient_CoordinatorResource
CISessionsManagedClient_SessionFactory
CISessionsManagedClient_Terminator
CISystemsManagedObjectModule_ISystemsManagedObject
CISystemsManagedObjectModule_ISystemsManagedObject_F
CNamingStringSyntax_IllegalStringSyntax
CNamingStringSyntax_StandardSyntaxModel
CNamingStringSyntax_StringName
CNamingStringSyntax_UnMatchedQuote
Connection Object
Contains collection or errors.
CORBA Any
CORBA FACTORY
CORBA Object
CORBA ORB Object
CORBA TypeCode
CORBA_ORBObjectId
CPing
CPingCopy
CPingKey
```

Figure 6. OLE Viewer

Access the OLE with Visual Basic

Now we can start developing our client using Visual Basic in order to access the interfaces exposed by the OLE components.

6.8 Packaging an application

The next step on the way to your ready-to-use component is to package the DLLs and create an installation image.

6.8.1 Creating an application family

An application family consists of several applications. In our samples, the Account component forms one application, and the client application that uses Account forms another.

During the installation, you select the applications of the application family that you want to install. The de-installation is also handled by this installation program.

6.8.1.1 Application Configuration -> Add Application Family

1. Name page
   Type, for example, `AccountAppFam` in the Name field.
2. Installation Information page
   Nothing to do.
3. Finish

6.8.2 Defining the client application

Essentially, client applications are built outside the Object Builder environment. An overview of how you do that is given in 6.7, “Writing a client application” on page 75.

6.8.2.1 Application Family (AccountAppFam) -> Add Application

Name and Environment page.
Type, for example, `AccountAppC` in the Name field.
Select **stopped** as the initial state of application because the installation process will start the application if you selected **running**.

Additional Executables page.
For the Account samples, include AccountApp.exe (the client program you created) and AccountC.dll (provided by Object Builder and containing the Account Interface, the Client Object's Key and Copy Helper):

Finish.

6.8.3 Defining a server application

To define the server application that will contain the Account component, follow these steps:

6.8.3.1 Application Family (AccountAppFam)->Add Application

1. Name and Environment page.
   Type, for example, `AccountAppS` in the Name field.
2. Select **stopped** as the initial state of application because the installation process will start the application if you selected **running**.
3. Additional Executables page.
   You need not add any DLLs or executables here. However, we will define a Managed Object for the server application that includes the necessary DLLs.
4. Finish.

6.8.4 Configuring a Managed Object

The Object Builder SmartGuide allows you to add the Managed Object to your object model.

6.8.4.1 Server Application (AccountAppS) -> Add Managed Object

1. Selection page.
   In the Managed Object field, select **AccountMO AccountMO**, and the other fields are filled with the appropriate objects:
   - Managed Object: **AccountMO**, in the AccountS DLL
   - Primary Key: AccountKey, in the AccountC DLL
   - Copy Helper: AccountCopy, in the AccountC DLL
2. Data Object Implementation page.
   Add the Data Object Implementation.
3. Container page.
   The default **CachedSystemsManagedObjects** is OK.
4. Home page.
   For all samples except the specialized home sample, you are done. After checking the **Customized Home**, the specialized home (on this page, called **Customized Home**) and the corresponding DLL is selected from the listbox.
5. Finish.

6.8.5 Creating an install image

The InstallShield development kit is needed to create the install image for your application family. If you do not want to use the InstallShield, you can do the installation process yourself.

6.8.5.1 Application Family (AccountAppFam) -> Generate

The generation makes a subdirectory (named *AccountAppFam* under *Working*) containing files needed to build the installation. The DDL file will set up your application with the System Management.

6.8.5.2 Application Family (AccountAppFam) -> Build

Build puts together all the files into a neat setup.exe package.

If you do not use InstallShield:

Copy the DLLs to the *bin* directory and follow the steps in 7.2.1, “Working with systems management” on page 90, through 7.2.1.6, “Configure the server with the host” on page 92.

6.9 Artifacts of the development process

On the following pages, we summarize the development process in table form.

6.10 Development overview

The tables provide a short overview of the development process. For a more detailed explanation, see Chapter 6, “Working with the Object Builder” on page 63.

6.10.1 Defining and building the objects

On the following pages you will find tables that summarize the information we have provided.

First we have an overview of component development using Object Builder:

Process	Your Input	What is produced
Create a new model	Open Object Builder and specify directory	An empty model
Import IDL to Object Builder **or Import OO model** **or define Interface in Object Builder**	**User-Defined Business Objects -> Import IDL** or open the model exported by the bridge or **User-Defined Business Objects -> Add File and Business Object File (x) -> Add Interface**	Node for Business Object File (x), **face** Node for Business Object Interface (x)
Add a Key	**Business Object Interface (x) -> Add Key** Select attribute(s) for the key	Node for the Key (xKey)
Add Copy Helper (optional)	**Business Object Interface (x) -> Add Copy Helper** Select attributes to be set by the Copy Helper	Node for the Copy Helper (xCopy)
Add Business Object Implementation	**Business Object Interface (x) -> Add Business Object Implementation** Select **Pattern for Handling State Data** Select default **Handle for storing pointers to the Business Object** And select **Attributes for the Data Object Interface**	Node for Business Object Implementation (xBO) Node for Data Object Interface (xDO)
Add a Managed Object	**Business Object (xBO) -> Add Managed Object**	Node for Managed Object (xMO)

Process	Your Input	What is produced
Add the Data Object Implementation	**Data Object Interface (xDO) -> Add Implementation**. Select **Environment** and **Persistent Behavior**	Node for Data Object Implementation (xDOImpl)
Implement the methods of the Interface	Select **Business Object Interface**. Select the method in the method-list pane, edit in Method Implementation pane, or specify an external file (properties SmartGuide of the method)	
Let the Code be generated	**Business Object File (x) -> Generate All**	Source code for generating the Business Object
Add Client DLL	**Build Configuration -> Add Client DLL Enter Name (xC)** and select source files	Node for client DLL (xC)
Add Server Dll	**Build Configuration -> Add Server DLL Enter Name (xS)** Select source files, select client library to link with	Node for server DLL (xS)
Generate Makefiles	**Build Configuration -> Generate All**	all.mak, xC.mak, xS.mak
Build the DLLs	**Build Configuration -> Build All Targets**	Client and server DLL

6.10.2 Create the install image

Now that you have created the DLLs, you can create the install image.

Next we have an overview of packaging applications in Object Builder:

Process	Your Input	What is Produced
Create an Application Family	**Application Configuration -> Add Application Family (xFamily)**	Node for application family (xFamily) Directory for Family in Working directory
Define a client Application	**Application Family (xFamily) -> Add Application (Client_x)**	Node for Application Client_x
Define a server application	**Application Family (xFamily) -> Add Application (Server_x)**	Node for Application (Server_x)
Configure a Managed Object	**Application (Server_x) -> Add Managed Object** Select MO, Data Object Implementation, and if needed a customized home.	Node for MO
Create an Install Image	**Application Family (xFamily) -> Generate**	DDL files for application

6.10.3 Generated Files

Finally, we show the files generated by the process above:

File	Contents
all.mak, xC.mak, xS.mak	Makefiles for Client and Server DLL
x.idl	Business Object Interface for client programs
xCopy.idl, xKey.idl, xBO.idl, xMO.idl, xDO.idl, xDOImpl.idl	Object Interfaces

File	Contents
xCopy.ih, xKey.ih, xBO.ih, xMO.ih, xDOImpl.ih	Implementation Interface, defines x_Impl class
x_I.cpp, xCopy_I.cpp, xKey_I.cpp, xBO_I.cpp, xMO_I.cpp, xDOImpl _I.cpp	Implementation of x.ih
x.hh, xCopy.hh , xKey.hh, xBO.hh, xDO.hh, xDOImpl.hh, xMO.hh	Contains the definition of the Skeleton class
x_C.cpp, xCopy_C.cpp, xKey_C.cpp, xDO_C.cpp, xDOImpl_C.cpp, xBO_C.cpp, xMO_C.cpp	Implementation of .hh - files for client side
xMO_S.cpp, x_S.cpp, xBO_S.cpp, xCopy_S.cpp, xDO_S.cpp, xDOImpl_S.cpp, xKey_S.cpp	Server-side dispatching code
xMO_IR.cpp	Implementation for registration of Managed Object in Interface Repository
dll.c	Additional information for building the DLL
xC.def, xS.def	Defines exported names of the DLL
xC.lib, xS.lib	Library files to link DLLs
xC.dll, xS.dll	Contains the binary implementation for the client and server side
xMO_IR.exe	Registration of interface in Interface Repository
xMO_IR.map, xC.map, xS.map	Compiler-specific files
xC.exp, xS.exp	Compiler-specific files
***.obj**	Object files containing compiled code

You have now read the general information on how to use the Object Builder in the development process. The following chapters describe our scenarios and provide detailed information on the specific sample applications we developed.

Chapter 7. Working with systems management

So far, you have built and packaged your application as explained in Chapter 6, “Working with the Object Builder” on page 63. You are now ready to deploy and test it.

You need to copy the PRODUCTION directory and its subdirectories generated by Object Builder to your deployment host.

Before you can run your application, you need to perform two additional steps:

- Install the application
- Configure the application by using the Systems Management User Interface

The process of configuring your application involves steps we explain in this chapter. You can see the graphical flow in 7.2.4, “Systems management summary” on page 95.

7.1 Install the application

1. Start Component Broker’s System Manager User Interface.
2. Select **View->View level->Control**.
3. Expand **Host Images** and select your host.
4. Right click on your host and from the pop up menu, select **Load Application**.
5. From your PRODUCTION\<yourApplicationName>AppFam directory, select **<yourApplicationName>AppFam.ddl** file.
6. Press **OK** twice.

This creates a directory called:
<drive>:\cbroker\data\ntApps3.0\<yourApplicationName>AppFam

It contains all the necessary files to configure and run your application.

Make sure the name server is running:

- Host Images
  - Your server host
    - Server Images
      - (Your server host) Name Server

Look at the bottom of the window for the current status of the Name Server and ensure that it is running

7.2 Configuring the application with systems management

When the setup program has been installed, you are ready to configure your server using the System Management User Interface.

7.2.1 Working with systems management

You can find a short introduction to systems management in the *IBM Component Broker Connector Overview* redbook. A detailed description of systems management concepts and procedures is given in the online documentation distributed with CBConnector.

We suggest that you get acquainted with systems management before you install the application.

In order to keep this chapter generic for all the Account samples, we write **Account..AppFam** for all the different samples, for instance **AccountTRAppFam**, **AccountDBAppFam**, and so on.

7.2.1.1 Start Systems Management User Interface

Start the Systems Management User Interface. To have access to all the functions you need in order to install an application, the user level needs to be set to **Expert** or **Super User**.

7.2.1.2 Create a Management Zone and Configuration

Management Zones allow you to cluster configurations logically in a network for management purposes. Configurations allow you to have alternative definitions of your server. For instance, you may want to have different daytime and nighttime configurations.

Create a new Management Zone for your application.

- **Management Zones -> Insert**
  Type `Account..MZ`

Now create a new configuration for this Management Zone.

Management Zones

- **Account..MZ -> New Configuration**
  Type `Account Config`

7.2.1.3 Configure the application with the Management Zone

When you installed the application, an entry was added representing the application in the host image, under application family installs. You want to add this application onto your newly created configuration.

The first step is to retrieve, or drag, the application from the host image.

- Host Images
  - Your server host image
    - Application Family Installs
      - Account..AppFam
        - Application Installs
          - **Account..S -> Drag**

Now you have your application in the systems management "clipboard", and you may drop it onto the configuration.

- Management Zones
  - Account MZ
    - Configurations
      - **Account Config -> Add Application**

At this point, you have defined a Management Zone that holds a configuration, and you have defined your application within this configuration. The next step is to create a server that this configured application can reside on.

7.2.1.4 Configure a server

You can define a free-standing server or a server group. A server group enables scalability and load balancing. In this book, we only work with free-standing servers.

In 4.1, “Get ready” on page 30, we installed all the Location Objects to be used with this book. When you define a server, you must give it the same name as defined by the corresponding Location Object in DCE.

Insert a server to your configuration.

- Management Zones
  - Account MZ
    - Configurations
      - Account Config
        - **New -> Server (free-standing)**

  Type `Account..Server` as the name for the server.

7.2.1.5 Associate the configured application with the server

Now, you have a configuration with a configured application and a brand new server. To continue, you need to associate the application with the server.

Pick up your configured application...

- Management Zones
  - Account MZ
    - Configurations
      - Account Config
        - Applications
          - **Your application -> Copy**

...and drop it onto the server:

- Management Zones
  - Account MZ
    - Configurations
      - Account Config
        - Servers (free-standing)
          - **Account..Server -> Configure Application**

A Configured Applications folder now appears under Account..Server. You can expand the folder to display the entry for Account..S.

Some of our samples include more than one application for the server. You need to follow the same process for each application to configure them to your server before you put your server on the host.

For applications that use an application adaptor to interface to a persistent store, such as DB2 or CICS, you also need to install the specific application for this application adaptor. This is described at the end of this chapter.

7.2.1.6 Configure the server with the host

Your server has an application configured, but it is still hanging in thin air. You need to give your server a physical place to stay. You need to configure it with a host.

Pick up your server...

- Management Zones
  - Account MZ
    - Configurations
      - Account Config
        - Servers (free-standing)
          - **Account..Server -> Copy**

...and drop it onto the host.

- Hosts
  - **Your server host->Configure Server**

Under Configured Servers (free-standing), there should be an entry for the Account..Server.

7.2.2 Activation

At this stage, you are almost set to run your server. The only thing you need to do now is to activate your configuration.

- Management Zones
  - Account MZ (the MZ you created)
    - Configurations
      - **Account Config -> Activate**

In Systems Management, there is the notion of host and host image. A host is the physical host where you install your server, and a host image is a representation of the server processes running on the host.

When the activation process has finished, your server should appear under:

- Host Images
  - Your server host
    - Server Images
      - Account..Server

When you select the server, the status line of the output window should show: `AccountServer run request stopped 0 0 poor`.

Note: The previous activation step actually starts the server. Thus, the separate Run the Server step (below) is not required. If the status line says something similar to `AccountServer run request server running 5 5 excellent`, your server is activated and running properly.

7.2.3 Run the server

Once the activation is complete, you are ready to start the server application and run the client application that will access the Account component.

- Host Images
  - Your server host
    - Server Images
      - **Account..Server -> Run Immediate**

And voila! Your server is up and running, and you may test your client against it.

To check the status of your server, select your server and look at the status line below.

7.2.3.1 Configure DB2

If you want to install an application that uses DB2, you need to install the DB2-specific application on your server before you put your server on the host.

Note: You should make sure that you have created the database that this application uses, and that you have bound your application to the database. If your application uses iterators, you need to bind the CBConnector-specific bind files to your database. These files are:

- C:\CBroker\etc\db2cntcs.bnd
- C:\CBroker\etc\db2cntrr.bnd

Pick up the DB2 application...

- Host Images
  - Your server host
    - Application Family Installs
      - iDB2IMApplications
        - Application Installs
          - **iDB2IMServices -> Copy**

...and add the application to your configuration.

- Management Zones
  - Account MZ
    - Configurations
      - Account Config -> Add Application

Configure the DB2 application on your server. Pick up the application...

- Management Zones
  - Account MZ (the MZ you created)
    - Configurations
      - Account Config
        - Applications
          - **iDB2IMServices -> Drag**

...and configure it onto the server.

- Management Zones
  - Account MZ
    - Configurations
      - Account Config
        - Servers (free-standing)->Insert
          - **Account..Server -> Configure Application**

After the DB2 application has been added, you continue to configure the server by adding it to the host, and then activating the configuration. After the activation process has finished, make sure to stop the server. Finally, edit the **open string** for the database and add a user ID and password for both the **AccntDB** and the **DB2ReferenceCollections**.

- Host Images
  - Your server host
    - Server Images
      - Account..Server
        - XAResourceManagerImages
          - **AccntDB -> Edit**
            Add the user ID and password to open string in the Main folder, for instance CBSRFCDB, ID, password.
          - **DB2ReferenceCollections -> Edit**

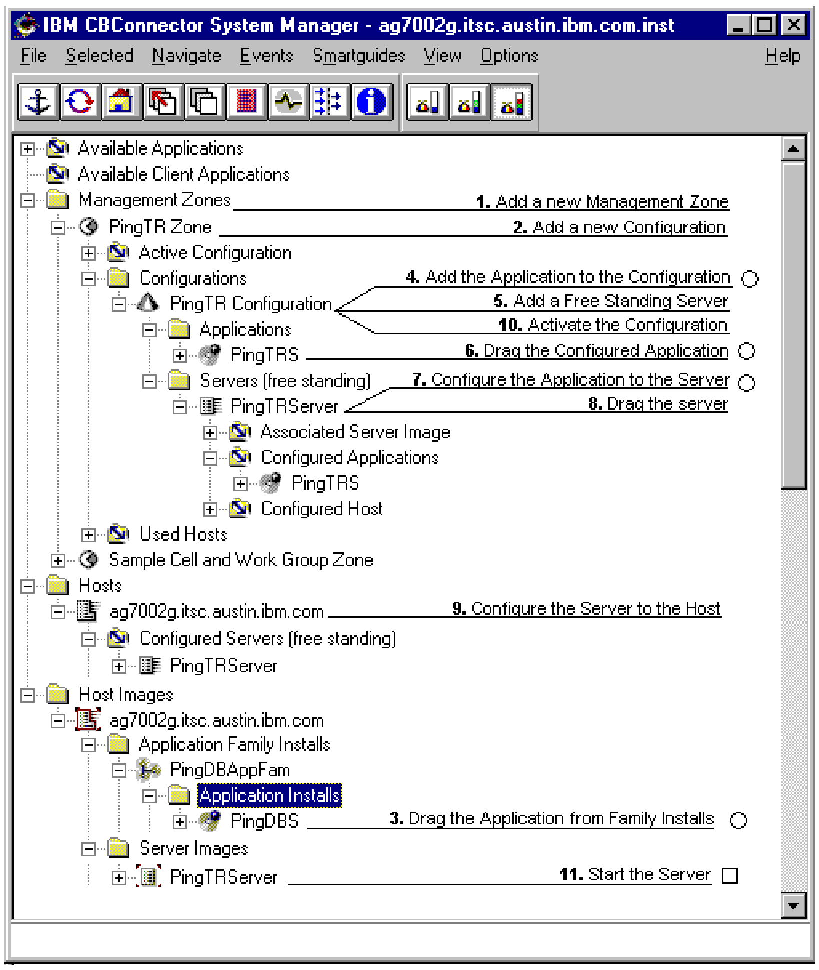
You may now start the server, and test your clients.

7.2.4 Systems management summary

Figure 7 shows a summary of the steps you need to perform to configure a server.

The circle denotes which steps you need to perform repetitively for each application that you want to put on your server.

The square denotes that you don't need to start the server in CBConnector, since the activation process does it for you. It is also a reminder to set the open string of your database-dependent applications, if you have any, before you restart the server.


IBM CBConnector System Manager - ag7002g.itsc.austin.ibm.com.inst
File
Selected
Navigate
Events
Smartguides
View
Options
Help

```
Available Applications
Available Client Applications
Management Zones                          1. Add a new Management Zone
  PingTR Zone                              2. Add a new Configuration
    Active Configuration
    Configurations                         4. Add the Application to the Configuration ○
      PingTR Configuration                 5. Add a Free Standing Server
                                           10. Activate the Configuration
        Applications
          PingTRS                          6. Drag the Configured Application ○
        Servers (free standing)            7. Configure the Application to the Server ○
          PingTRServer                     8. Drag the server
            Associated Server Image
            Configured Applications
              PingTRS
            Configured Host
    Used Hosts
  Sample Cell and Work Group Zone
Hosts
  ag7002g.itsc.austin.ibm.com              9. Configure the Server to the Host
    Configured Servers (free standing)
      PingTRServer
Host Images
  ag7002g.itsc.austin.ibm.com
    Application Family Installs
      PingDBAppFam
        Application Installs
          PingDBS                          3. Drag the Application from Family Installs ○
    Server Images
      PingTRServer                         11. Start the Server ☐
```

Figure 7. Systems management summary

In the next section, you can read about the SmartGuides that help you to define your server more easily.

7.2.5 System Management Interface SmartGuides

The System Management Interface has a set of SmartGuides that help you create and configure your application servers.

7.2.5.1 Configure application server

Here are the SmartGuide options:

- Create servers
- Create client styles
- Configure servers
- Configure client styles

Following are the alternative configuration steps:

1. Define your Management Zone.
2. Define your Configuration.
3. Server group: The default Server Group is selected; you can set a new name. You have no possibility to create a free-standing server, but in 7.2.5.2, “Migration to a free-standing server” below, you can convert a server-member of a server group to free-standing server.
4. Configure your server.
   - Applications.
   - Select **Management Zone**.
   - Select **Configuration**.
   - Select from the available server groups your server group.

7.2.5.2 Migration to a free-standing server

Here are the steps:

1. Management Zones, Configurations, Server Groups, <your server group name>, servers (member of group),<your server>
2. Right mouse click: Convert to free-standing

Part 3. Go

This part of our cookbook focuses on the incremental sample development. You will learn to design your first object model in CBConnector Object Builder and follow all the necessary steps to compile and package your distributed application. We also give you a suggestion on how to approach a client-side development. In Chapter 8, “The Account scenario” on page 101, you can read about our account scenario and the incremental samples.

Good luck!

Chapter 8. The Account scenario

In this book, we use a simple banking scenario to guide you through the basic parts of Component Broker. We use a simple example, which we extend incrementally, demonstrating one feature of CBConnector at a time.

8.1 Scenario objects and functionality

For the major part of this cookbook, the only object introduced is a simplistic bank account. An account's class — suitably also called **Account** — offers the following functionality:

- Users can look up information on the account by using its 10-digit account number.
- They can also change the account's balance as long as the new balance is zero or above.
- Using two accounts, money can be transferred between them, provided there is enough money in the source account.

  You can see from Figure 8 that we make the operation **changeBalance(...)** throw an exception if there isn't enough money in the account. Whether this is the right choice in a production environment is questionable. However, we wanted to show the use of exception-handling in a CORBA environment.

The following IDL file shows the interface of **Account**:

```
#ifndef _Account_idl
#define _Account_idl

// Generated from Account.idl
// on 05/22/98 12:05:52
// by Object Builder

#include "IManagedClient.idl"

interface Account : IManagedClient::imanageable
{
  exception NotEnough {
  }; // end exception  NotEnough
  attribute string<10> accountNumber;
  attribute string<30> accountHolder;
  double getBalance( );
  void changeBalance(in double anAmount ) raises
       (Account::notenough); }; // end interface Account
#endif
```

Figure 8. Account IDL file

The main criteria for selecting this scenario was that it should be simple enough to not "cloud" our real goal of explaining and exercising key Component Broker functionality. We didn't want you to have to wade through a lot of unnecessary application detail before you could even begin to look at the Component Broker functionality in our sample code.

8.2 Sample implementation sequence

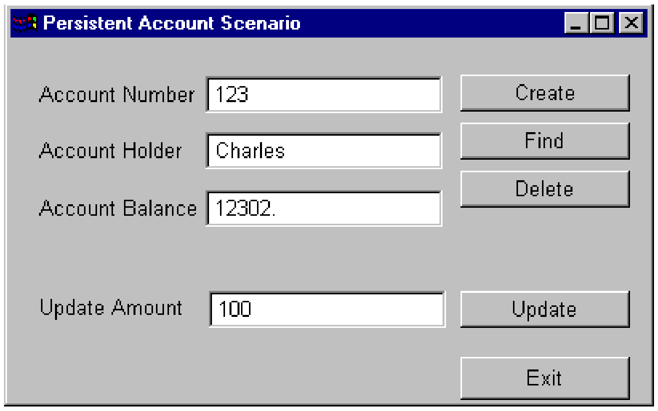
We found that an incremental approach to introducing Component Broker functionality would be best suited to the cookbook approach. Starting with the very basics — such as hooking up to the Naming Service — we introduce one aspect at a time, always building on previous code and explanations.

Before running these samples, we presuppose that you have successfully installed and run the Ping samples, to verify that your system is installed correctly. Another prerequisite is that the client is hooked up to its bootstrap host and the root Naming Service context.

The following is the implementation sequence of our samples:

1. Transient Object sample
   The transient account sample operates on one transient account at a time. A Transient Object is not persistent or stored, which means that it will disappear when the server is stopped. As far as Component Broker facilities are concerned, we deal with the following in this sample:
   - Initializing the global Component Broker environment
     This includes resolving the references to the ORB and the Naming Service.
   - Finding a home
     You can then create new accounts, as well as find and display information on existing accounts.
     In Component Broker terms, this means creating and using Primary Key objects, and performing methods on these objects. For example, having found an Account Object, you can invoke the **changeBalance()** method on it.
   - Create Transient Objects using Copy Helper
     Here we use a local Copy Helper Object for creating transient accounts. It does so by putting the two public instance variables into the Copy Helper and then calling **createFromCopyString** on the Transient Home. This optimizes performance over the ORB. Since you create and instantiate the object with values using only one call, you don't need to set values by calling the object's setter methods.
   - Java Business Object on the server
     Since Version 1.2 of Component Broker, Java Business Objects are supported in addition to C++ Business Objects. In the Transient sample, we show you how to generate Java Business Objects.
2. Specialized home sample
   In this sample, we show you how you can extend the previous sample with a specialized home that enables you to create your own interface to manipulate the Home Objects. We create new accounts from the account holder's name, that the end user gives us. The methods we create are **createAccount** and **findByAccountNumber**, which do not require a Primary Key from the client.
3. Persistent Object sample
   The persistent account sample lets you store your objects in a DB2 database table using the CBConnector DB2 Application Adaptor.
   From a client perspective, the sample executes exactly the same as the transient one. The only difference is that it connects to a different Factory Finder, as explained in Chapter 18, “LifeCycle Service” on page 287.

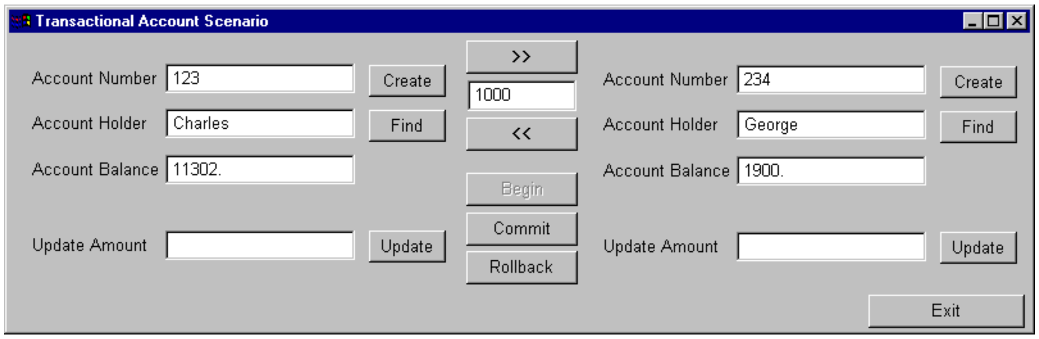
On the server side, it introduces the capability to connect to a back-end DB2 database to provide persistence for its essential state through relational database tables as shown in Figure 9.



Account Number: 123
Account Holder: Charles
Account Balance: 12302.
Update Amount: 100

Figure 9. The persistent account sample

4. Transactional Object sample
   The transactional account sample shown in Figure 10 is making use of two persistent accounts and shows functionality such as **begin transaction**, **end transaction**, **commit**, and **rollback**.



Account Number: 123
Account Holder: Charles
Account Balance: 11302.
Update Amount:

1000

Account Number: 234
Account Holder: George
Account Balance: 1900.
Update Amount:

Figure 10. The transactional account sample

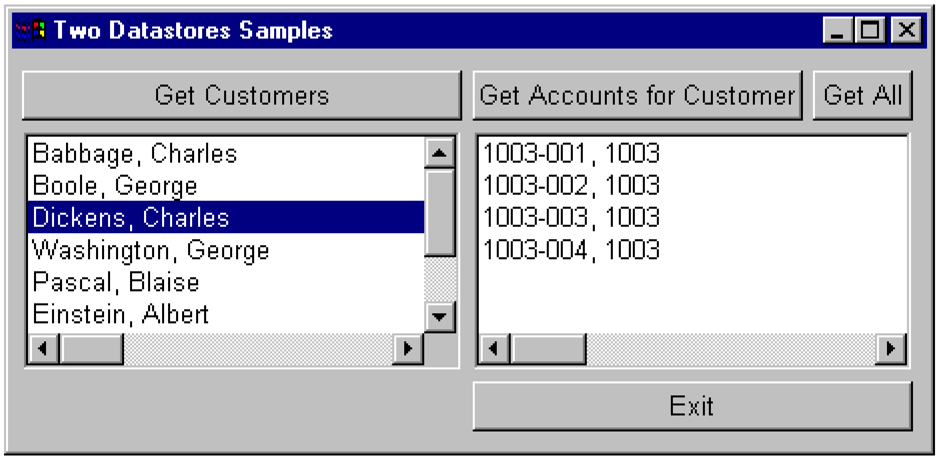
5. Object with UUID key and model dependency
   This sample introduces a Transient Object which is assigned a universal unique identifier key (UUID key). The client is similar to the transactional sample, but in the place of managing two Account Objects on the client side, the client interfaces to a Transaction Manager Object on the server that handles a transaction in one method. It is this Manager Object that is created by a UUID Key.

6. Event service
   Here, we implement a notification scheme (not to be confused with the Notification Service, planned by OMG) that allows a client to execute callback methods upon receiving an event from the server.
   This is a multi-threaded client that updates one Transient Server Object concurrently. In the sample, we show the use of Concurrency Service constructs, such as locks and lock sets.

7. Query service — reuse of an existing table
   This sample shows how to define a Persistent Object from a relational table by reusing its database data definition language statements. The resulting Persistent Object can then be linked up with the Data Objects and Business Objects developed in the usual top-down way.
   The sample also introduces the concept of the **iterator, which lets you work on a collection of objects in a home. We demonstrate four types of iterator methods:**

   - **createIterator** — Create a general iterator on all objects in a home
   - **homeEvaluator** — Obtain an iterator including only those objects that match an evaluator (or where clause)
   - **evaluateToIterator** — Obtain an array of object references by passing an OO-SQL statement
   - **evaluateToDataArray** — Obtain an array of attributes as strings by passing an OO-SQL statement

8. Meet-in-the-middle paradigm with two datastores
   This sample builds on the Persistent Object sample and the sample in Chapter 19, “Query Service — reuse of an existing table” on page 291.
   It implements a 1-to-n relationship from a Customer Object to its Account Objects, so you can list all the accounts of a customer. Account Objects might store their persistent state in one back-end database while Customer Objects could use a different one. See Figure 11.



Babbage, Charles
Boole, George
Dickens, Charles
Washington, George
Pascal, Blaise
Einstein, Albert

1003-001, 1003
1003-002, 1003
1003-003, 1003
1003-004, 1003

Figure 11. The two datastores sample

9. IOM with two datastores
   Here, we extend the previous sample to show how two Business Objects implemented in different languages communicate with each other. The Account Business Object is implemented in C++, and it communicates with the Customer Business Object implemented in Java.

We suggest that you work through the samples in the specified sequence.

Chapter 9. Client programming model — atomic fundamentals

In this chapter, we present the basic fundamentals of the client programming model. We will use C++ sample code to visualize the syntax. In the next Chapter 10, “Sample client development approach” on page 117, you can study the layered structure of a client application, that implements the atomic fundamentals in C++ and Java. The ActiveX implementations will keep the atomic approach.

For the syntax reference, read the CBConnector product manuals.

After reading this chapter, you should understand the following basic steps in the client development model:

- Initializing the ORB
- Finding a Factory Finder
- Finding a home
- Creating a Primary Key
- Creating an object
- Finding an object
- Invoking operations on an object
- Cleaning-up the client environment
- Transaction support

9.1 Initializing the ORB

Upon client program startup, one of first things that has to be done is to initialize the Object Request Broker (ORB) and deliver the initial reference to the Naming Service. Component Broker provides you with the **CBSeriesGlobal** object to accomplish these tasks. The **CBSeriesGlobal::Initialize** method calls the **CORBA::ORB_init** function and the **resolve_initial_references** method on the ORB object on your behalf. The **resolve_initial_references** method contacts the ORB daemon in the bootstrap host and requests a reference for the root naming context.

The following code snippet shows what you have to do to get your ORB environment set up.

```
    ...
    CORBA::ORB_var op;
    ...

 // method:     initializeNS()
 // source file: AccntTr.cpu
    ...
 // Initialize the CBConnector environment
    CBSeriesGlobal::Initialize();
     ...
// Initialize ORB pointer
   op = CBSeriesGlobal::orb();
   ...
// Check to see if Name Service initialized O.K.
   if ( CORBA::is_nil( CBSeriesGlobal::nameService()   ) ) {
   ...Do Error processing and whatever is necessary
   ...at this time
     ERROR, the CBSeriesGlobal::nameService pointer is nil.
   ...
   }

// If it did initialize O.K. - we are in good shape
   else {
   ...
   }
```

9.2 Finding a Factory Finder

After initialization, you are ready to start using the ORB and the Services. The next thing you usually need is a Factory Finder. A *Factory Finder* is part of the LifeCycle Service. You can resolve a Factory Finder by calling **resolve_with_string** on the root naming context and passing in the name of an object you are trying to resolve. This results in a method call on the real object residing in the server, since the naming trees only reside in the server. The Naming Service searches the Cell Directory Service (CDS) for a corresponding entry satisfying requested criteria (defined by the Location Object) and if found, returns a reference to Factory Finder. Otherwise, an exception is thrown.

Factory Finders vary according to the samples we are dealing with in this publication. Since this scenario uses transient accounts, we want to find a Factory Finder that is instantiated with a Location Object defined to establish **AccountTRServer** as its scope. Please refer to Chapter 18, “LifeCycle Service” on page 287 for more information.

```
IExtendedLifeCycle::FactoryFinder_var myFinder;
CORBA::Object_var it = CBSeriesGlobal::nameService()- >
      resolve_with_string(theScopeString);
myFinder = IExtendedLifeCycle::FactoryFinder::_narrow (it);
if (CORBA::is_nil(myFinder))
   ...do appropriate Error processing
   ...ERROR, No FactoryFinder Object found
 else
   ...We are in good shape so far
```

9.3 Finding a home

The next object you need is a home. You find a home by calling **find_factory_with_string** on the Factory Finder reference. Depending whether you are trying to find or create objects, you call **findByPrimaryKeyString** or **createFromPrimaryKeyString** on the home reference, passing in the Key string formed with the Primary Key object (by calling **IManagedLocal::toString**). Objects can also be created and initialized within the same call with **createFromCopyString** by passing in the copy string formed with the **CopyHelper** object. If successful, all these methods will return an object reference.

```
::CosLifeCycle::Factory_var theFactory;
theFactory = myFinder->find_factoriy_from_string(
                      "Account.object interface");
theAccountHome = IManagedClient::IHome::_narrow(t heFacto
if (CORBA::is_nil(theAccountHome) == FALSE)
   ...AccountHome found, we are fine
   ...ERROR, No Home for Account objects was found
```

9.4 Creating the Primary Key

In order to operate on objects in the home, we have to set up an object of class **PrimaryKey**. In our case, we use the 10-character accountNumber field as our Primary Key and create it as follows:

```
// First create a PrimaryKey
 ...
thePrimaryKey = createPrimaryKey(anAccountNumber);
 ...

// Since PrimaryKeys have to be created in more than one place,
// it made sense to extract that functionality into one method.

AccountKey* AccntTr::createprimarykey(char* anAccountNumber)
{
   ...
   AccountKey* tmpKey = AccountKey::_create();
   tmpKey->accountNumber((const char*)anAccountNumber);
   ...
   return tmpKey;
}
```

9.5 Creating an object

In the following example, we use the Primary Key to create an Account Object.

```
 // method:      createAccount(...)
 // source file: AccntTr.cpu

 // If the Account doesn't exist create it
 moPtr = theAccountHome->createFromPrimaryKeyString
                    (*thePrimaryKey->toString());
 if( CORBA::is_nil(moPtr))
   ...we didn't manage to create the account

 // narrow the MO to an Account Object
 theAccount = Account::_narrow(moPtr);
if( CORBA::is_nil(theAccount))
  ...we couldn't _narrow the MO to an Account Object
  ...(which shouldn't really occur, if everything is set-up
  ...correctly.
```

9.6 Finding an object

The following example shows the code for finding an Account using its 10-character account number.

```
// Find the Account using the PrimaryKey
  try
  {
     moPtr = theAccountHome->findByPrimaryKeyString(
                            *thePrimaryKey->toString());

     if (CORBA::is_nil(moPtr) == FALSE) { // Managed Object  found ?

        // Since the type of the object returned is MO, it must be
       // narrowed to an Account Object
      theAccount = Account::_narrow(moPtr);

       if ( CORBA::is_nil(theAccount) == FALSE)
       ... we have found an Account Object
       else
       ... we have found some MO, but definitely not an Account
    }
    else
       ...We didn't even find an MO

 }

 catch
    ... handle all kinds of errors in appropriate catch blocks,
    ... such as
        IManagedClient::INoObjectWKey
        IManagedClient::IInvalidKey
        CORBA::INV_OBJREF
        CORBA::SystemException
```

9.7 Invoking operations on an object

Operations on server objects are invoked by using the interface of their client proxies.

In the following example, we invoke the methods to set the name of the Account Holder and to change the account's balance. The code snippet assumes that we have just created a new account or else found an existing one. In any case, we hold an object reference to an account.

```
// Set the Name of the Account Holder
theAccount->accountHolder(anAccountHolder);

// Change the Account Balance (if there is not enough money
// in the account, Exception "NotEnough" will be thrown)
theAccount->changeBalance(theAmount);
```

9.8 Cleaning up the client environment

You may have to do some "cleansing" operations from time to time. As an example, when you are done with any objects, you should release the references to them by invoking **release()** on the proxies. This tells the server to decrease the reference count, an action that is vital for memory management.

"Releasing" an object is quite different from actually "removing" it. The latter has the expected effect on the server side, for example, to cause deletion of the corresponding row in the SQL table within the back-end database.

```
// Release Account Proxy
  if (CORBA::is_nil(theAccount) == FALSE)
    CORBA::release(theAccount);
```

As a final "cleansing" action, you should release the reference to the ORB.

```
// Release the ORB Pointer
CORBA::release(op);
```

9.9 Transaction support

In order to invoke methods on the server objects that use transactions, we can either explicitly use a **CosTransactions::Current** object on the client or let the container policy for Transaction Services handle this for us. In case your client application should have control over the scope of the transaction, we will use the **CosTransactions::Current** object, which contains the required methods to perform the following on our transactions:

- Start
- Commit
- Roll back

For more details, please read Chapter 14, “Transactional Object sample” on page 217.

In our simple C++ sample, we show you how to declare the necessary objects and initialize them at runtime, as follows:

```
   ...
   ...


   }
         Get Current Object
         -------------------
org.omg.CORBA.Current orbCurrent;
org.omg.CORBA.Object obj =
orb.resolve_initial_references("TransactionCurrent");
org.omg.CosTransactions.Current current =
org.omg.CosTransactions.CurrentHelper.narrow(obj);
try
   {
      cout << "About       current->begin( );" << endl;

      current->begin( );

      key = AccountKey::_create( );
      key->accountNumber( argv[1]);
      // convert the contents of the key to ByteString
      keyString = key->toString( );

      try
      {

      ..
      ..
      ..


        mo = accountHome->findByPrimaryKeyString( *keyString );

      current->commit( 1 );
   }

   return 0;

} // end main()
```

9.10 Compiling client applications

Now, when you understand the basic fundamentals, your next question should be, "What do I need to compile my client application?" For the answer, we direct you to the following sections of this document:

6.7.1, “C++ client programming model” on page 75

6.7.2, “Java client programming model” on page 76

6.7.4, “ActiveX client programming model” on page 76

Chapter 10. Sample client development approach

This chapter describes a sample approach to client application development. We try to isolate the client application code and the GUI parts from having to know about the CBConnector client programming model.

We would like to point out that our sample approach is not fully implemented and complete.

Note

Production-level code will have difficulty using this pattern because all of the error handling involves *count* and then *return*.

The following topics are described in this section.

- The Object model
- C++ implementation
- Java implementation
- ActiveX implementation

The C++ and Java implementation is done according to the Object model, while the ActiveX implementation is not. The aspects of the ActiveX implementation relating to CBConnector are described in this chapter; so the description of the various samples may focus on the specific code needed for the sample.

10.1 Object model

A CBConnector client program ought to know as little about CBConnector internals as possible. There is, however, some handshaking which needs to be done before the client program may concentrate on its Business Objects.

The tasks which have to be performed before a Business Object is obtained are the following:

- ORB initialization
- Resolving Name Service
- Resolving Factory Finder
- Resolving factory
- Resolving home
- Create Primary Key (or copy string)
- Create Managed Object (or find it)
- Narrow to Business Object

In addition to these tasks, it is natural to implement the transactional services in a way that isolates them from the client application. The services needed in our samples are:

- begin
- commit
- rollback

The last category of services that may be implemented in a general way is a set of Query Services based on OO-SQL.

This chapter describes a structure of three layers implementing the tasks and services mentioned above.

The reason for breaking the structure into three layers, is that the application needs to create one and only one proxy object representing the ORB, hence the bottom layer.

For every Business Object, there may be an encapsulation of the corresponding home, which is application-independent, hence the second (general) layer.

The general layer is also a natural location for the transaction support.

The top layer contains the minimal application-dependent information in order to create/find Business Objects.

Figure 12 illustrates the layered structure of the client application.

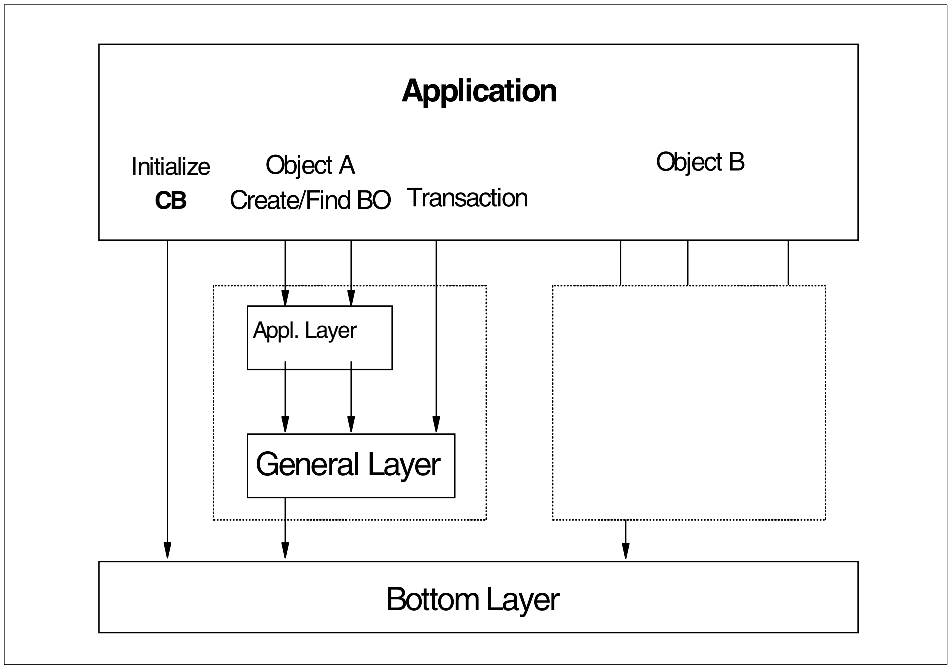


Figure 12. Layered client application

10.1.1 Bottom layer

Every client application has to connect to CBConnector, represented on the client side by the ORB. This has to be done once per client application. The bottom layer encapsulates the proxy object representing the ORB. An **init** method is invoked to create the proxy object. The naming resolution is also performed at this stage.

This layer would typically also provide utility methods relating to the proxy object, for the general layer to use. In addition, utility methods for Query Services may be provided here.

10.1.2 General layer

In the general layer, we focus on Managed Objects, transaction services and session services.

This implies that we need one instance of the general layer per Business Object. The layer provides an **init** method with two parameters: scope and interface.

The result of the **init** method is that a home is found (if everything is OK), and a pointer to the Home Object is stored within the general layer.

If the Business Object needs to use transactional services, they are provided as methods in this layer.

- beginTX
- commitTX
- rollbackTX

Each one of these methods checks to see if there is a current object stored within the general layer, and if not, it is created and stored. Then the methods are executed directly on the current object.

Finally, the general layer supports methods to create and to find Managed Objects:

- createMO
- createMOfromCopy
- findMO

The **createMO** uses a key string as input parameter, while the **createMOfromCopy** uses a copy string as input parameter.

10.1.3 Application layer

Obviously, the application layer is dependent upon your application. In this book, the most common Business Object is the Account Object. In this chapter, we describe our approach to creating the application layer for this particular object. We have samples in the book which require different objects, and they are described accordingly in their respective chapters.

In the (account) application layer, we create/find our Business Objects. To do this, we need application-specific information (include files for C++ and class files for Java).

The methods provided in the application layer are **createBO** and **findBO**. They need the **createMO** (create Managed Object) and **findMO** (find Managed Object) from the general layer.

The main task of the application layer is to provide the key for **createMO** and **findMO**, based on the key object of the Business Object. It would also be responsible for creating a copy string, when the **createMOfromCopy** method in the general layer is invoked.

After having invoked **createMO**/**findMO** in the general layer, the application layer uses the Business Object's helper class to narrow the Managed Object to the Business Object.

In the next sections, we show the actual implementations of the layered approach, in C++ and Java. We implemented the layers a little differently in the respective languages, to better conform to the language constraints.

10.2 C++ Implementation

The C++ implementation is a two-layered implementation. The bottom layer and the general layer are represented by one class called **cbcproxy**. The application layer is represented by a class called **acctproxy**, which is a subclass of **cbcproxy**. Of course, this application-specific class varies from Business Object to Business Object, but because most of our samples use the **Account** class, we decided to describe the **acctproxy** class (which uses information from the **Account** class).

The two classes are described next.

10.2.1 The cbcproxy class

The **cbcproxy** class implements four categories of methods:

- The create/find Managed Objects methods
- The transactional methods
- The session methods
- The query methods using OO-SQL

In addition, it implements two **init** methods, one returning a pointer to a factory and one finding a default home, stored as an instance variable and returning 1 if successful, otherwise returning 0.

Both methods initialize the ORB, which is held as a class variable. The methods check if it is already initialized in order to initialize it only once.

In the following, we show you in detail the implementation of the two **init** methodsInitialize CBConnector and Resolve to Home.

Here we called the private **initCBC** method of the **cbcproxy** class to get the factory, and then we narrowed the factory in order to resolve to the home. A pointer to the home is stored in an instance variable in order to use it in succeeding calls. The type of the *Home* instance variable is *IManagedClient::IHome_ptr*.

The following code snippet in Figure 13 shows you the overloaded **init** method, returning the resolved factory only.

```
/*
;------------------------------------------------------------------;
;       initialize CBC and resolve to factory                     ;
;------------------------------------------------------------------;
*/
CORBA::Object_ptr _Export cbcproxy::init(char *scope, char *interface,
                                  int dummy)
  {
/*
;------- Initialize CBC
*/
   return initCBC(scope, interface);
  }
```

Figure 13. Initialize CBConnector and resolve to factory

Note that in order to overload a method, it has to differ from the original method in the parameters passed, hence the dummy parameter. This method is only calling the private method, **initCBC**, of the **cbcproxy** class and passing its return value to the caller.

The **initCBC** method initializes the ORB, resolves the Factory Finder, and resolves the factory. The implementation of these tasks is shown in the following code snippets. The **initORB** is a private method of the **cbcproxy** class, while the "Resolve to factoryfinder" and "Resolve to factory" code snippets are taken directly from the source code of the **initCBC** method.

In the code snippet in Figure 14, the global initializer is called, the ORB is initialized and the Name Service pointer is validated.

```
/*
;------------------------------------------------------------------;
;       initialize the ORB                                         ;
;------------------------------------------------------------------;
*/
int  cbcproxy::initORB()
  {
/*
;------- Call the Global Initializer
*/
   CBSeriesGlobal::Initialize();
/*
;------- initialize the ORB
*/
   cout << "ORBinit" << endl;
   orb = CBSeriesGlobal::orb( );
   if (CORBA::is_nil(orb))
      {
       cout << "Couldn't initialize the ORB" << endl;
       return FALSE;
      }
/*
;------- Check if the Name Service pointer is valid
*/
   cout << "Resolving Name Service" << endl;
   if (CORBA::is_nil(CBSeriesGlobal::nameService()))
      {
       cout << "Error in service pointer" << endl;
       return FALSE;
      }
   return TRUE;
  }
```

Figure 14. Initialize the ORB

The type of the *orb* variable is *CORBA::ORB_ptr*, and this variable is a class variable of **cbcproxy**.

A code snippet to resolve to Factory Finder is shown in Figure 15.

```
cout << "Resolving factory finder with scope: " << endl;
cout << "[" << fullscope << "]" << endl;
try
  {
   obj = CBSeriesGlobal::nameService( )->resolve_with_string(fullscope);
   ff = IExtendedLifeCycle::FactoryFinder::_narrow(obj);
   ::CORBA::release(obj);
  }
catch(CORBA::UserException &e)
  {
   cout << "Couldn't resolve to factory finder - exception: "
       << e.id() << endl;
   return NULL;
  }
catch(CORBA::SystemException &e)
  {
   cout << "Couldn't resolve to factory finder - exception: "
       << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
   return NULL;
  }
catch( ... )
  {
   cout << "Couldn't resolve to factory finder" << endl;
   return NULL;
  }
cout << "Factory finder OK" << endl;
```

Figure 15. Resolve to Factory Finder

The *fullscope* variable is a string containing the full scope path, as follows: *"/host/resources/factory-finders/"xxxxx"*, where xxxxx is the scope supplied. The type of the *obj* variable is *CORBA::Object_ptr*, while the type of the *ff* variable is *IExtendedLifeCycle::FactoryFinder_ptr*.

A code snippet to resolve to factory is shown in Figure 16.

```
cout << "Resolving factory with interface: " << endl;
cout << "[" << interface << "]" << endl;
try
  {
   obj = ff->find_factory_from_string(interface);
  }
catch(CORBA::UserException &e)
  {
   cout << "Couldn't resolve to factory - exception: "
        << e.id() << endl;
   return NULL;
  }
catch(CORBA::SystemException &e)
  {
   cout << "Couldn't resolve to factory - exception: "
        << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
   return NULL;
  }
catch( ... )
  {
   cout << "Couldn't resolve to factory " << endl;
   return NULL;
  }
cout << "Factory OK" << endl;
```

Figure 16. Resolve to factory

The *interface* variable is a string containing the interface information needed, as follows: *"xxxxx.object interface"*, where xxxxx is the interface supplied. The type of the *obj* variable is *CORBA::Object_ptr*, while the type of the *ff* variable is *IExtendedLifeCycle::FactoryFinder_ptr*.

Having shown you the different tasks involved in CBConnector initialize, we are ready to look into the higher level services of:

- Create/find Managed Objects
- Transaction services
- Session services
- Query services

The following code snippets illustrate the two Managed Object create methods, **createMO** and **createMOfromCopy**, as well as the **findMO** method. We start with the example in Figure 17.

```
/*
;------------------------------------------------------------------;
;       Create Managed Object (MO)                                 ;
;------------------------------------------------------------------;
*/
IManagedClient::IManageable_ptr _Export
            cbcproxy::createMO(ByteString keyString)
 {
  if (Home == NULL)
    {
     cout << "Couldn't find the home" << endl;
     return NULL;
    }
  try
    {
     mo = Home->createFromPrimaryKeyString(keyString );
    }
  catch(IManagedClient::IInvalidKey)
    {
     cout << "Invalid key" << endl;
     return NULL;
    }
  catch(IManagedClient::IDuplicateKey)
    {
     cout << "Managed object already created" << endl;
     return NULL;
    }
  catch(CORBA::UserException &e)
    {
     cout << "Couldn't create Managed Object - exception: "
         << e.id() << endl;
     return NULL;
    }
  catch(CORBA::SystemException &e)
    {
     cout << "Couldn't create Managed Object - exception: "
         << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
     return NULL;
    }
  catch( ... )
    {
     cout << "Couldn't create Managed Object" << endl;
     return NULL;
    }
  return mo;
 }
```

Figure 17. Create Managed Object from Primary Key

We are basically executing one single statement:

```
mo = Home->createFromPrimaryKeyString(keyString );
```

In this code snippet, while all the rest of the code handles errors, mainly through catch statements, the catch statements become more and more general towards the end of the code. We have tried to handle the try/catch clauses in this way throughout this chapter.

The type of the *mo* variable is *IManagedClient::IManageable_ptr*.

Now consider the code snippet in Figure 18:

```
/*
;-------------------------------------------------------------------;
;       create Managed Object (MO) from Copy string                 ;
;-------------------------------------------------------------------;
*/
IManagedClient::IManageable_ptr _Export
              cbcproxy::createMOfromCopy(ByteString copyString)
 {
  if (Home == NULL)
    {
     cout << "Couldn't find the home" << endl;
     return NULL;
    }
  try
    {
     mo = Home->createFromCopyString(copyString );
    }
  catch(IManagedClient::InvalidCopy)
    {
     cout << "Invalid copy" << endl;
     return NULL;
    }
  catch(IManagedClient::IDuplicateKey)
    {
     cout << "Managed object already created" << endl;
     return NULL;
    }
  catch(CORBA::UserException &e)
    {
     cout << "Couldn't create Managed Object - exception: "
         << e.id() << endl;
     return NULL;
    }
  catch(CORBA::SystemException &e)
    {
     cout << "Couldn't create Managed Object - exception: "
         << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
     return NULL;
    }
  catch( ... )
    {
     cout << "Couldn't create Managed Object"  << endl;
     return NULL;
    }
  return mo;
 }
```

Figure 18. Create Managed Object from Copy Helper

The type of the *mo* variable is *IManagedClient::IManageable_ptr*.

Next, consider the code snippet in Figure 19:

```
/*
;-------------------------------------------------------------------;
;       find Managed Object (MO)                                    ;
;-------------------------------------------------------------------;
*/
IManagedClient::IManageable_ptr _Export
             cbcproxy::findMO(ByteString keyString)
 {
  if (Home == NULL)
    {
     cout << "Couldn't find the home" << endl;
     return NULL;
    }
  try
    {
     mo = Home->findByPrimaryKeyString(keyString );
    }
  catch(IManagedClient::IInvalidKey)
    {
     cout << "Invalid key" << endl;
     return NULL;
    }
  catch(IManagedClient::INoObjectWKey)
    {
     cout << "Managed object doesn't exist " << endl;
     return NULL;
    }
  catch(CORBA::UserException &e)
    {
     cout << "Couldn't find Managed Object - exception: "
         << e.id() << endl;
     return NULL;
    }
  catch(CORBA::SystemException &e)
    {
     cout << "Couldn't find Managed Object - exception: "
         << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
     return NULL;
    }
  catch( ... )
    {
     cout << "Couldn't find Managed Object"  << endl;
     return NULL;
    }
  return mo;
 }
```

Figure 19. Find Managed Object from Primary Key

The type of the *mo* variable is *IManagedClient::IManageable_ptr*.

We are now ready to show the implementation of the transaction services, where we selected to implement just the methods needed for the samples:

- beginTX
- commitTX
- rollbackTX

The code snippet in Figure 20 illustrates all three methods:

```
/*
;-------------------------------------------------------------------;
;       xxxxx transaction (xxxxx is begin, commit or rollback)      ;
;-------------------------------------------------------------------;
*/
int _Export cbcproxy::xxxxxTX()
  {
/*
;------- Do we have the current object?
*/
   if (current == NULL)
      if (!getCurTX())
         return FALSE;
/*
;------- xxxxx transaction
*/
   try
      {
       current->xxxxx( );  // Parameter 0,1 for commit (for debug)

       cout << "xxxxx transaction" << endl;
       return TRUE;
      }
   catch (CosTransactions::NoTransaction)
      {
       cout << "Caught - no transaction exception " << endl;
       return FALSE;
      }
   catch(CORBA::UserException &e)
      {
       cout << "Couldn't xxxxx transaction - exception: "
            << e.id() << endl;
       return FALSE;
      }
   catch(CORBA::SystemException &e)
      {
       cout << "Couldn't xxxxx transaction - exception: "
            << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
       return FALSE;
      }
   catch( ... )
      {
       cout << "Couldn't xxxxx transaction" << endl;
       return FALSE;
      }
 }
```

Figure 20. Begin, commit, and rollback transaction

You see that we start out to check the status of the instance variable *current* of type *CosTransactions::Current_ptr* , which is a proxy object to represent the transactional status. The private method **getCurTX** of the **cbcproxy** class instantiates the proxy object, as shown in Figure 21:

```
/*
;------------------------------------------------------------------;
;       get current proxy object for TX                            ;
;------------------------------------------------------------------;
*/
int cbcproxy::getCurTX()
 {
CORBA::Current_ptr tmp;

  tmp = orb->get_current("CosTransactions::Current");
  current = CosTransactions::Current::_narrow( tmp );
  CORBA::release( tmp );
  if (current == NULL)
    {
     cout << "Couldn't get current object" << endl;
     return FALSE;
    }
  return TRUE;
 }
```

Figure 21. Get current transactional proxy object

Notice the parameter to the **get_current** method. The temporary object returned from the operation is narrowed by the **_narrow** method of the **CosTransactions** class.

Of the session services available in CBConnector, we implemented only the two necessary for us to do the CICS sample:

- beginSess
- endSess

The code snippet in Figure 22 shows the two methods:

```
/*
;--------------------------------------------------------------------;
;        xxxxx Session  (xxxxx is begin or end)                      ;
;--------------------------------------------------------------------;
*/
int _Export cbcproxy::xxxxxSess()
  {
/*
;------- Do we have the current object?
*/
   if (cursess == NULL)
      if (!getCurSess())
         return FALSE;
/*
;------- xxxxx session
*/
   try
      {
       cursess->xxxxxSession("session");
       cout << "Session xxxxx" << endl;
       return TRUE;
      }
   catch(CORBA::UserException &e)
      {
       cout << "Couldn't xxxxx session - exception: "
            << e.id() << endl;
       return FALSE;
      }
   catch(CORBA::SystemException &e)
      {
       cout << "Couldn't xxxxx session - exception: "
            << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
       return FALSE;
      }
   catch( ... )
      {
       cout << "Couldn't xxxxx session" << endl;
       return FALSE;
      }
 }
```

Figure 22. Begin/end session

You see that we start out to check the status of the instance variable *curses* of type *ISessions::Current_ptr* , which is a proxy object to represent the session status.

The parameter to the **xxxxxSession** method is required, but has no documented meaning.

The private method **getCurSess** of the **cbcproxy** class instantiates the proxy object, as shown in Figure 23:

```
/*
;------------------------------------------------------------------;
;       get current proxy object for session                      ;
;------------------------------------------------------------------;
*/
int cbcproxy::getCurSess()
 {
CORBA::Current_ptr tmp;

   tmp = orb->get_current("ISessions::Current");
   cursess = ISessions::Current::_narrow( tmp );
   CORBA::release( tmp );
   if (cursess == NULL)
     {
      cout << "Couldn't get current object" << endl;
      return FALSE;
     }
   return TRUE;
  }
```

Figure 23. Get current proxy object for session

Notice the parameter to the **get_current** method. The temporary object returned from the operation is narrowed by the **_narrow** method of the **ISessions** class.

10.2.2 The acctproxy class

In this section, we describe the **acctproxy** class because this is the most typical application-specific class used in our samples. The more specific application-related classes are described in the chapters where they are used.

The **acctproxy** class implements the following methods:

- initCBC
- **createBO** (from Primary Key)
- **createBO** (from Copy Helper)
- findBO

The **initCBC** method is used only to supply the *scope* and the *interface* information to the **init** method of the **cbcproxy** class, as can be seen in Figure 24.

```
/*
;-----------------------------------------------------------------;
;       initialize Component Broker                                ;
;-----------------------------------------------------------------;
*/
int  _Export acctproxy::initCBC()
  {
   return init("AccountTRScope","Account");
  }
```

Figure 24. Initialize

Here we initialized CBConnector on behalf of the transient sample.

Next, we create (or find) our Business Object from the Primary Key, as shown in Figure 25:

```
/*
;-------------------------------------------------------------------;
;       xxxxxx the Business Object - from key (xxxxxx create, find) ;
;-------------------------------------------------------------------;
*/
Account_ptr _Export acctproxy::xxxxxxBO(char *acckey)
  {
IManagedClient::IManageable_ptr mo = NULL;
AccountKey_ptr              key;
Account_ptr                 myAccount;
ByteString                  *keyString;
/*
;------- create key ------------------------------------------------;
*/
   try
     {
      key = AccountKey::_create( );
      key->accountNumber(acckey);

      keyString = key->toString( );
      CORBA::release(key);
     }
   catch( ... )
     {
      cout << "Creating AccountKey failed" << endl;
      return NULL;
     }
   cout << "Key object OK" << endl;
/*
;------- xxxxxx Managed Object --------------------------------------;
*/
   mo = xxxxxxMO(*keyString);
   if (mo == NULL)
      return NULL;
/*
;------- narrow to Business Object ----------------------------------;
*/
   try
     {
      myAccount = Account::_narrow( mo );
     }
   catch( ... )
     {
      cout << "narrow Managed Object failed" << endl;
      return NULL;
     }
   return myAccount;
  }
```

Figure 25. Create/find Business Object from key

We execute three distinct steps:

1. Create a ByteString key from the character *acckey* passed as a parameter, by using the **key** object of the **AccountKey** class.
2. Invoke the **createMO** or the **findMO** method of the **cbcproxy** class, with the *keyString* variable as parameter.
3. Narrow the Managed Object to a Business Object using the **_narrow** method of the **Account** class.

See Figure 26 for the remainder of this example:

```
/*
;-------------------------------------------------------------------;
;       create the Business Object - with Copy Helper               ;
;-------------------------------------------------------------------;
*/
Account_ptr _Export acctproxy::createBO(char *acckey, char *holder)
  {
IManagedClient::IManageable_ptr mo = NULL;
AccountCopy_ptr                copy;
Account_ptr                    myAccount;
ByteString                     *copyString;
/*
;------- create copy string -----------------------------------------;
*/
   try
     {
      copy = AccountCopy::_create( );
      copy->accountNumber(acckey);
      copy->accountHolder(holder);

      copyString = copy->toString( );
      CORBA::release(copy);
     }
   catch( ... )
     {
      cout << "Creating AccountCopy failed" << endl;
      return NULL;
     }
   cout << "Copy object OK" << endl;
/*
;------- create Managed Object --------------------------------------;
*/
   mo = createMOfromCopy(*copyString);
   if (mo == NULL)
      return NULL;
/*
;------- narrow to Business Object ----------------------------------;
*/
   try
     {
      myAccount = Account::_narrow( mo );
     }
   catch( ... )
     {
      cout << "narrow Managed Object failed" << endl;
      return NULL;
     }
   return myAccount;
  }
```

Figure 26. Create Business Object with Copy Helper

10.2.3 GUI

This section proposes a way of integrating the Component Broker Services and the Business Objects in the visual layer of the application.

The C++ and Java clients follow different architectural approaches. The C++ client follows the VisualAge paradigm of visual programming, while the Java client focuses on pure coding. The two approaches are not dependent on the programming language, but demonstrate two different solutions to CBConnector client development.

The main elements of the client application are the Visual User Interface, an object that manages Component Broker Services for a Business Object, and an object representing the Business Object itself.

In the VisualAge Family of products, you may not include any class in the visual building process. If a class is to be represented within the visual builder, it needs to be defined within the VisualAge framework as a non-visual part.

For every Business Object, we need a class wrapping the application-specific Component Broker Services, as well as a class wrapping the Business Objects. We will call these classes **CBCProxyWrapper** and **BOWrapper** respectively, but in a client implementation, they will be named according to the interface name, for instance *AccountCBCProxyWrapper* and *AccountWrapper*.

In the following sections, we give a description of the two nonvisual classes, **CBCProxyWrapper** and the **BOWrapper**, before we continue with how these objects fit in with the Visual User Interface.

10.2.3.1 The CBCProxyWrapper

Our clients need to communicate with the application-specific layer described in the previous section. The application-specific layer provides us with Component Broker Services, like initializing to the ORB, finding a home, handling transactions and creating and finding Business Objects. The class that implements these services is called **applproxy** here, where **appl** is a general name for a Business Object name abbreviation. (The Account version of this class, for example, is called **acctproxy**.) To incorporate the **applproxy** class, we create a nonvisual object called **CBCProxyWrapper** that delegates user interface events to the **applproxy** class. By introducing the extra layer of a wrapper class, you may manipulate incoming calls for the **applproxy**, like validation, type conversion, and so on., and you may also manipulate the returns given to you by the **applproxy** class.

The only attribute in the **CBCProxyClass** is the **applproxy** itself. Furthermore, we define actions we want to be visible to the user interface. Actions we define may be:

- **init** — initializes the ORB and finds a home.
- **begin** — begins a transaction.
- **commit** — commits a transaction.
- **rollback** — rolls back a transaction.
- **createBO** — takes a key as input and returns a **BOWrapper** object.
- **findBO** — takes a key as input and returns a **BOWrapper** object.

Notice that the methods **createBO** and **findBO** returns a **BOWrapper** object, and not the Business Object itself. This enables us to use the result of a **createBO** or **findBO** within the framework of the Visual Builder.

All methods need not be defined on the visual level; any methods implemented by **applproxy** may be passed directly to the **applproxy** in a user-defined method. The **applproxy** is obtained by a getter method, retrieving the instance of **applproxy** stored as an attribute of **CBCProxyWrapper**.

10.2.3.2 The BOWrapper

As mentioned earlier, we need a placeholder, or a wrapper, for the Business Objects in our client application. We create one **BOWrapper** class for each Business Object we want to interface to; in our samples, we have an **AccountWrapper** and a **CustomerWrapper**.

We implement such a wrapper for the same reasons that we implement a **CBCProxyWrapper**; so that the attributes and actions of a Business Object will be accessible in the VisualAge framework.

The **BOWrapper** acts like a copy of the Business Object itself. It has the same attributes as the Business Object (or a subset if you do not want or need all the attributes in your visual programming), and an attribute that holds a reference to the real Business Object. The **BOWrapper** implements the actions we want to access in the Visual Builder. These actions basically execute the corresponding method on the Business Object, adding perhaps some validation or type conversions.

The attributes of the **BOWrapper** need to be set to the attributes of the Business Objects by the client application. When the methods **createBO** and **findBO** of the **CBCProxyWrapper** return a **BusinessObjectWrapper** object, it instantiates the attributes of the **BOWrapper** to be that of the Business Object. Furthermore, when changes are made to the **BOWrapper**, like calling the **changeBalance** method on an Account for instance, the **getBalance** method is called to update balance of the **BOWrapper** object.

10.2.3.3 Putting it all together

In an application, you need one instance of **CBCProxyWrapper** for each Business Object used in the interface. You also need a variable holder of the type **BOWrapper**. The number of **BOWrapper** variables depend on how many instances of a Business Object you need at a time for this client.

As an example, we create a visual part that lets you create and find an account and update information on accounts.

Figure 27 illustrates the VisualAge programming environment and the connection between the GUI and the Wrapper Objects.

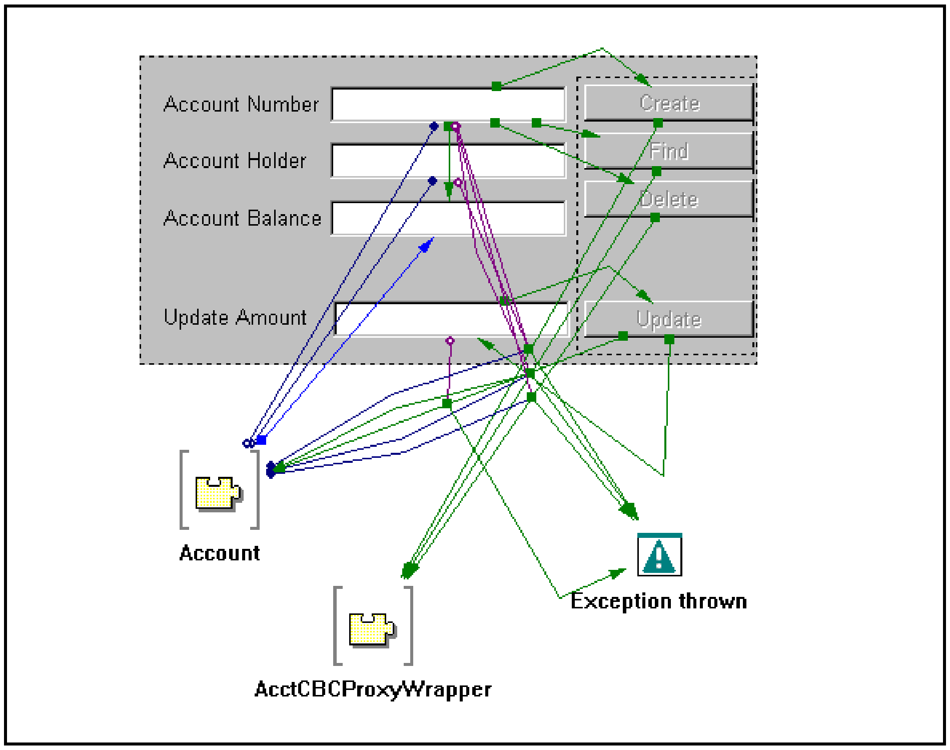


Figure 27. Visual programming example for the account sample

First, we create an **AccountWrapper**. The **AccountWrapper** has the following attributes:

- `account` — holds the pointer of the Business Object **Account**.
- `accountNumber` — represents the `accountNumber` of **Account**.
- `accountHolder` — represents the `accountHolder` of **Account**.
- `balance` — represents the `balance` of **Account**.

We also define the following action for **AccountWrapper**:

- `changeBalance(anAmount)` — executes the **changeBalance** method on the Business Object **Account**

Second, we create an **AccountCBCProxyWrapper**. The **AccountCBCProxyWrapper** has the following attributes:

- `cbcProxy` — holds the pointer of the **acctproxy** class

- We also define the following actions for **AccountCBCProxyWrapper**:
- **init** — initializes the ORB and finds a home.
- **createBO** — takes a key as input and returns an **Accountrapper** object.
- **findBO** — takes a key as input and returns an **AccountWrapper** object.
- **deleteBO** — takes a key as input and returns an empty **AccountWrapper** object.

The actions need to be defined within the generated skeleton of user-defined methods.

We add the **AccountCBCProxy** part to the visual part. This means that an instance of **AccountCBCProxy** will be created when the visual part is shown. In addition, we add a variable holder for the **AccountWrapper**. This is a variable because it will hold several instances of the Account Business Object, one at a time.

To initialize the global CBConnector environment and find a home, we call the **init** method of the **AccoutnCBCProxyWrapper** when the window is shown. Then we call the **createBO** and **findBO** method of the **AccountCBCProxyWrapper** by clicking the **create** and **find** button. respectively. These methods take the entry field values as incoming parameters and return an instance of **AccountWrapper**. This **AccountWrapper** has already copied the values of its Business Object attributes into its own corresponding attributes. It also has a pointer to the actual Business Object, **Account**, in an attribute. This **AccountWrapper** object is placed in the variable holder for the **Account** Business Object. The attributes of the **AccountWrapper** are connected to the fields of the GUI, and methods may be called on the **AccountWrapper** from the GUI.

When the Update button is clicked, the action **changeBalance** of the **AccountWrapper** is called, passing the GUI's entry field value of the amount. The **AccountWrapper** converts the entry field value from text to a numeric value, and calls the **changeBalance** method of **Account** with the numeric value as an incoming parameter. **Account** is stored as an attribute within the **AccountWrapper**. Then it calls the **getBalance** method of **Account** and updates its own balance attribute with the new value. This value is displayed in the GUI’s balance entry field.

10.3 Java implementation

In the Java implementation, we stick with the three-layer approach as described in the chapter overview. The bottom layer is implemented as a class called **CBCBase**, providing a lot of static methods, while the general layer is implemented as the **CBProxy** class. The **CBProxy** class invokes helper methods in the **CBCBase** class to perform its functions. The application layer is implemented as a class **AccountProxy**, which is a subclass of **CBProxy**. Of course, this application-specific class varies from Business Object to Business Object, but because most of our samples use the **Account** class, we elected to describe the **AccountProxy** class.

In the following section, we describe the three classes that implement our client object model.

10.3.1 The Proxy.CBCBase class

The **CBCBase** class stores references to the ORB and Name Service, and provides utility methods. Because all attributes and methods are static in this class, you must first initialize it using **init**. The **init** method initializes the ORB and Name Service based on the bootstrap host and port. You may also supply a listbox as a parameter if you want to trace messages in a GUI. The code snippet in Figure 28 shows the implementation of the **init** method.

```
/**
 * This initializes the ORB and the root naming context
 * @return boolean
 * @param hostname java.lang.String
 * @param port java.lang.String
 * @param tracingListbox java.awt.List
 */
public static boolean init (String hostname, String port,   java.awt.List tracingListbox ) {

  if (!initialized) {

   try {

    if (tracingListbox != null) setTracingListbox(tracingListbox);

    if (hostname != null) setHostname(hostname);
    else {
     traceIt("The Bootstrap hostname is not set.");
     throw new NullPointerException();
    }

    if (port != null) setPort(port);
    else setPort("900");   // set to default

    /*
     *------ First resolve the orb -------------------------------------
     */
    orb = resolveOrb();
    if (orb == null) throw new NullPointerException();

    /*
     *------- Resolve to the root naming context. This will establish ----
     *------ a remote object reference to root of the name space. -------
     */
    nameService = resolveNameService();
    if (nameService == null) throw new NullPointerException();
    traceIt("CBProxy initialized.");

   }
   catch (Throwable e) {
     traceIt("Unable to initialize CBProxy.");
     handleException(e);
   }
  }

  return initialized;
}
```

Figure 28. Initializing the CBCBase class

The **init** method returns `true` if the ORB and Name Service are resolved successfully. The methods **resolveOrb** and **resolveNameService** are called here and shown later in this section.

The following helper methods are implemented in **CBCBase**.

- resolveOrb
- resolveNameService
- resolveDefaultFactoryFinder
- resolveFactoryFinder
- resolveFactory
- resolveHome
- resolveHomeViaFindFactories
- resolveQueryableHome
- resolveQueryableHomeViaFindFactories
- getCurrent
- traceIt

Not all of these methods are used in the samples. Here, we only show the implementations of the most common methods used in the samples. The others will be introduced, as needed, in the samples.

An example of resolving the ORB is shown in Figure 29.

```
/**
 * resolves the orb
 * @return com.ibm.CORBA.iiop.ORB
 */
private static com.ibm.CORBA.iiop.ORB resolveOrb ( ) {

  com.ibm.CORBA.iiop.ORB tmpOrb = null;
  String[] args = null;
  java.util.Properties props = null;

  try {
     // create a property list using the hostname and port parameters
     props = new java.util.Properties();
     props.put("org.omg.CORBA.ORBClass", "com.ibm.CORBA.iiop.ORB");
     props.put("com.ibm.CORBA.BootstrapHost", getHostname());
     props.put("com.ibm.CORBA.BootstrapPort", getPort());

     // calling ORB.init
     tmpOrb = (com.ibm.CORBA.iiop.ORB) ORB.init (args, props);
   }
   catch (Throwable e) {
    return null;
   }

   return tmpOrb;
}
```

Figure 29. Resolving the ORB

An example of resolving the Name service is shown in Figure 30.

```
/**
 * resolves the Name Service
 * @return COM.ibm.IExtendedNaming.NamingContext
 */
private static NamingContext resolveNameService ( ) {

  NamingContext tmpNameService = null;
  org.omg.CORBA.Object obj;

  try {
     obj = orb.resolve_initial_references("NameService");
     tmpNameService = NamingContextHelper.narrow(obj);
  }
  catch (Exception e) {
     return null;
  }

  return tmpNameService;
}
```

Figure 30. Resolving the Name Service

An example of resolving to a Factory Finder is shown in Figure 31.

```
/**
 * resolves the factoryFinder for a specific scope
 * @return factoryFinder (com.ibm.IExtendedLifeCycle.FactoryFinder)
 */
public static FactoryFinder resolveFactoryFinder(String scope) {

  org.omg.CORBA.Object obj = null;
  FactoryFinder factoryFinder = null;

  try {
   obj = nameService.resolve_with_string( "host/resources/factory-finders/" + scope );
   factoryFinder = FactoryFinderHelper.narrow(obj);
  }
  catch (Exception e) {
   return null;
  }

  return factoryFinder;
}
```

Figure 31. Resolving to a Factory Finder

This method allows you find a Factory Finder for a specific scope.

An example of resolving to a home is shown in Figure 32.

```
/**
 * resolves the home of a specific object
 * @return COM.ibm.IManagedClient.IHome
 */
public static IHome resolveHome(FactoryFinder factoryFinder, String objectInterface) {

  IHome home = null;
  org.omg.CORBA.Object obj = null;

   try {
     obj = factoryFinder.find_factory_from_string( objectInterface );
     home = IHomeHelper.narrow(obj);
   }
   catch (Exception e) {
     return null;
   }

   return home;
}
```

Figure 32. Resolving to a home

This method allows you to find the home of a specific *type* of Managed Objects based on that object's interface, using a Factory Finder that has already been found for a certain scope. For example, to find the home of **Account** Managed Objects, you would invoke this method using `resolveHome(factoryFinder, "Account.object interface")`.

An example of creating a current Transactional Object is shown in Figure 33.

```
/**
 *  Get the current transactional object
 */
public static org.omg.CosTransactions.Current getCurrent() {

  try {
    org.omg.CORBA.Current orbCurrent;
    orbCurrent = orb.get_current("CosTransactions::current");
    org.omg.CosTransactions.Current current =
          org.omg.CosTransactions.CurrentHelper.narrow(orbCurrent);

    if (current == null)
    throw new NullPointerException();
    else
    return current;
  }
  catch (Throwable e) {
    return null;
  }
}
```

Figure 33. Creating a current Transactional Object

This method shows you how to get a current Transactional Object from the ORB. This object is needed to start a unit of work within the context of a transaction.

10.3.2 The Proxy.CBCProxy class

In the **CBProxy** class, we focus on creation and finding of Managed Objects, as well as the transactional services needed in the various samples. This class stores the Factory Finder and home of the objects based on the Factory Finder scope and object interface it is initialized with when you invoke the **init** method. See Figure 34.

```
/*
 *  initialize the object with scope and object's interface
 *  so that a home may be found
*/
public void init(String scope, String objectInterface) {

 try {
  /*
  ------- resolve to the factory finder ------------------------------
  */
  factoryFinder = CBCBase.resolveFactoryFinder(scope);

  if (factoryFinder == null) {
    throw new NullPointerException();
  }

  /*
  ------- resolve to the home ----------------------------------------
  */
  home = CBCBase.resolveHome(factoryFinder, objectInterface);

  if (home == null) {
    throw new NullPointerException();
    }
 }
 catch (Throwable e) {
   CBCBase.traceIt("unable to initialize CBProxy for object of interface \""
    + objectInterface + "\" in the scope " + scope);
 }
}
```

Figure 34. Initializing the CBProxy class

If either the Factory Finder or home cannot be resolved, a *NullPointerException* is thrown.

The following methods show how to create a Managed Object either from a Primary Key or a Copy Helper, and how to find a Managed Object with a Primary Key. These methods return a reference to a Managed Object. It is the job of the caller of these methods to *narrow* (cast) to the specific *type* of Managed Object; for example, an **Account** object.

You can see these methods displayed in Figure 35, Figure 36, and Figure 37.

```
/*
 *  create Managed Object (MO) from Primary Key String
 *
 */
org.omg.CORBA.Object createMO(byte[] keyString)
     throws com.ibm.IManagedClient.IInvalidKey,
              com.ibm.IManagedClient.IDuplicateKey,
              NullPointerException {

  org.omg.CORBA.Object o = null;

  try {
    o = home.createFromPrimaryKeyString(keyString);
  }
  catch (IInvalidKey e) {
    throw e;
  }
  catch (com.ibm.IManagedClient.IDuplicateKey e) {
    throw e;
  }
  catch (Throwable e) {
    o = null;
  }

  if (o == null) {
    throw new NullPointerException();
  }

  return o;
}
```

Figure 35. Creating a Managed Object from a Primary Key string

```
/*
 *   create Managed Object (MO) from Copy String
 */
org.omg.CORBA.Object createMOFromCopy(byte[] copyString)
            throws com.ibm.IManagedClient.IInvalidCopy,
                     com.ibm.IManagedClient.IDuplicateKey,
                     NullPointerException {

  org.omg.CORBA.Object o = null;

  try {
    o = home.createFromCopyString(copyString);
  }
  catch (IInvalidCopy e) {
    throw e;
  }
  catch (com.ibm.IManagedClient.IDuplicateKey e) {
    throw e;
  }
  catch (Throwable e) {
    o = null;
  }

  if (o == null) {
    throw new NullPointerException();
  }

  return o;
}
```

Figure 36. Creating a Managed Object from a Copy string

```
/*
 *  find Managed Object (MO)
 */
org.omg.CORBA.Object findMOFromKey(byte[] keyString)
     throws com.ibm.IManagedClient.IInvalidKey,
            com.ibm.IManagedClient.INoObjectWKey,
            NullPointerException {

  org.omg.CORBA.Object o = null;

  try {
    o = home.findByPrimaryKeyString(keyString);
  }
  catch (IInvalidKey e) {
    throw e;
  }
  catch (com.ibm.IManagedClient.INoObjectWKey e) {
    throw e;
  }
  catch (Throwable e) {
    o = null;
  }

  if (o == null) {
    throw new NullPointerException();
  }

  return o;
}
```

Figure 37. Finding a Managed Object by a Primary Key string

Finally, the transaction services are implemented in this class. They are:

- beginTX
- commitTX
- rollbackTX

See the example in Figure 38.

```
/*
 *  begin transaction
 */
public boolean beginTX() {
org.omg.CORBA.Object obj = orb.resolve_initial_references("TransactionCurrent");
org.omg.CosTransactions.Current currentTransaction =
org.omg.CosTransactions.CurrentHelper.narrow(obj);

  try {
    currentTransaction.begin();
    return true;
  }
  catch (java.lang.Throwable e) {
    return false;
  }
}
```

Figure 38. Beginning a transaction

To begin a transaction, you must first get a current Transactional Object from the ORB. This method gets a Transactional Object from the ORB if one doesn't exist and begins a transaction. The current Transactional Object is stored as a member of this class.

See the examples shown in Figure 39 and Figure 40.

```
/*
 *  commit transaction
 */
public boolean commitTX() {

  if (currentTransaction == null) {
    CBCBase.traceIt("no trasaction exists to commit");
    return false;
  }

  try {
    currentTransaction.commit(true);
    return true;
   }
  catch (java.lang.Throwable e) {
    return false;
  }
}
```

Figure 39. Committing a transaction

```
/*
 *  rollback transaction
 */
public boolean rollbackTX() {

  if (currentTransaction == null) {
    CBCBase.traceIt("no transaction exists to rollback");
    return false;
  }

  try {
    currentTransaction.rollback();
    return true;
  }
  catch (java.lang.Throwable e) {
    return false;
  }
}
```

Figure 40. Rolling back a transaction

10.3.3 The Proxy.AccountProxy class

This class is a subclass of **CBProxy** and is responsible for creating and finding **Account** objects by narrowing the Managed Objects returned by the corresponding methods of the **CBProxy** super class.

Refer to the examples shown below in Figure 41, Figure 42, and Figure 43.

```
/**
 * create an Account Managed Object from PrimaryKey
 * @return Account
 */
public Account createAccountFromKey (String accountNumber) {
Account account = null;
try {
      // create Primary Key
    AccountKey theKey = AccountKeyHelper._create();
    theKey.accountNumber( accountNumber );
    byte[] theKeyString = theKey._toString();
 org.omg.CORBA.Object o = null;
    try {
      o = createMO(theKeyString);
    }
    catch (IInvalidKey e) {
      CBCBase.traceIt("Invalid key..." + e);
      o = null;
    }
    catch (com.ibm.IManagedClient.IDuplicateKey e) {
      CBCBase.traceIt("Account number already exists :  " + e);
      o = null;
    }
    catch (NullPointerException e) {
      o = null;
    }
if (o == null) {
      throw new NullPointerException();
    }
// narrow to Account
    account = AccountHelper.narrow(o);

    if (account == null) {
      throw new NullPointerException();
    }
  }
  catch(Throwable e) {
    CBCBase.traceIt("Unable to create Account");
    account = null;
  }
  return account;
}
```

Figure 41. Creating an Account from a Primary Key

```
/**
 * create an Account Managed Object from PrimaryKey
 * @return Account
 */
public Account createAccountFromCopy (String accountNumber, String accountHolder) {

  Account account = null;

  try {
    // create Copy Helper
    AccountCopy theCopy = AccountCopyHelper._create();
    theCopy.accountNumber( accountNumber );
    theCopy.accountHolder( accountHolder );
    byte[] theCopyString = theCopy._toString();

    org.omg.CORBA.Object o = null;
    try {
      o = createMOFromCopy(theCopyString);
    }
    catch (IInvalidCopy e) {
      CBCBase.traceIt("Invalid Copy..." + e);
      o = null;
    }
    catch (IDuplicateKey e) {
      CBCBase.traceIt("Account number already exists :  " + e);
      o = null;
    }
    catch (NullPointerException e) {
      o = null;
    }

    if (o == null) {
      CBCBase.traceIt("Unable to create an Managed Object");
      throw new NullPointerException();
    }

    // narrow to Account
    account = AccountHelper.narrow(o);

    if (account == null) {
      CBCBase.traceIt("Unable to narrow to Account");
      throw new NullPointerException();
    }
  }
  catch(Throwable e) {
    account = null;
  }

  return account;
}
```

Figure 42. Creating an Account using a Copy Helper

```
/**
 * find an Account Managed Object from PrimaryKey
 * @return Account
 */
public Account findAccountByKey (String accountNumber)  {

  Account account = null;

  try {
    // create Primary Key
    AccountKey theKey = AccountKeyHelper._create();
    theKey.accountNumber( accountNumber );
    byte[] theKeyString = theKey._toString();

    org.omg.CORBA.Object o = null;
    try {
      o = findMO(theKeyString);
    }
    catch (IInvalidKey e) {
      CBCBase.traceIt("Invalid key..." + e);
      o = null;
    }
    catch (com.ibm.IManagedClient.INoObjectWKey e) {
      CBCBase.traceIt("No Account with this number exists : " + e);
      o = null;
    }
    catch (NullPointerException e) {
      o = null;
    }

    if (o == null) {
      CBCBase.traceIt("Unable to find a Managed Object");
      throw new NullPointerException();
    }

    // narrow to Account
    accountMO = AccountHelper.narrow(o);

    if (accountMO == null) {
      CBCBase.traceIt("Unable to narrow to Account");
      throw new NullPointerException();
    }
  }
  catch(Throwable e) {
    accountMO = null;
  }

  return accountMO;
}
```

Figure 43. Finding an Account using a Copy Helper

10.3.4 GUI

Throughout this book, each Component Broker sample builds upon what you have created and learned in the previous chapters. So do the Java client programs associated with the samples. This section shows you how we developed a Java applet to function as a simple Component Broker client program using our client model described in the previous sections. This applet is basically the client for the first sample in Chapter 11, “Transient Object sample” on page 161.

10.3.4.1 The applet

We create a simple Java applet which performs essentially three main tasks:

- Initializes the **CBCBase** class (ORB and Name Service) with the `bootstrap hostname` and `port number` parameters passed to the applet. Also, we have a trace window we use to follow the steps of our program; so we also initialize the **CBCBase** class with this window.
- Initializes an **AccountProxy** (Factory Finder and home) with the `location scope` of where the **Account** home resides.
- Initializes an **AccountView** with the **AccountProxy**. The **AccountView** is a visual part (JavaBean) that interacts with the user to create, find, update, and remove **Accounts** through the **AccountProxy**.

10.3.4.2 The AccountView visual part

The **AccountView** visual part, or JavaBean, provides and handles the interaction between the user and **Account** objects residing on the Component Broker server.

This user interface has fields for an `account number`, `account holder` and a new `balance`. To create an **Account**, you type in an account number and holder, then press the *Create* button. Very simple. Similarly, you can update balances and find and remove **Accounts**. Each button corresponds to a direct call to either the **AccountProxy**, which creates and finds Managed Objects or to the **Account** Managed Objects themselves.

10.3.4.3 The TraceLogView visual part

The **TraceLog** is a graphical listbox that logs messages provided for guidance (or debugging) through execution of the program. For our samples, we log messages about each step taken in the process of the communication between a Component Broker client application and server object.

You create a log message by calling the **traceIt** method of **CBCBase**. This will print the message to standard out and to the **TraceLogView**. If you would like to print log messages to a file, you could simply overwrite this method.

10.4 ActiveX implementation

The ActiveX versions of all samples in this book are built using Microsoft Visual Basic. However, it is also possible to use Microsoft Visual C&plus.&plus. instead of Basic.

Due to time limitations, we couldn't implement the ActiveX framework in the same manner as for the C++ and Java clients. In this chapter, we therefore only discuss the programming steps common for all ActiveX clients.

In general, the code to access the Business Objects through Microsoft Visual Basic looks very similar to the code of the other programming languages. In fact, you will soon notice only a few differences coming from the specialities of Microsoft Visual Basic.

We have tried to make the code of the samples as plain as possible in order to see the essence of the program (such as how to use the Component Broker features through Microsoft Visual Basic). A real-world application would look different since you'd probably have to check for errors and implement a user interface.

The following sections show the steps you have to do in your client code in order to connect to a server and for invoking Business Object methods.

Let's start with the ActiveX interface to the ORB and how to install it.

Note: The following code snippets intentionally leave out some details (such as the general exception-handling code).

10.4.1 Initializing the ORB

The following code snippets show what you have to do to get your client application set up:

```
Dim anObject As Object
Dim orb As Object
Dim namingService As Object
 ...

' get the ORB object
Set orb = CreateObject("CORBA.ORB")
Set namingService = CreateObject("IDL:iextendednaming.NamingContext")
 ...

' get the NameService
Set anObject = orb.ResolveInitialReference("NameService")
' narrow the NameService
namingService.narrow anObject
```

10.4.2 Finding a Factory Finder

Factory Finders vary according to the scenario we are dealing with. We want to find a Factory Finder that is instantiated with a Location Object defined to establish a specific scope (scope="AccountTRScope"). Please look for the theory behind the following code in Chapter 18, “LifeCycle Service” on page 287.

```
Dim factoryFinder As Object
...
Set factoryFinder = CreateObject("IDL:iextendedlifecycle.FactoryFinder")
...
scope="AccountTRScope"
Set anObject = namingService.resolve_with_string("host/resources/factory-fin
ders/"+scope)
factoryFinder.narrow anObject
```

10.4.3 Finding a home

We have almost made it. Having resolved to the correct Factory Finder using the Naming Service, the only preparatory step that still remains to be accomplished before actually doing anything with our Business Objects is to find their home. Add a string `object interface` to the complete interface name as shown.

```
Dim myHome As Object
...
Set myHome = CreateObject("IDL:imanagedclient.IHome")
...
interfaceName="Account.object interface"
Set anObject = factoryFinder.find_factory_from_string(interfaceName)
myHome.narrow anObject
```

10.4.4 Creating a Primary Key

In order to operate on objects in the home, we have to set up a Primary Key. Here, we use an accountNumber field as our Primary Key and create it as follows:

```
Dim keyVariant As Variant
Dim keyString() As Integer
Dim i As Integer
...
' creating the key
appName="Account"
Set key = CreateObject("IDL:" + appName + "Key")
' create the client side object
Set theObject = CreateObject("IDL:" + appName)

' take the input from the Textbox
key.accountNumber = key_Id_String
keyVariant = key.toString()

ReDim keyString(UBound(keyVariant))
For i = 0 To UBound(keyVariant)
keyString(i) = keyVariant(i)
Next i
```

10.4.5 Creating the Business Object from Key

In the next snippet, we use the Primary Key to create an **Account** Object. Note the error-checking code.

```
Set anObject = myHome.createFromPrimaryKeyString(keyString, e)
If e.EX_majorCode() <> 0 Then
' 0 means OK, 1=System_exception, 2=User_exception
' e.EX_repositoryID() contains the error message
...
End If

' now narrow the generic anObject to the pingObject
theObject.narrow anObject
```

To find an object, you replace the method name, **createFromPrimaryKeyString**, in the above example by **findByPrimaryKeyString**.

```
Set anObject = myHome.findByPrimaryKeyString(keyString, e)
```

10.4.6 Invoking operations on Business Objects

The next snippet shows how to invoke the **changeBalance** and **getBalance()** methods. It looks easy. Indeed it is.

```
theObject.changeBalance 100
x = theObject.getBalance()
```

10.4.7 Cleaning up the client environment

You may have to do some clean-up operations from time to time. As an example: When you are done with any objects, you should release the references to them. You do this by setting the object to **nothing**. This tells the server to decrease the reference count, an action that is vital for memory management. However, objects are freed automatically by Microsoft Visual Basic, and you need not to set an object to **nothing**. It will happen automatically when leaving the function or program.

Releasing an object is quite different from actually removing it. The latter has the expected effect on the server side, for example, to cause deletion of the corresponding row in the SQL table within the back-end database.

```
 ' cleanup
' Release Account Proxy
Set theObject=nothing
```

After implementing the previous steps along with your client-specific code (GUI and so forth), you are ready to compile and run your client. In the following samples, we refer back to this chapter and only show the specifics of the ActiveX client in the samples.

10.5 What did you learn?

This chapter described a possible approach to client development.
We isolated the CBConnector client programming model from the client application code and the GUI. We described the implementation in C++ and Java.

Please note that our sample approach is not fully implemented and complete.

Chapter 11. Transient Object sample

After testing your environment and reviewing a general description of the development process, you can now start to use the Component Broker Connector development environment.

You can find the Object Builder model, code, and client applications for this sample on the CD for this book in the directory *\CBConnector\11-Transient-Object*.

11.1 What you will learn

The application we develop in this chapter is a very simple client/server application. It uses the **Account** scenario described in Chapter 8, “The Account scenario” on page 101. You will create a server application for transient Account Managed Objects, and then you will see how to develop client applications which instantiate these Account Objects using the Primary Key and the Copy Helper. These Transient Objects will exist in memory on the server until the server process is stopped. You will implement the Business Object for the server application in both C++ and Java. You will also learn how to import and export XML files, which contain definitions of your object model, to and from the Object Builder.

11.2 CBConnector componentry

In this section, we briefly describe the most important CBConnector components involved in this sample. This gives you a high-level picture of the framework involved in a simple transient server application and the client programs you will implement in this chapter.

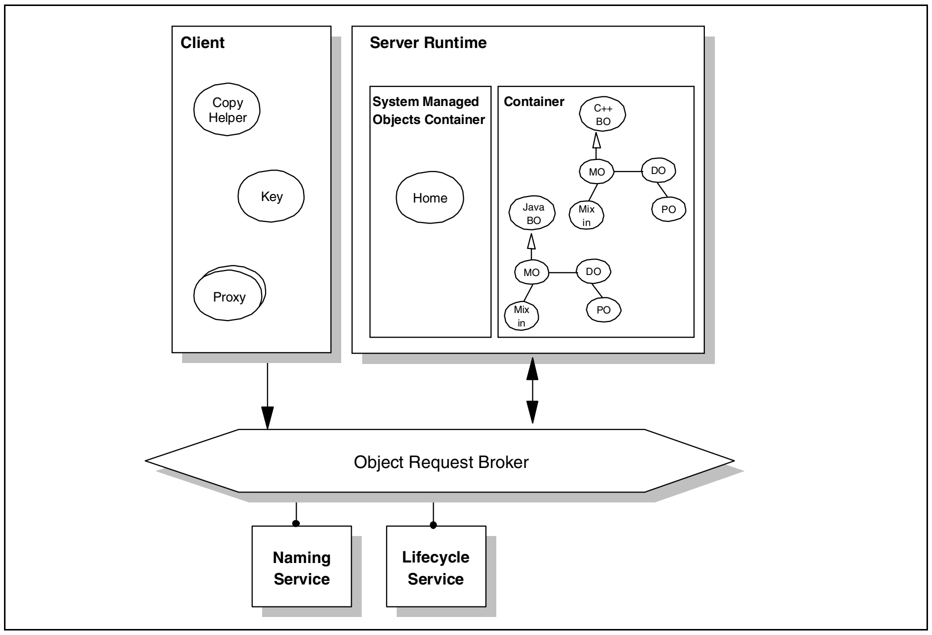


Figure 44. Transient sample components

Figure 44 shows the main components used in our simple transient scenario.

The Component Broker features highlighted in this chapter include:

- Transient Data Object
- Java Business Object
- Primary Key
- Copy Helper

11.2.1 Transient Data Object

The essential state of a Business Object (BO) is kept in the Data Object (DO) and can additionally be cached in the BO. When the implementation of the DO is transient, it exists in memory on the server only for the lifetime of the server process in which it runs, or until the Managed Object for that DO is removed by a client program by calling the **remove()** method on that Managed Object.

11.2.2 Java Business Object

Most of the Component Broker server infrastructure is implemented in C++; so when you create a C++ BO, method calls made on that Business Object are straightforward method invocations. But when you have a Java BO, this implies that calls made to that BO will have to cross the C++/Java language boundary. Component Broker provides what is called an Interlanguage Object Model (IOM) capability, which handles this cross-language communication in the same server process.

Figure 45 gives a high-level overview of the case of clients invoking methods on Business Objects implemented in C++.

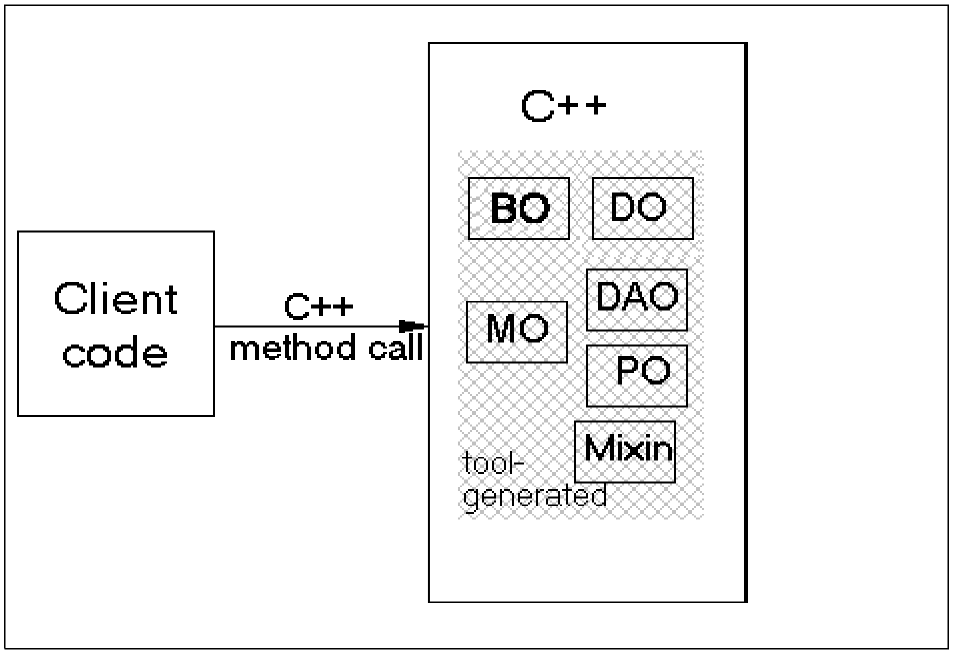


Figure 45. C++ Method invocation model for business objects

Figure 46 illustrates how having a Java BO changes the situation.

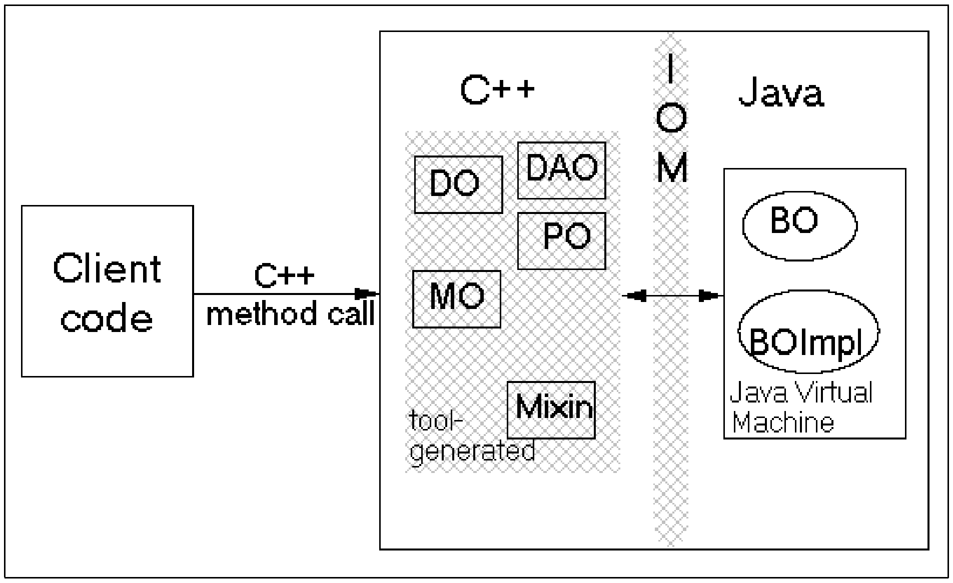


Figure 46. Java method invocation model for business objects

This is only a quick overview of the differences in the componentry involved in implementing you Business Objects in C++ and Java. We explain a bit more about IOM in Chapter 23, “IOM with two datastores” on page 373. You can find a more in-depth explanation of language interoperability in the redbook *IBM Component Broker Connector Overview*, SG24-2022.

11.2.3 Primary Key

A Primary Key is a helper object that contains data about a Managed Object which uniquely identifies that object from the other Managed Objects in a home.

The Primary Key is used to find and create Managed Objects. A client can find a Managed Object by first creating a Primary Key with the appropriate attributes, then calling the **findByPrimaryKeyString()** method on the home, passing it the stringified Primary Key as a parameter. A client can also create a Managed Object by calling the method **createFromPrimaryKeyString()** on the home, similarly passing it the stringified Primary Key.

When an object is created on the server, it will only be initialized with the attributes that were set in the Primary Key. Once a Managed Object is found or created, the client must make calls across the ORB to get or set any other attributes for that object that were not specified in the Primary Key.

11.2.4 Copy Helper

The Copy Helper is similar to the Primary Key in its use, but provides clients with a way to optimize performance and minimize the number of trips across the ORB during the creation of Managed Objects. Whereas the Primary Key contains only the attributes that make an object unique, the Copy Helper contains all the attributes necessary to fully initialize a Managed Object on the server.

A client can create and set all the attributes of the Copy Helper object in its own language and process space, then pass a string version of this local object as an argument to the **createFromCopyString()** method of the home. In this case, a client can create a fully initialized Managed Object in one trip across the ORB.

11.3 C++ BO Server

This section takes you through all the steps you need to perform to build, package, and install a Transient Object server application. We will write the implementation of the Business Object for this server application in C++. In the following section, we will walk you through building the same server application using Java as the implementation language for the Business Object.

Note

All code snippets have enumerated lines for easier orientation in the code. For your convenience, all C++ code snippets used in this sample are on the CD located in *\CBConnector\11-Transient-Object\CppBOServer\code_snippets.cpp*

11.3.1 Build

Implementing the server application consists of two major steps:

1. Specify the interface and implementation of the Business Object, Data Object, and the Key and Copy Helpers.
2. Generate and build the application.

Through the Object Builder walkthrough, we provide you with the correct parameter settings for the SmartGuides. If no suggestions are made by us for a SmartGuide page, use the default settings.

We strongly recommend that you open Help for each field of the SmartGuide pages in order to understand more of the dialectic behind the input fields.

11.3.1.1 Create the Model

1. Create a new Object Builder model in a separate directory.
2. From **User-Defined Business Objects**, add a new file, **Account**, using the following parameters:

Parameter	Value
Name	Account
Constructs: Exception	NotEnough

3. For this file, add an interface, **Account**, with the following parameters:

Parameter	Value
Name	Account
Constructs: Exception	NotEnough

   Add attributes to the Account Interface as described below:

Attribute	Type	Implementation
accountNumber	string<10>	Public
accountHolder	string<30>	Public
balance	double	Private

   Add methods, **changeBalance** and **getBalance** with the following:

Method	Return Type	Parameter	Exception
changeBalance	void	anAmount double In	NotEnough
getBalance	double		

4. For the newly created Account Interface, add a Key Helper using the `accountNumber` as the Primary Key.

5. Add a Copy Helper with the attributes:
   - `accountNumber`
   - `accountHolder`

   The attribute `balance` is not added because it is a private attribute of the object.

6. Now add an implementation of the BO. Use the following parameters:

Parameter	Value
Pattern for handling state data	Caching
Object Reference	Use lazy evaluation
Data Object Interface	Create a new one now
Implementation Inheritance	IManagedClient IManagedClient::IManageable
Implementation Language	C + +
Key Selection	AccountKey
Handle	none
Attributes to Override	none
Methods to Override	none
Data Object File Name	AccountDO
Data Object Interface Name	AccountDO
State data	all

7. Implement the **changeBalance** method. The implementation of the BO methods is the only code you have to write for the server application. For this application, we implement the methods in C++. Include the following logic:

```
1. if ( ( iBalance + anAmount ) < 0 )
2. throw new NotEnough( );
3. Dim namingService As Object
4. iBalance += anAmount;
```

8. Implement the **getBalance** method:

```
1. return iBalance;
```

9. For the AccountDO, add a Data Object implementation with the following parameters:

Parameter	Value
Environment	BOIM with any Key
Form of Persistent Behavior and Implementation	Transient
Handle for Storing Pointers	Home name and Key
Key helper	AccountKey
Copy helper	AccountCopy

The Data Object and its implementation are now displayed under the **User-Defined Data Objects** folder. The AccountDO in this folder is simply another view of the AccountDO displayed under the AccountBO in the **User-Defined Business Objects** folder.

10. Add a Managed Object to the **AccountBO**. Accept all default settings.
11. Generate the code for all files from **User-Defined Business Objects**.

11.3.1.2 Build the Server Application

In the **Build Configuration** folder, execute the following steps:

1. Add a client DLL named AccountTRC with the following parameters:

Parameter	Value
Name	AccountTRC
Client Source Files	all
Libraries to Link With	none

2. Add a server DLL named AccountTRS with the following parameters as shown below:

Parameter	Value
Name	AccountTRS
Server Source Files	all
Libraries to Link With	AccountTRC (the client library)

3. Generate the makefiles for all targets.
4. Build all targets of the server application.

11.3.2 Package

In this section, we describe how to package the transient Account application. This includes the following steps:

1. Definition of the application using the Object Builder.
2. Creation of an installation image using InstallShield.

11.3.2.1 Define the Application

1. From the **Application Configuration** folder, add an Application Family named **AccountTRAppFam**.
2. For this application family, add a server application named **AccountTRAppS** with parameters as follows:

Parameter	**Value**
Application Name	AccountTRApps

3. Add a Managed Object for the server application with the following parameters:

Parameter	**Value**
Managed Object	AccountMO AccountMO
Primary Key	AccountKey AccountKey
Copy Helper	AccountCopy AccountCopy
Data Object Implementation	AccountDOImpl AccountDOImpl
Container	CachedTransientObjects
Default Home	Default Home
Home Name	BOIMHomeOfRegHomes
Name in Factory Finding Service Registry	AccountMOFactory
Name in Naming Service Registry	AccountMOHome

11.3.2.2 Create an installation image

1. From the AccountTRAppFam, generate the installation scripts. This step also creates the System Management data in the form of a DDL file.

11.3.3 Install

To install and configure the server application, follow these steps:

1. Install the server application on your server machine (see 7.1, “Install the application” on page 89).
2. Configure an **AccountTRServer** in a new Management Zone, **AccountTR Management Zone**. Use a new configuration, **AccountTR Configuration**, for this server.
3. Activate the configuration.

11.4 Java BO server

In the previous section, you created a server application with a Business Object implemented in C++. This section takes you through all the steps you need to perform to build, package, and install a Transient Object server application which has a Business Object implemented in Java.

Note

All code snippets have enumerated lines for easier orientation in the code. For your convenience, all Java code snippets used in this sample are on the CD located in *\CBConnector\11-Transient-Object\JavaBOServer\code_snippets.java*

11.4.1 Build

Implementing the server application consists of two major steps:

1. Specify the interface and implementation of the Business Object, Data Object, and the Key and Copy Helpers.
2. Generate and build the application.

If you have already created the model for the transient sample in the previous section, where you implement the BO in C++, then you may consider skipping 11.4.1.1, “Create the model” on page 171 and beginning your reading of this section with 11.4.1.2, “Create the model with XML” on page 174. The only difference between the model with the C++ BO and the Java BO is that instead of specifying C++ as your implementation language, you specify Java in the SmartGuide for the BO implementation. This means that you will implement the methods for your BO in Java, rather than C++.

If you have not created the model for the C++ BO server, we suggest you look at our C++ BO server model on the CD in the directory *\CBConnector\11-Transient-Object\CppBOServer*. Then work through 11.4.1.2, “Create the model with XML” on page 174, so you can become familiar with importing and exporting XML into the Object Builder.

Through the Object Builder walkthrough, we provide you with the correct parameter settings for the SmartGuides. If no suggestions are made by us for a SmartGuide page, use the default settings.

11.4.1.1 Create the model

1. Create a new Object Builder model in a separate directory.
2. From **User-Defined Business Objects**, add a new file, **Account**, using the following parameters:

Parameter	Value
Name	Account
Constructs: Exception	NotEnough

3. For this file, add an interface, **Account**, with the following parameters:

Parameter	Value
Name	Account
Constructs: Exception	NotEnough

   Add the attributes to the Account Interface as described below:

Attribute	Type	Implementation
accountNumber	string<10>	Public
accountHolder	string<30>	Public
balance	double	Private

   Add methods, **changeBalance** and **getBalance** with the following definitions:

Method	Return Type	Parameter	Exception
changeBalance	void	anAmount double in	NotEnough
getBalance	double		

4. For the newly created Account Interface, add a Key helper using the `accountNumber` as the Primary Key.
5. Add a Copy Helper with the attributes:

•`accountNumber`

•`accountHolder`

The attribute `balance` is not added because it is a private attribute of the object.

6. Now add an implementation of the BO. Use the following parameters:

Parameter	Value
Pattern for handling state data	Caching
Object Reference	Use lazy evaluation
Data Object Interface	Create a new one now
Implementation Inheritance	IManagedClient IManaged Client::IManageable
Implementation Language	Java
Key Selection	AccountKey
Handle	none
Attributes to Override	none
Methods to Override	none
Data Object File Name	AccountDO
Data Object Interface Name	AccountDO
State data	all

7. Implement the **changeBalance** method. The implementation of the BO methods is the only code you have to write for the server application.

For this application, we implement the methods in Java. Include the following logic:

```
 if( ( iBalance + anAmount ) < 0 )
throw new NotEnough ( ) ;

iBalance += anAmount ;
namingService.narrow anObject
```

8. Implement the **getBalance** method.

```
1 return iBalance
```

9. Under the **File Adornments** folder in the Methods panel, select **AccountBOPrologue**. This is where you can include any import statements your methods may need to resolve exterior class definitions. Include the following import statement:

```
import AccountPackage.*;
```

Note

The package AccountPackage is generated when you build the DLLs for your server application in the following section. As part of the build process, Java files are generated from the IDL files from the model. In this sample, *Account.idl* contains the exception `NotEnough`. When the Java files are generated, the `NotEnough` exception classes are placed into the AccountPackage, and because of the way Object Builder generates the files for your Java BO, you must specify the import statement above to resolve the reference to `NotEnough` in the **changeBalance** method.

10. For the AccountDO, add a Data Object implementation with the following parameters:

Parameter	**Value**
Environment	BOIM with any Key
Form of Persistent Behavior and Implementation	Transient
Handle for Storing Pointers	Home name and Key
Key helper	AccountKey
Copy helper	AccountCopy

The Data Object and its implementation are now displayed under the **User-Defined Data Objects** folder. The AccountDO in this folder is simply another view of the AccountDO displayed under the AccountBO in the **User-Defined Business Objects** folder.

11. Add a Managed Object to the **AccountBO**. Accept all default settings.
12. Generate the code for all files from **User-Defined Business Objects**.

11.4.1.2 Create the model with XML

Using XML, you can define the object model for your server application; then import those definitions into the Object Builder. From the Object Builder, you can also export your object model out to XML format.

1. Open the object model for the C++ BO Server, either the model you created in the previous section or the model on the CD from the directory *\CBConnector\11-Transient-Object\CppBOServer*.
2. From **User-Defined Business Objects**, select **Export**. This will export the BO definitions in XML format and place them in a file *Export\udbo.xml* in the *Working* directory.
3. Create a new Object Builder model in a separate directory. This will be your model for the Java BO server application.
4. From **User-Defined Business Objects** in your new model, select **Import**, and select the file *Export\udbo.xml* from the C++ BO Server *Working* directory.

   Once the XML file has been imported, you will have an exact copy of the C++ BO model in your current model. You need only to alter a few settings to change the C++ BO into a Java BO.
5. Open the properties of the BO implementation, **AccountBO**. On the Implementation Language SmartGuide page, select **Java**.

   Now you need to rewrite the **getBalance** and **changeBalance** methods in Java.
6. Under the **User-Defined Methods** folder, open the properties of each method and verify that **Use the implementation defined in the editor pane** is selected.
7. Implement the **changeBalance** method.

```
if ( ( iBalance + anAmount ) < 0 )
throw new NotEnough ( ) ;

iBalance += anAmount;
```

8. Implement the **getBalance** method in Java.

```
1.return iBalance;
```

9. Under the **File Adornments** folder in the Methods panel, select **AccountBOPrologue**. This is where you can include any import statements your methods may need to resolve exterior class definitions. Include the following import statement:

```
1  import AccountPackage.*;
```

Note

The package AccountPackage is generated when you build the DLLs for your server application in the following section. As part of the build process, Java files are generated from the IDL files from the model. In this sample, *Account.idl* contains the exception `NotEnough`. When the Java files are generated, the `NotEnough` exception classes are placed into the AccountPackage. and because of the way Object Builder generates the files for your Java BO, you must specify the import statement above to resolve the reference to `NotEnough` in the **changeBalance** method.

10. Generate the code for all files from **User-Defined Business Objects**.

11. Because of the way Object Builder generates the Java BO implementation, you must edit the generated Java file *_AccountBOBase.java* located in the *Working* directory. Open this file in an editor, and in the definition of **changeBalance**, replace `Account.NotEnough` in the `throws` clause and `NotEnough` in the `throw` statement with `AccountPackage.NotEnough`.

11.4.1.3 Build the server application

In the **Build Configuration** folder, execute the following steps:

Add a client DLL named AccountTRJC with the following parameters:

Parameter	Variable
Name	AccountTRJC
Client Source Files	all
Libraries to Link With	none

Add a server DLL named AccountTRJS with the following parameters:

Parameter	Value
Name	AccountTRJS
Server Source Files	all
Libraries to Link With	AccountTRJC (the client library)

Generate the makefiles for all targets.

Build all targets of the server application.

11.4.2 Package

In this section, we describe how to package the transient Account application. This includes the following steps:

1. Definition of the application using the Object Builder.
2. Creation of an installation image using InstallShield.

11.4.2.1 Define the application

From the **Application Configuration** folder, add an Application Family named **AccountTRJAppFam**.

For this application family, add a server application named **AccountTRJAppS** with parameters:

Parameter	Value
Application Name	AccountTRJAppS
Initial state of application	stopped
Additional Executables	none

Add a Managed Object for the server application with the following parameters:

Parameter	Value
Managed Object	AccountMO AccountMO
Primary Key	AccountKey AccountKey
Copy Helper	AccountCopy AccountCopy
Data Object Implementation	AccountDOImpl AccountDOImpl
Container	CachedTransientObjects
Default Home	Default Home
Home Name	Default Home
Name in Factory Finding Service Registry	AccountMOFactory
Name in Naming Service Registry	AccountMOHome

11.4.2.2 Create an installation image

From the AccountTRJAppFam, generate the installation scripts. This step also creates the System Management data in the form of a DDL file.

Build the installation image. This process should create a subdirectory *AccountTRJAppFam* in your working directory.

11.4.3 Install

To install and configure the server application, follow these steps:

Install the server application on your server machine (see 7.1, “Install the application” on page 89).

Configure an **AccountTRJServer** in a new Management Zone, **AccountTRJ Management Zone**. Use a new configuration, **AccountTRJ Configuration**, for this server.

Note: For this sample, you must use exactly this server name for the application server to enable the execution of the client programs, because the single Location Object you installed refers to it.

Activate the configuration.

11.5 Client

These sections guide you through the implementations of C++, Java, and ActiveX clients. In the following samples, we only describe what is different in the client applications, as related to this sample and the general descriptions given in Chapter 10, “Sample client development approach” on page 117.

11.5.1 The C++ client

We remember that the **acctproxy** class is the application-dependent class, a subclass of the **cbcproxy** class, which hides the CBConnector specifics from the client application.

11.5.1.1 The acctproxy

The implementation of **acctproxy** is exactly the same as described in 10.2, “C++ Implementation” on page 121, apart from the hardcoded scope and interface information in the **initCBC** method of **acctproxy**.

11.5.1.2 The GUI

The GUI displays the three fields of the **Account** object; the accountNumber, the accountHolder and the balance. In addition, there is a button that lets you create a new account based on an accountNumber and an accountHolder, a button that lets you find an account with a certain accountNumber, and a button that will delete an account based on an accountNumber.

At the bottom, there is an input field where you can type an amount and a button to let you update the balance of the current account with this amount.

The GUI consists of an Account pane and a main window. To make the Account Business Object View reusable, it is put in its own pane.

The pane contains an **AcctCBCProxyWrapper** and an **AccountWrapper** as described in 10.2, “C++ Implementation” on page 121.

In the main window, where the Account pane resides, we have an **AcctCBCProxyWrapper** part. This part is instantiated when the window is created, and it is passed as a parameter to the Account pane that has a variable holder for it.

The **AcctCBCProxyWrapper** handles the initialization of the global CBConnector environment, and finds the home. This is done by connecting the visible event of the window to the **init** method of the **AcctCBCProxyWrapper**.

The **createBO** method of the **AcctCBCProxyWrapper** class in the Account pane is called when the **create** button is clicked. The text property of the accountNumber field is passed as the incoming parameter. The **AcctCBCProxyWrapper** returns an **AccountWrapper** instantiated with the attributes of the Account BO and places it in the **AccountWrapper** variable holder. The fields of the **AccountWrapper** variable are connected to the fields of the window. Note that a conversion must take place to show the balance of type long in a text entry field. The **findBO** and **deleteBO** methods are called in the same way. Notice that if an exception is thrown, a dialog box will display.

The update button calls the **changeBalance** method of the current **AccountWrapper**. It passes the amount as a parameter. The **AccountWrapper** converts the entry field string to a double and calls the **changeBalance** method on the Account Business Object. After the method is executed, it calls the **getBalance** method on the Business Object and updates its balance attribute with the new value. If an exception occurs, a dialog box will appear notifying the user that there is not enough money in the account to make this update.

The Transient sample with Copy Helper. For this sample, we added a button called Create with Copy Helper on the Account pane. This button calls the **createBOWithCopyHelper** method of the **CBCProxyWrapper**. It takes both the accountNumber and the accountHolder as parameters, and returns an account wrapper.

The **AccountWrapper** uses the *Account* header files and the *AccountTRC.dll* and *.lib* files that were created when building the Business Object in Object Builder. The CBCProxyWrapper uses the **cbcproxy** and **acctproxy** classes.

You compile the code using the Workframe environment. This process is described in Appendix C, “Client development with VisualAge for C++” on page 413.

Run the client by typing the following command in the *Visual C++* directory:

```
AccountTRC.exe
```

Note

This client uses the Java BO implementation by referencing:

```
AccountTRJServer-server-scope associated to the Java BO implementation.
```

11.5.2 The Java client

All Java source and class files for this sample are located on the CD in the directory *\CBConnector\11-Transient-Object\Client\Java*.

11.5.2.1 Client proxies

A Java application needs stubs, Key, and Copy local objects to create, find, and use remote objects. Object Builder generates them for you and puts all the required files in a file named: jcb<client_dll_name>.jar. For example, if you named your client DLL AccountTRC, then the jar file will be jcbAccountTRC.jar.

This file is located in the directory:
<drive>:<your_directory>\Working\Nt\PRODUCTION\Jcb

11.5.2.2 Implement the Java client

For a full description of how to initialize the Component Broker client environment, and how we have implemented the Java client for this sample, please read 10.3, “Java implementation” on page 141. Because this sample highlights the use of the Primary Key and Copy Helper, pay close attention to how we create **Account** objects in the two different ways.

11.5.2.3 Generate executables

To create the CLASS files for the Java client, you need to invoke the Java compiler (javac) from the *Source* directory, specifying the subdirectories where all the sources are located and the directory to put the resulting CLASS files:

```
set classpath=.;.\jcbAccountTRJC.jar;%classpath%
javac -d ..\class *.java
```

11.5.2.4 Run the client

You can run the Java client in three different ways:

- As an application
- As an applet
- Using HotJava or any other browser supporting JDK 1.1

Run application

Place the JDK *Lib* directory and Component Broker Java ORB classes, *somojor.zip*, in your environment `CLASSPATH`, and make sure the JDK runtime is in your environment `Path`.

```
set path=<drive>:\jdk1.1.7\bin;%path%
set classpath=.;<drive>:\jdk1.1.7\lib\classes.zip;
<drive>:\cbroker\lib\somojor.zip;
.\jcbAccountTRC.jar;.;%classpath%;
```

Run the Java client program with the following command:

```
java AccountApp <hostname> <port> AccountTRServer-server-scope
```

In general, you can find sample command (CMD) files on the CD under the *Class* directory for the samples. For this sample, you will find *runApp_CppBO.cmd* which runs the client against the C++ BO server, and *runApp_JavaBO.cmd* which runs against the Java BO server. The only difference between the two commands is the name of the `scope`. If you use the CMD files, make sure you change the `hostname` and `port` to reflect your server machine.

Run applet

Create an HTML page that loads the *AccountApp* applet with the `hostname` and `port` as parameters. The HTML pages provided on the CD in the *Class* directory for this sample include the applet tag as follows:

```
<html>
<head>
 <title>Transient Object Sample Applet</title>
</head>

<body>

<applet codebase="." code="AccountApp" width=684 height=503>
 <param name=package value="Transient">
 <param name=hostname value="atlantic.almaden.ibm.com">
 <param name=port value="900">
 <param name=scope value="AccountTRServer-server-scope">
</applet>

</body>
</html>
```

You will also find the file *Transient_JavaBO.html*, which runs the applet against the Java BO server and the *Transient_CppBO.html* which runs the applet against the cpp BO server. The only difference between the applet tags in the two files is the value of the `scope` parameter.

Place the JDK *Lib* directory and Component Broker Java ORB classes, *somojor.zip*, in your environment `CLASSPATH`, and make sure the JDK runtime is in your environment `Path`.

```
set path=<drive>:\jdk1.1.7\bin;%path%
set classpath=.;<drive>:\jdk1.1.7\lib\classes.zip;
<drive>:\cbroker\lib\somojor.zip;
.\jcbAccountTRC.jar;.;%classpath%;
```

Run the applet in the applet viewer provided with the JDK, using the the HTML page created above. Do this by executing the following command:

```
appletviewer Transient_CppBO.html
```

In general, you can find sample command (CMD) files on the CD under the *Class* directory for the samples that run the applets. For this sample, you will find *runApplet_CppBO.cmd*, which runs the applet against the C++ BO server, and *runApplet_JavaBO.cmd*, which runs against the Java BO server. If you use the CMD files, make sure you change the `hostname` and `port` applet parameter values to reflect your server machine.

Run applet in a browser

You can run the client applet in HotJava or any other browser that supports JDK 1.1. To use HotJava:

- Create an HTML page that loads the applet as described above.
- Install and start a Web server on any machine. You can literally run a Java client from anywhere as long as you have the Java ORB class files (*somojor.zip*), and you know the name of the bootstrap hostname and port number.
- Copy the following files into a WWW directory of your Web server (let's call it *CBApplets*):

  somojor.zip

  All the CLASS files for the Java client applet (in a ZIP file if you prefer)

  The HTML file that loads the applet (*Transient_CppBO.html*)

In HotJava, you still need to configure the security in the following way:

- From the **Edit** menu, select **preferences -> applet security**.
- Check unsigned applet to medium security.
- On the next page (advanced security), you need to add a site, which is your Web server, and then select **applet may access all properties**.

Run the applet by opening your HTML page located on your Web server from HotJava:

```
http://<hostname>/CBapplets/Transient_CppBO.html
```

11.5.3 The ActiveX client

The table in 11.5.3.2, “File cross reference” on page 183 shows all source files used in this sample. You can find the files in the following subdirectory:

```
\CBConnector\11-Transient-Object\Client\ActiveX
```

Refer back to 10.4, “ActiveX implementation” on page 157, for an explanation of how implement your client and how to find the Business Object and invoke its methods. The variables in the code snippets in the general ActiveX chapter and their values for this sample are shown below:

Parameter	Value
scope	"AccountTRJServer-server-scope"
InterfaceName	"Account.object interface"
appName	"Account"

11.5.3.1 Building a client install image

In the following steps, you build the client install image:

- Invoke *init account* to generate the includes for the makefile.
- Execute `nmake`.
- Register the DLLs with *register.bat*.

11.5.3.2 File cross reference

The table below explains functionalities of the files used in this scenario for ActiveX client implementation:

File	Description
Account.dll	Dynamic Link Library containing Client runtime bindings
AccountKey.dll	Dynamic Link Library containing runtime bindings for Key proxy
AccountCopy.dll	Dynamic Link Library containing runtime bindings for Key proxy
register.bat	Batch file used to register the Account, AccountKey and AccountCopy DLLs
unregister.bat	Batch file used to unregister the Account, AccountKey and AccountCopy DLLs

At this point, we have implemented the server and the client applications. We are finally ready to deploy and execute our transient scenario sample.

11.5.3.3 Using the ActiveX CD sample

When you run the ActiveX Transient sample (accountTRX.exe), a window with an account panel pops up.

Create a new account by typing in a number in the "Account #"-box; type in an account holder and press **Create**. Do the same for a couple more accounts.

Find one of your previously created accounts by typing in an account number and pressing **Find**.

Finally, put some money in your found account by typing in a balance in the "Update balance"-box and press **changeBalance**. Voila'. You are now a rich person!

Note

This client uses the Java BO implementation by referencing:

```
AccountTRJServer-server-scope associated to the Java BO implementation.
```

11.5.3.4 Create an installation image

From the AccountTRClientAppFam, generate the installation scripts.

Build the installation image. This process should create an *AccountTRClientAppFam* subdirectory in your working directory. The install image is located in the *AccountTRClientAppFam\Disk1* directory. You can now transfer the Disk1 directory to the installation site, and run the setup to install your clients.

11.6 What did you learn?

This concludes our first Component Broker Connector application. You learned how to create a Transient Object server application with both a C++ BO and a Java BO, how to import and export object model definitions in XML format, and how to create objects from a client using both the Primary Key and Copy Helper.

In the next chapter, we continue with our **Account** sample by extending it with a specialized home.

Chapter 12. Specialized home sample

The specialized home provides you with a richer interface to homes than what is provided by default. In this chapter, we show you how you can create object keys on the server side and create objects in an optimized way.

12.1 What you will learn

Component Broker Connector provides a default home implementation called **IHome**. This class has only limited functionality to find or create a Managed Object. There are a number of cases where you would like to extend the interface of the home class, for example, when you want to optimize the creation of an object. The extended home objects are called **specialized** or **customized homes**.

In this chapter, you will learn how to extend our transient account application from the previous chapter with a specialized home that is able to create the Account Object with all attributes set by only one method request. We also add a **findByAccountNumber** method, which simplifies the usage of the **findByPrimaryKey** method.

12.2 CBConnector componentry

The **IHome** class provides your home with the methods **findByPrimaryKey** and **createFromPrimaryKeyString** in order to find and create objects. In some cases, these two methods may not be sufficient for what you want to do. The specialized home enables you to create your own interface to manipulate the Home objects. Examples of such methods can be **createAccount** or **findByAccountNumber**, which do not require a Primary Key from the client.

Sometimes, the client program might not have a clue how to generate a Primary Key value for an object and thus have no means to create an object by using **createFromPrimaryKeyString** or **createFromCopyString**. In this chapter, you will see how a specialized home creates an account with an artificial account number by combining system time components (hhmmssfff).

In real life, this kind of account number generation might not be appropriate, since you might want to keep track of generated account numbers and generate the account number according to some persistent information.

Like normal homes, the specialized homes reside in the System Managed Object Container (see Figure 47).

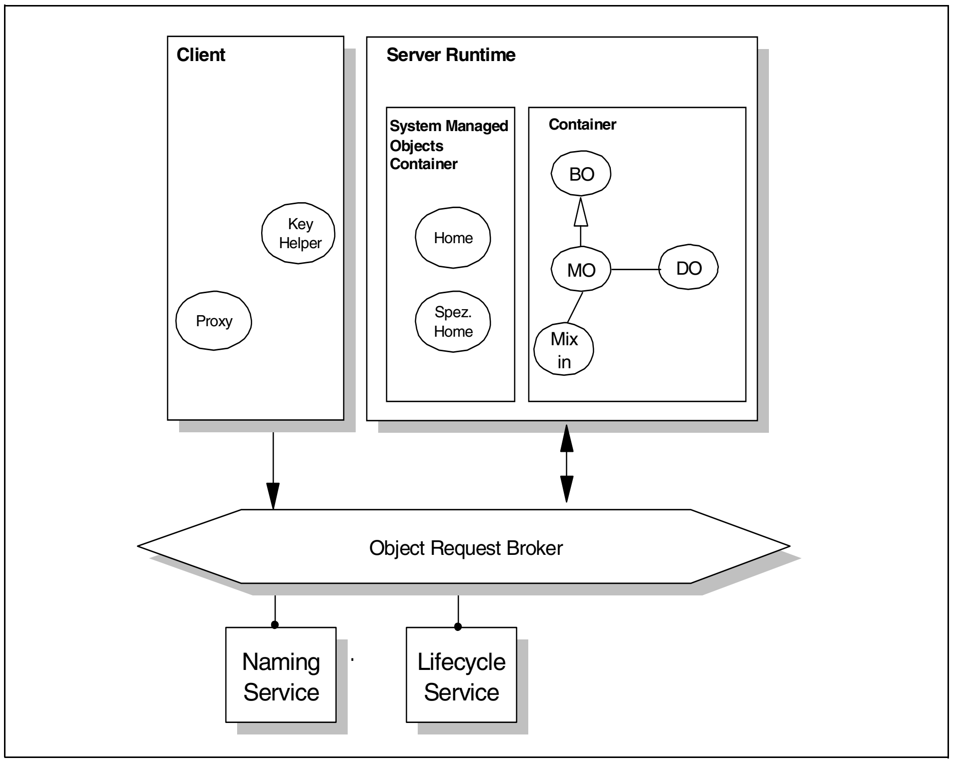


Figure 47. Specialized home sample

The figure also shows the main components used in this scenario:

- Object Request Broker
- Naming Service
- LifeCycle Service
- Container
- Account Managed Object Assembly (Business Object BO, Managed Object MO, Data Object DO, Mixin Object) together with the proxy object in the client address space

The specialized home is a Managed Object itself. The Managed Object assembly is shown in Figure 48.

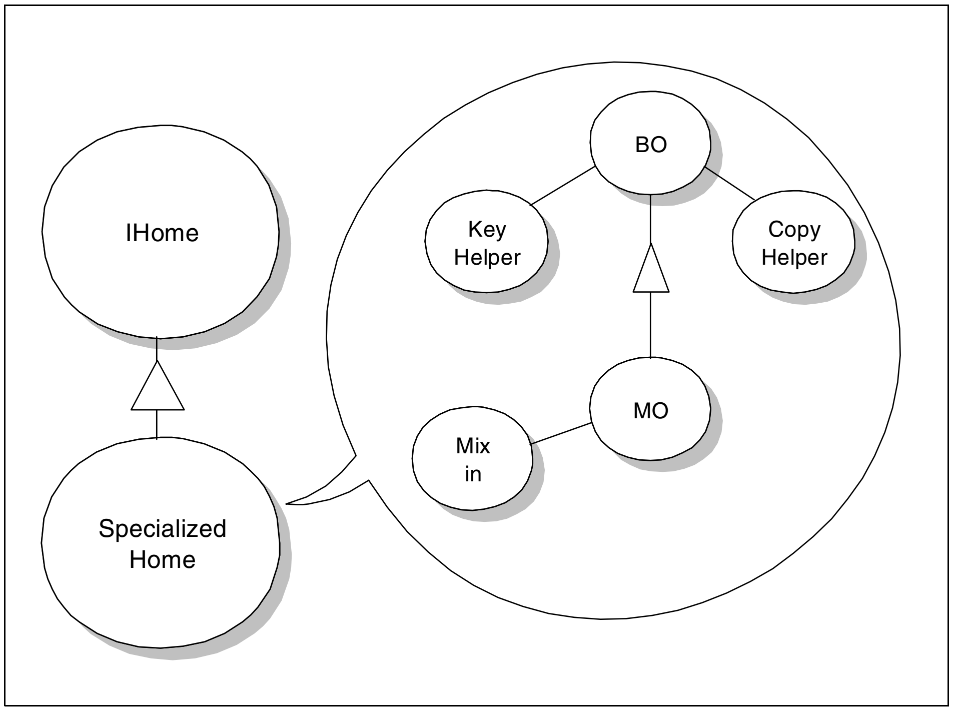


Figure 48. Specialized home as Managed Object

In this figure, a Key Helper and a Copy Helper are included which we use within the methods of the Home Object.

The Key Helper was used in the first transient account sample. The **Copy Helper** is used to create new objects in the **createAccount** method. Normally, an **Account** Object is created by calling the **createFromPrimaryKeyString** method on a home reference.

In this case, however, the Account Object is created by passing the `accountHolder` string to the server. A timestamp is used as a key for this object, and the object's accountNumber is set to this timestamp.

For convenience, we use the Copy Helper class to create the object in one go as we did in Chapter 11, “Transient Object sample” on page 161.

12.3 Server

The following sections take you through all the steps you need to perform in order to build, package, and install the specialized home server application.

Note

All code snippets have enumerated lines for easier orientation in the code.

12.3.1 Build

In this section, we describe how to extend the transient Object Builder model with a specialized home.

Through the Object Builder walkthrough, we provide you with the correct parameter settings. If no suggestions are made by us for an Object Builder page, use the defaults provided by Object Builder.

We strongly recommend that you open Help for each field of the SmartGuide pages in order to understand more of the dialectic behind the input fields.

For the specialized home, you can leave the Account BO and Managed Object as it is. Only the application configuration will change.

To keep the data of the transient model, it is a good idea to make a copy of the model data and the content of the *Working* directory.

To implement a specialized home for the Account Object, you need to create a new Managed Object, that we call **AccountHome**.

12.3.1.1 Create the model

Create a new Object Builder model in a separate directory.

From **User-Defined Business Objects**, add a new file, **AccountHome**, using the following parameters:

Parameter	Value
Name	AccountHome
Constructs	none
Include files	IManagedClient Account

In the "Files to include" pane, click the **Component Broker Customized Home** button to indicate that this is not a Business Object, but a customized Home Object.

For this file, add an interface, **AccountHome**, with the following parameters:

Parameter	Value
Name	AccountHome
Constructs	none
Interface inheritance	IManagedClientIManagedClient::IHome
Attributes	none

Add methods **createAccount** and **findByAccountNumber** with the following definitions:

Method	Return Type	Parameter	Exception
createAccount	Account Account	accountHolder in <string30>	IManagedClient IManagedClient::DuplicateKey
fingByAccountNumber	Account Account	in accountNumber string<10>	IManagedClient IManagedClient::NoObjectWKey

Add an implementation of the Business Object. Use the following parameters:

Paramter	Value
Implementation Inheritance	BOIMExtSystemObject IBOIMExtSystemObject::Home
Implementation Language	C + +
Key Selection	none
Handle	none
Attributes to Override	none

Implement the **createAccount** method:

```
AccountCopy_var accountCopy;
CORBA::object_var object;
ByteString* copyString;
SYSTEMTIME st;
char key [10];

GetSystemTime( &st );
sprintf( key,"%02d%02d%02d%03d",( int ) st.wHour,(int)st.wMinute,
 ( int) st.wSecond,( int ) st.wMilliseconds );

accountCopy = AccountCopy::_create( );
accountCopy->accountNumber( key );
accountCopy->accountHolder( accountHolder );
copyString = accountCopy->toString( );

object = createFromCopyString( *copyString );

delete copyString;

return Account::_narrow( object );
```

Implement the **findByAccountNumber** method:

```
CORBA::object_var object;
AccountKey_var accountKey;
ByteString* keyString;

accountKey = AccountKey::_create( );
accountKey->accountNumber( accountNumber );
keyString = accountKey->toString( );

object = findByPrimaryKeyString( *keyString );

delete keyString;

return Account::_narrow( object );
```

Add a Managed Object named **AccountHomeMO**. Use the following parameters:

Parameter	Value
Implementation Inheritance	IBOIMExtSystemObject IBOIMExtSystemObject::Home

Generate the code for all files from **User-Defined Business Objects**.

12.3.1.2 Build the server application

In the **Build Configuration folder**, execute the following steps:

Add a client DLL named AccountTRC with the following parameters:

Parameter	Value
Name	AccountTRC
Client Source Files	all
LIbraries to Link With	none

Add a server DLL named AccountTRS with the following parameters:

Parameter	Value
Name	AccountTRS
Server Source Files	all
Libraries to Link With	AccountTRC (the client library)

Generate the makefiles for all targets.

Build all targets of the server application.

12.3.2 Package

In this section, we describe how to package the specialized home server. This includes the following steps:

1. Definition of the application using the Object Builder
2. Creation of an installation image using InstallShield

12.3.2.1 Define the application

If you have extended the transient account program, delete the application family from the **Application Configuration**.

From the **Application Configuration**, add an Application Family named **AccountSHAppFam**.

For this Application Family, add a server application named **AccountSHAppS** with parameters:

Parameter	Value
Application Name	AccountSHAppS
Initial state of application	stopped
Additional executables	none

Add a Managed Object, **AccountMO**, for the server application with the following parameters:

Paramter	Value
Data Object Implementation	AccountDOImpl AccountDOImpl
Container	CachedTransientObjects
Default Home	Customized Home
Home Name	AccountHomeMO AccountHomeMO
Select DLL (for customized home)	AccountSHS
Name in Factory Finding Service Registry	AccountMOFactory
Name in Naming Service Registry	AccountMOHome
Name of Contexts	none
Locatable by Cell	yes
Locatable by Workgroup	yes
State of Home	exists

Add a Managed Object, **AccountHomeMO**, for the server application with the following parameters:

Parameter	Value
Data Object Implementation	none
Container	CachedSystemsManagedObjects
Default Home	Default Home
Home Name	BOIMHomeOfNotRegHomes
Name in Factory Findning Service Registry	none
Name in Naming Service Registry	none
Name Contexts	host/resources/servers/ $ServerName$/collections
Locatable by Cell	not checked
Locatable by Workgroup	not checked
State of Home	created

12.3.2.2 Create an installation image

From the AccountSHAppFam, generate the installation scripts. This step also creates the System Management data in the form of a DDL file.

Build the installation image. This process should create a *AccountSHAppFam* subdirectory of your working directory. The install image is located in the *AccountSHAppFam\Disk1* directory. You can now transfer the Disk1 directory to the installation site and run the setup to install.

12.3.3 Install

To install and configure the server application, follow these steps:

1. Install the server application on your server machine.
2. Configure an **AccountSHServer** in a new Management Zone, **AccountSH Management Zone**. Define a new configuration **AccountSH Configuration** for this server.
3. Activate the configuration, and start the server with `Run Immediate`.

12.4 Client

These sections describe what is different in the client applications, as related to the general descriptions given in Chapter 10, “Sample client development approach” on page 117.

12.4.1 The C++ client

We remember that the **acctproxy** class is the application dependent class, a subclass of the **cbcproxy** class, which hides the CBConnector specifics from the client application.

12.4.1.1 The acctproxy

This implementation of **acctproxy** defines the three methods **initCBC**, **createBO** and **findBO**, as we have seen earlier.

The **initCBC** calls the overloaded **init** method of **cbcproxy** (the parent of **acctproxy**),to get the factory for the specialized home. Thereafter, it uses the **narrow** method of the **AccountHome** class to get the specialized home object, which is stored in the **acctproxy** object. See Figure 49.

```
;-------------------------------------------------------------------;
;       initialize Component Broker                               ;
;-------------------------------------------------------------------;
*/
int  _Export acctproxy::initcbc()
  {
CORBA::object_ptr factory;
/*
;------- calling init with dummy argument - to call overload method -;
*/
   factory = init("AccountSHScope","Account",0);
   try
     {
      accountHome = AccountHome::_narrow(factory);
      CORBA::release(factory);
     }
   catch ( ... )
     {
      cout << "narrow home failed" << endl;
      return FALSE;
     }
   if (CORBA::is_nil(accountHome))
     return FALSE;
   else
     return TRUE;
  }
```

Figure 49. Initialize component broker

The **createBO** takes the **accountHolder** as a parameter, and invokes the **createAccount** on the specialized home to create the Managed Object. Then it is narrowed to the business object in the usual way. The **findBO** takes the key as parameter, but uses the **findByAccountNumber** method on the specialized home to find the Managed Object. Then it is narrowed to the Business Object in the usual way. See Figure 50.

```
/*
;------------------------------------------------------------------;
;       create the Business Object - from accountHolder             ;
;------------------------------------------------------------------;
*/
Account_ptr _Export acctproxy::createbo(char *holder)
  {
IManagedClient::imanageable_ptr mo = NULL;
/*
;------- create Managed Object -------------------------------------;
*/
   mo = accountHome->createAccount(holder);
   if (CORBA::is_nil(mo))
      return NULL;
/*
;------- narrow to Business Object ---------------------------------;
*/
   try
      {
       return Account::_narrow( mo );
      }
   catch( ... )
      {
       cout << "narrow Managed Object failed" << endl;
       return NULL;
      }
  }
/*
;------------------------------------------------------------------;
;       find the Business Object - from generated key               ;
;------------------------------------------------------------------;
*/
Account_ptr _Export acctproxy::findbo(char *acckey)
  {
IManagedClient::imanageable_ptr mo = NULL;
/*
;------- find Managed Object - key generated on server side ---------;
*/
   mo = accountHome->findByAccountNumber(acckey);
   if (CORBA::is_nil(mo))
      return NULL;
/*
;------- narrow to Business Object ---------------------------------;
*/
   try
      {
       return Account::_narrow( mo );
      }
   catch( ... )
      {
       cout << "narrow Managed Object failed" << endl;
       return NULL;
      }
  }
```

Figure 50. Create/find the business object

12.4.1.2 The GUI

The GUI doesn't change in appearance, only in function. The create button will now create an object based on the **accountHolder** field, and not the **accountNumber** field. When the object is created, the key generated by the server will show up in the **accountNumber** field. To find an object, you need to use this generated **accountNumber** as input.

For this client, all we need to do is to have the **accountHolder** field as an input to the create object method, instead of the **accountNumber** as in the previous sample.

Finally, we replace our **acctproxy** to that generated for this sample, and change the interface headerfiles and the client DLL generated by the Object Builder for the specialized home.

Run the client by typing the following command in the VisualC++ directory:

```
AccountSHC.exe
```

12.4.2 The Java client

All Java source and class files for this sample are located on the CD in the directory *\CBConnector\12-Specialized-Home\Client\Java*.

12.4.2.1 Client Proxies

Generate the Java client proxies for these IDL files as described in 6.7.2, “Java client programming model” on page 76.

12.4.2.2 The SpecializedHomeAccountProxy class

In the Java client described in 11.5.2, “The Java client” on page 180 used an application dependent class called **AccountProxy** which is a subclass of **CBProxy**. When the **AccountProxy** is initialized, it finds the Factory Finder and home in which the **Account** Managed Objects reside.

In the server application for this sample, you created an **Account** that resides in a specialized home called **AccountHome AccountHome** is simply a subclass of a normal home of type **com.ibm.IManagedClient.IHome**. So, we created a **SpecializedHomeAccountProxy** class that is initialized as usual, finding the Factory Finder and home of **Account** objects, but once the home is found, we have to further *narrow* it to an **AccountHome**, so that we can use the methods defined by its interface. This is shown in the **init** method in Figure 51:

```
/**
 * init SpecializedHomeAccountProxy
 */
public void init (String scope) {

  super.init(scope, "Account.object interface");

  try {
    CBCBase.traceIt("narrowing to an AccountHome");
    accountHome = AccountHomeHelper.narrow(super.home);

    CBCBase.traceIt("AccountHome found");
  }
  catch (Throwable e) {
    CBCBase.traceIt("Unable to narrow to an AccountHome");
  }
}
```

Figure 51. Narrow to a specialized accounthome

Once you have the **AccountHome**, you can create and find **Account** objects using the methods defined for this "specialized" home. Instead of creating an object with a normal home by creating a Copy Helper and calling **createFromCopyString()**, all you have to do is call the **createAccount()** method of the **AccountHome** by passing it the name of the AccountHolder. Simarily, to find an object, instead of having to pass a Primary Key to the **findByPrimaryKeyString()** method, all you have to do is call **findByAccountNumber()** by passing it an `accountNumber`. This is shown in the following code snippets in Figure 52:

```
/**
 * create an Account Managed Object using the AccountHome
 * @return Account
 */
public Account createAccount (String accountHolder) {

  Account account = null;

  try {
    org.omg.CORBA.Object o = null;
    o = accountHome.createAccount(accountHolder);

    if (o == null) {
      throw new NullPointerException();
    }

    // narrow to an Account
    account = AccountHelper.narrow(o);

    if (account == null) {
      throw new NullPointerException();
    }
   }
   catch(Throwable e) {
    account = null;
   }

   return account;
}

/**
 * find an Account Managed Object using the AccountHome
 * @return Account
 */
public Account findAccountByAccountNumber (String accountNumber)  {

  Account account = null;
  try {
    org.omg.CORBA.Object o = null;
    o = accountHome.findByAccountNumber(accountNumber);

    if (o == null) {
      throw new NullPointerException();
    }

    // narrow to an Account
    account = AccountHelper.narrow(o);

    if (account == null) {
      throw new NullPointerException();
    }
  }
  catch(Throwable e) {
    account = null;
  }

  return account;
}
```

Figure 52. Creating and finding accounts using the accounthome

12.4.2.3 The GUI

The GUI doesn't change in appearance from the Transient Object sample, only in function. The create button will now create an object based on the accountHolder field, and not the accountNumber field. When the object is created, the key generated by the server will show up in the accountNumber field. To find an object, you need to use this generated accountNumber as input.

12.4.2.4 Run the client

You can run this Java client in the three ways described in 11.5.2.4, “Run the client” on page 180–as an application, as an applet in the applet viewer, and as an applet in a browser.

To run the client application, execute the command:

```
java AccountApp <hostname> <port> AccountSHServer-server-scope
```

To run the client as an applet, embed the applet in an HTML page (*SpecializedHome.html*, and open that page in the applet viewer with the command:

```
appletviewer SpecializedHome.html
```

There are sample command (CMD) files on the CD in the directory *\CBConnector\12-Specialized-Home\Client\Java\Class* for running the client as both an application an applet. They are called *runApp.cmd* and *runApplet.cmd*. If you use these, remember to change the `hostname` and `port` parameters (in *runApplet.cmd* and *SpecializedHome.html*) to reflect your bootstrap server.

12.5 What did you learn?

This chapter demonstrates the implementation and the usage of a specialized home. Specialized homes let you implement a richer interface to homes than what is provided by default. Specialized homes can also be a very efficient way to configure a new object with the essential state date with one method request avoiding many requests over the ORB.

In the next chapter, we continue with our **Account** example by making our server objects persistent in a relational database.

Chapter 13. Persistent Object sample

The Persistent Object scenario allows us to move another step forward in our incremental samples. In this sample, we store the object's essential state to a datastore.

13.1 What you will learn

In this chapter, you will learn how to store our Account objects' essential state of data to a relational database. We introduce the Persistent Object and the Database Schema mapper as new concepts.

13.2 CBConnector componentry

In this section, we briefly describe the most important CBConnector components relating to this sample.

13.2.1 Transaction Service

The Object Transaction Service takes care of all server-side transaction management. If the client application requires control when the transactions are started and committed, it can use the **Current** Object to define the transaction boundaries. The container where objects are assigned defines the transactional behavior. The object itself remains untouched even though the transaction policy changes. The three transaction policies available are:

- **Start a new transaction and complete call**. Use this option if components require the Transaction Services at all times. If a method is called outside of the scope of an existing transaction, the Transaction Services start a new transaction for that method and commit the transaction when the method completes.
- **Throw exception and abandon call**. Use this option if components require the Transaction Services at all times, but you don't want transactions to be started automatically. If a method is called outside of the scope of an existing transaction, the object throws the following exception: **CORBA::transaction_REQUIRED**. If you select this option, transactions must be explicitly started and committed (for example, by the client that calls the method).

- **Ignore condition and complete call**. Use this option if some but not all components require the Transaction Services. If a method is called outside of the scope of an existing transaction, the method will complete without transaction support. While the Transaction Services will support components in this container, transactions must be explicitly started and committed when they are needed.

13.2.2 Data Object

The Data Object (DO) contains the logic to store, update, and delete objects from datastore, and to retrieve objects from the datastore. The Data Object knows how to cooperate with an Application Adaptor (AA) and thus implements all methods required by the AA. However, the (SQL) communication with the datastore is delegated to PO, thus keeping the DO's own code understandable and clean.

13.2.3 Persistent Object

The **Persistent Object** (PO) encapsulates the embedded SQL statements needed to insert, update, delete, and retrieve the essential state to and from the datastore. The PO contains the same attributes as the Data Object; so the mapping from DO to PO is very straightforward. The PO is generated by the Object Builder in the form of an SQX file that is processed by the generated makefile in order to create a DB2 bind file for the application. For an example, refer to Figure 53.

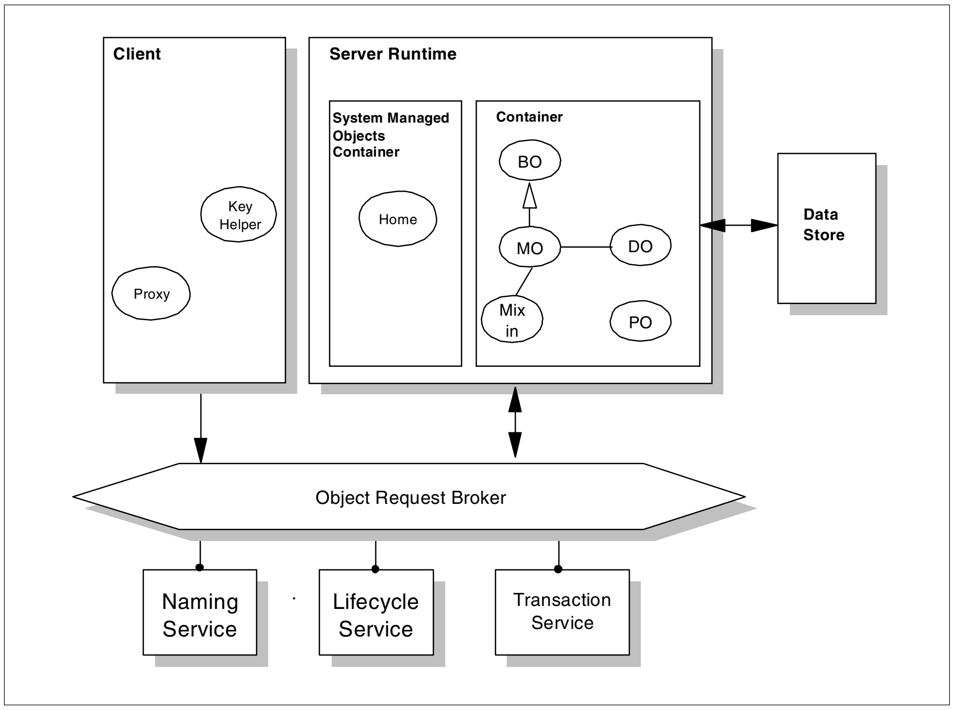


Figure 53. Persistent sample

Figure 53 shows the main components used in our simple transient scenario:

- Object Request Broker
- Naming Service
- Life Cycle Service
- Transaction Service
- Container
- Account Managed Object Assembly (Business Object BO, Managed Object MO, Data Object DO, MixIn Object, and Persistent Object PO) together with the proxy object in the client address space
- Datastore (in our case, a DB2 relational database)

13.3 Server

The following sections take you through all the steps you need to perform in order to build, package, and install the persistent server application.

Note

All code snippets have enumerated lines for easier orientation in the code.

13.3.1 Build

Implementing the server application consists of three major steps:

1. Specify the interface and the implementation of the Business Object and Data Object, including the helper objects.
2. Create the database.
3. Generate and build the application modules.

Through the Object Builder walkthrough, we provide you with the correct parameter settings. If no suggestions are made by us for an Object Builder page, use the defaults provided by Object Builder.

We strongly recommend to open help for each field of the SmartGuide pages in order to understand more of the dialectic behind the input fields.

13.3.1.1 Create the model

1. Create a new Object Builder model in a separate directory.
2. From **User-Defined Business Objects**, add a new file, **Account**, using the following parameters:

Parameter	Value
Name	Account
Constructs	Exception: NotEnough

3. For this file, add an interface, **Account**, with the following parameters:

Parameter	Value
Name	Account
Constructs	Exception: NotEnough

4. Add attributes to the Account Interface as described in the following table:

   Define the Account Interface with the attributes as described in the following table.

Attribute	Type	Implementation
accountNumber	string<10>	Public
accountHolder	string<30>	Public
balance	double	Private

   Add methods **changeBalance** and **getBalance** with the following definitions:

Method	Return Type	Parameter	Exception
changeBalance	void	anAmount double in	NotEnough
getBalance	double		

5. For the newly created Account Interface, create a Key Helper using the `accountNumber` as the Primary Key.
6. Create a Copy Helper with the attributes:
   - `accountNumber`
   - `accountHolder`

   The attribute `balance` is not added because it is a private attribute of the object.

7. Now add an implementation of the BO. Use the following parameters:

Parameter	**Value**
Pattern for handling state data	Caching
Object Reference	Use lazy evaluation
Data Object Interface	Create a new one now
Implementation Inheritance	IManagedClient IManagedClient::IManageable
Implementation Language	C + +
Key Selection	AccountKey
Handle	none
Attributes to Override	none
State data	all

8. Implement the **changeBalance** method. Include the following logic.

```
if ( ) iBalance + anAmount ) < 0 )
throw NotEnough ( );

IBalance += anAmount;
```

9. Implement the **getBalance** method.

```
return iBalance;
```

10. For the AccountDO, add a Data Object implementation with the following parameters:

Parameter	Value
Environment	BOIM with any Key
Form of Persistent Behavior and Implementation	Embedded SQL
Data Access Pattern	Local copy (*)
Handle for Storing Pointers	Home name and Key
Implementation Inheritance	IRDBIMExtLocalToServer IRDBIMExtLocalToServer::IDataObject
Key Helper	AccountKey
Copy helper	AccountCopy

Note

(*) - The recommended value and default is *delegating*. The matched setting in the container instance definition in *cached data in Data Object* is **Yes**. This combination locks the DO; so there is no multithreaded access on it. This configuration is also best from the performance perspective. In our sample, we just tested the possibility with *local copy*.

The Data Object and its implementation are now displayed under the **User-Defined Data Object** folder. The AccountDO in this folder is simply another view of the AccountDO displayed under the AccountBO in the **User-Defined Business Object** folder.

11.In the **User-Defined Data Object** folder, define a Schema by adding a Persistent Object and Schema to the **AccountDOImpl** object with the following parameters:

Parameter	Value
Group Name	AccntDB Schemas
Database	AccntDB
UserId	<your user id>
Package	AccntDB

You can leave the rest of the parameters at their defaults. The mapping between the definitions of data types in the database table and the attributes of the Persistent Object are manipulated here.

12.Add a Managed Object to the **AccountBO**. Accept all default settings.

13.Generate the code:

- Generate all files from User-Defined Business Objects.
- Generate all files from User-Defined Data Objects.

13.3.1.2 Create the database

Before you can compile and link the server application, you have to create the database and database table because the SQL precompiler needs to access the database definition.

To create the database and table, follow these steps:

1. Open a MSDOS window and execute the `db2cmd` command.
2. Change the path to your working directory.
3. Enter:

```
Db2 create database AccntDB
Db2 connect to AccntDB
Db2 -tf Account.sql
Db2 select * from Account
Db2 connect reset
```

13.3.1.3 Build the server application

In the **Build Configuration** folder, execute the following steps:

1. Add a client DLL named AccountDBC with the following parameters:

Parameter	Value
Name	AccountDBC
Client Source Files	all
Libraries to Link With	none

2. Add a server DLL named AccountDBS with the following parameters:

Parameter	Value
Name	AccountDBS
Server Source Files	all
LIbraries to Link With	AccountDBC (the client library)

3. Generate the makefiles for all targets.
4. Build all targets of the server application.

13.3.2 Package

In this section, we describe how to package the persistent account application.

13.3.2.1 Define the application

1. From the **Application Configuration** folder, add an application family named **AccountDBAppFam**.

2. For this application family, define a server application named **AccountDBAppS** with parameters as follows:

Parameter	Value
Application Name	AccountDBAppS
Initial state of application	stopped
Additional executables	none

3. From the **Container Definitions** folder, create a container instance named **ContainerOfAccountDBs** with parameters as follows:

Parameter	Value
Services	Use Transaction Services
Transaction Policies	Start a new transaction and complete the call
Passivation Policy	Passivate component at end of transaction
Business Object	Caching
Data Object	Local Copy
use of caching service	no

4. Add a Managed Object for the server application with the following parameters:

Parameter	Value
Data Object Implementation	AccountDOImpl AccountDOImpl
Container	ContainerOfAccountDBs
Default Home	Default Home
Home Name	BOIMHomeOfRegHomes
Name in Factory Finding Service Registry	AccountMOFactory
Name in Naming Service Registry	AccountMOHome
Locatable by Cell	yes
Locatable by Workgroup	yes
State of Home	exists

13.3.2.2 Create an installation image

From the AccountDBAppFam, generate the installation scripts.

13.3.3 Install

To install and configure the server application, follow these steps:

1. Install the server application on your server machine.

2. Configure an **AccountDBServer** in a new Management Zone, **AccountDB Management Zone**. Use a new configuration, **AccountDB Configuration**, for this server.

   When you install the server application, you will see that an application named Specific AccountDBAppS is created in addition to the AccountDBAppS application. When you configure the AccountDBAppS application, Specific AccountDBAppS and the default DB2 application named iDB2IMServices, need to be configured to the AccountDBServer as well.

   The Specific AccountDBAppS application is found under **Host Images -> Application Family Installs -> SpecificAccountDBAppFam**. The iDB2IMServices application is found under **Host Images -> Application Family Installs -> iDB2IMApplications**.

3. Ensure that you bound the *AccountPO.bnd* bind file to your database, AccntDB. If you did not, issue the following set of commands from the DB2 command line.

```
DB2 connect to AccntDB
DB2 bind AccountPO.bnd
DB2 connect reset
```

4. Activate the configuration.

   This scenario uses the DB2 Application Adaptor; therefore, stop the server and set the **open string** with the database administrator user ID and password for the database in the **XA Resource Manager images** of this server image.

5. Start the server with `Run Immediate`.

13.4 Client

These sections describe what is different in the client applications as related to the general descriptions given in Chapter 10, “Sample client development approach” on page 117.

13.4.1 The C++ client

We remember that the **acctproxy** class is the application-dependent class, a subclass of the **cbcproxy** class, which hides the CBConnector specifics from the client application.

13.4.1.1 The acctproxy

The implementation of **acctproxy** is exactly the same as described in 10.2, “C++ Implementation” on page 121, apart from the hardcoded scope and interface information in the **initCBC** method of **acctproxy**.

13.4.1.2 The GUI

The GUI implementation will not change from the transient account sample. Just replace the old **acctproxy** with the **acctproxy** object generated for this sample, and change the interface header files and the client DLL generated by the Object Builder for the persistent sample.

Run the client by typing the following command in the *VisualC++* directory:

AccountDBC.exe

13.4.2 The Java client

All Java source and class files for this sample are located on the CD in the directory *Samples\13-Persistent-Object\Client\Java*.

The Java client for this sample is exactly the same as the Java client described in 19.4.2, “The Java client” on page 314. The *only* difference in the code that you see on the CD is that we placed the application, AccountApp in a package named Persistent.

When you run the **AccountApp** application, remember to supply the `hostname` and `port` of your server machine, and use "AccountDBServer-server-scope" for the `scope`. Remember to change the parameters similarly for the applet tag in the HTML file.

13.4.3 The ActiveX client

This section explains the implementation of your ActiveX client.

13.4.3.1 Initializing

Refer back to 10.4, “ActiveX implementation” on page 157 for an explanation of how to implement your client and how to find the Business Object and invoke its methods. This section contains only the specific changes that must be made to the general case. Actually, only one line needs to be changed.

You will find all the source files for the code of this section in the following directory:

\CBConnector\13-Persistent-Object\Client\ActiveX

The scope for the Name Service is the only line in the program that you have to change. That is because we use a Location Object in order to guarantee that the different samples do not interfere. Thus, the variable to change in the General ActiveX Client is:

Parameter	Value
scope	"AccountDBServer-server-scope"

13.4.3.2 Building a client install image

Build and register your install image in the same way as for the Transient ActiveX client sample.

13.4.3.3 Using the ActiveX CD sample

When you run the ActiveX Persistent sample (accountDBX.exe), a window with an Account panel pops up.

Create a new account by typing in a number in the "Account #" box; type in an account holder and press **Create**. Do the same for a couple of more accounts.

Find one of your previously created accounts by typing in an account number and pressing **Find**.

Finally, put some money in your found account by typing in a balance in the "Update balance" box and press **changeBalance**. Make sure you put in enough money for your upcoming expenses.

13.5 What did you learn?

This chapter demonstrated the implementation and usage of a Persistent Object with implicit transaction control, where transactions are handled by CBConnector.

In the next chapter, we continue with our **Account** example using explicit transaction control, where the client is responsible for handling the transactions.

Chapter 14. Transactional Object sample

This chapter describes the Transactional Object scenario, using the policy where the client has control of the transaction scope. In our Persistent Object sample, we used the Transaction Service with implicit transaction control. This sample enhances our Persistent Object sample by using the Transaction Service with explicit control.

14.1 What you will learn

In this chapter, you will learn how to use the Transaction Service and control the scope of a transaction from your application in the CBConnector environment.

14.2 CBConnector componentry

This section briefly describes the use of the Transaction Service and how the client controls the handling of transactions.

14.2.1 Transaction Service

We have already talked about the **Transaction Service** in the previous example (see Chapter 13, “Persistent Object sample” on page 203). The Transaction Service was used in the Persistent Object sample so that for each request, an implicit transaction was started and committed. Therefore, the client had no control over when the transaction was started or committed.

In this sample, we want the client to control the transaction. The key to this is the pseudo-object **Current**, which is available to each client program (see Figure 54). The **Current** Object allows the client to start, suspend, resume, commit, and rollback transactions.

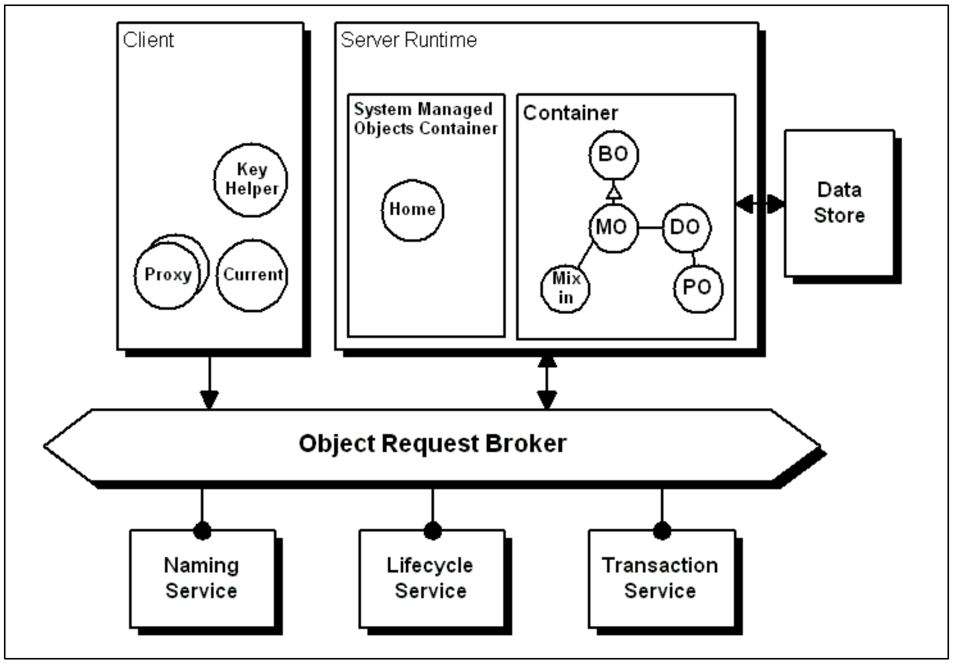


Figure 54. Transactional sample

The client program, as the originator of the transaction, uses the **Current** object to begin a transaction.

The client program then issues requests, which involve a Transactional Object.

When a request is issued to a Transactional Business Object, the transaction context is automatically propagated to the server and attached to the thread executing the method of the BO. If there is already a thread attached for a given transaction, this same thread handles all method requests.

Using the **Current** object, the client can commit or rollback the transaction.

14.3 Server

The following sections take you through all the steps you need to perform in order to build, package and install the transactional server application.

Note

All code snippets have enumerated lines for easier orientation in the code.

14.3.1 Build

Implementing the server application consists of three major steps:

1. Specify the interface and the implementation of the Business Object and Data Object, including the helper objects.
2. Create the database.
3. Generate and build the application modules.

Through the Object Builder walkthrough, we provide you with the correct parameter settings. If no suggestions are made by us for an Object Builder page, use the defaults provided by Object Builder.

14.3.1.1 Create the model

1. Create a new Object Builder model in a separate directory.
2. From **User-Defined Business Objects**, add a new file, **Account**, using the following parameters:

Parameter	Value
Name	Account
Constructs	Exception: NotEnough

3. For this file, add an interface, **Account**, with the following parameters:

Parameter	Value
Name	Account
Constructs	Exception: NotEnough

4. Add attributes to the Account Interface as described in the following table:

Define the Account Interface with the attributes as described in the following table:

Parameter	Value	Implementation
accountNumber	string<10>	Public
accountHolder	string<30>	Public
balance	double	Private

Add methods **changeBalance** and **getBalance** with the following definitions:

Method	Return Type	Parameter	Exception
changeBalance	void	anAmount double in	NotEnough
getBalance	double		

5. For the newly created Account Interface, create a Key Helper using the accountNumber as the Primary Key.
6. Create a Copy Helper with the attributes:
   - `accountNumber`
   - `accountHolder`

The attribute `balance` is not added because it is a private attribute of the object.

7. Now add an implementation of the BO. Use the following parameters:

Parameter	Value
Pattern for handling state data	Caching
Object Reference	Use lazy evaluation
Data Object Interface	Create a new one now
Implementation Inheritance	IManagedClient IManagedClient::IManageable
Implementation Language	C + +
Key Selection	AccountKey

Parameter	Value
Handle	none
Attributes to Override	none
State data	all

8. Implement the **changeBalance** method. Include the following logic.

```
if ( ( iBalance + anAmount ) > 0 )
throw NotEnough ( ) ;

iBalance += anAmount ;
```

9. Implement the **getBalance** method.

```
return iBalance ;
```

10. For the AccountDO, add a Data Object implementation with the following parameters:

Parameter	Value
Environment	BOIM with any Key.
Form of Persistent Behavior and Implementation	Embedded SQL
Data Access Pattern	Local copy (*)
Handle for Storing Pointers	Home name and Key
Implementation Inheritance	IRDBIMExtLocalToServer IRDBIMExtLocalToServer::IDataObject
Key Helper	AccountKey
Copy Helper	AccountCopy

Note

(*) - The recommended value and default is *delegating*. The matched setting in the container instance definition in *cached data in Data Object* is **Yes**. This combination locks the DO; so there is no multithreaded access on it. This configuration is also best from the performance perspective. In our sample, we just tested the possibility with *local copy*.

The Data Object and its implementation are now displayed under the **User-Defined Data Object** folder. The AccountDO in this folder is simply another view of the AccountDO displayed under the AccountBO in the **User-Defined Business Object** folder.

11. In the **User-Defined Data Object** folder, define a Schema by adding a Persistent Object and Schema to the **AccountDOImpl** object with the following parameters:

Parameter	**Value**
Group Name	AccntTX Schemas
Database	AccntDB
UserId	<your user id>
Package	AccntTX

You can leave the rest of the parameters at their defaults. The mapping between the definitions of data types in the database table and the attributes of the Persistent Object can be manipulated here.

12. Add a Managed Object to the **AccountBO**. Accept all default settings.

13. Generate the code:

   • Generate all files from User-Defined Business Objects.

   • Generate all files from User-Defined Data Objects.

14.3.1.2 Create the database

Before you can compile and link the server application, you have to create the database and database table because the SQL precompiler needs to access the database definition.

To create the database and table, follow these steps:

Open an MSDOS window and execute the `db2cmd` command.

Change the path to your working directory.

Enter:

```
Db2 create database AccntDB
Db2 connect to AccntDB
Db2 -tf Account.sql
Db2 select * from Account
Db2 connect reset
```

14.3.1.3 Build the server application

In the **Build Configuration** folder, execute the following steps:

1. Add a client DLL named AccountTXC with the following parameters:

Parameter	**Value**
Name	AccountTXC
Client Source Files	all
Libraries to Link With	none

2. Add a server DLL named AccountTXS with the following parameters:

Parameter	**Value**
Name	AccountTXS
Server Source Files	all
Libraries to Link With	AccountTXC (the client library)

3. Generate the makefiles for all targets.
4. Build all targets of the server application.

14.3.2 Package

In this section, we describe how to package the transactional account application.

14.3.2.1 Define the application

1. From the **Application Configuration** folder, add an application family named **AccountTXAppFam**.
2. For this application family, define a server application named **AccountTXAppS** with parameters as follows:

Parameter	Value
Application Name	AccountTXAppS
Initial state of application	stopped
Additional executables	none

3. From the **Container Definitions** folder, create a container instance named **ContainerOfAccountTXs** with parameters as follows:

Parameter	Value
Services	Use Transaction Services
Transaction Policies	Throw an exception and abandon the call
Passivation Policy	Passivate a component at end of transaction
Business Object	Caching
Data Object	Local Copy
use of caching service	no

4. Add a Managed Object for the server application with the following parameters:

Parameter	Value
Data Object Implementation	AccountDOImpl AccountDOImpl
Container	ContainerOfAccountTXs
Default Home	Default Home
Home Name	BOIMHomeOfRegHomes
Name in Factory Finding Service Registry	AccountMOFactory
Name in Naming Service Registry	AccountMOHome
Locatable by Cell	yes
Locatable by Workgroup	yes
State of Home	exists

14.3.2.2 Create an installation image

From the AccountTXAppFam, generate the installation scripts. This step also creates the System Management data in the form of a DDL file.

14.3.3 Install

To install and configure the server application, follow these steps:

1. Install the server application on your server machine.
2. Configure an **AccountTXServer** in a new Management Zone, **AccountTX Management Zone**. Use a new configuration, **AccountTX Configuration**, for this server.

   When you install the server application, you will see that an application named Specific AccountTXAppS is created in addition to the AccountTXAppS application. When you configure the AccountTXAppS application, Specific AccountTXAppS and the default DB2 application named iDB2IMServices, need to be configured to the AccountTXServer as well.

   The Specific AccountTXAppS application is found under **Host Images -> Application Family Installs -> SpecificAccountTXAppFam**. The iDB2IMServices application is found under **Host Images**. Application Family Installs, iDB2IMApplications.

Note: For this sample, you must use exactly this server name for the application server to enable the execution of the client programs, because the Location Object you installed refers to it.

3. Ensure that you bound the *AccountPO.bnd* bind file to your database, AccntDB. If you did not, issue following set of commands from the DB2 command line.

```
DB2 connect to AccntDB
DB2 bind AccountPO.bnd
DB2 connect reset
```

4. Activate the configuration.

   Note: In CBConnector Release 1.2, the server will start automatically when you activate the configuration.

   This scenario uses the DB2 Application Adaptor; therefore, stop the server and set the **open string** with the database administrator user ID and password for the databases in the **XA Resource Manager images** of this server image.

5. Start the server with `Run Immediate`.

14.4 Client

These sections describe what is different in the client applications as related to the general descriptions given in Chapter 10, “Sample client development approach” on page 117.

14.4.1 The C++ client

We remember that the **acctproxy** class is the application-dependent class, a subclass of the **cbcproxy** class, which hides the CBConnector specifics from the client application.

14.4.1.1 The acctproxy

This implementation of **acctproxy** is the same as described in 10.2, “C++ Implementation” on page 121.

14.4.1.2 The GUI

This GUI is at first glance quite different from the persistent sample GUI, but at a closer look, you will see that the Transactional GUI basically is two persistent GUIs with some buttons in the middle.

The functionality of the GUI is to find two accounts and to transfer an amount of money from one account to the other. In addition, we are showing off the Transactional Services, by letting the user begin a transaction, perform updates to the database, and choose to commit or rollback the changes.

The GUI has one **AcctCBCProxyWrapper** object, to which both Account panes have a pointer. To be able to do any changes, the user clicks the begin button, which in t urn calls the **beginTX** method on the **acctproxy** object. The user may then find or create and update accounts as in the persistent sample. When two accounts are active, the user may type in an amount in the middle entry field and choose to transfer the amount from one account to the other (the arrows indicate the flow of money).

The transfer is done by calling the **changeBalance** method to the two **Account** objects, with the negated amount for the source **Account**. After updates have been performed, the user can end the transaction scope by choosing to commit or rollback the changes. To perform further changes, the user must start a new transaction by pressing the **Begin** button.

To implement this GUI, two Account panes are added, and the necessary functionality to access the transaction methods from the **acctproxy** is implemented in the **AcctCBCProxyWrapper**.

The old **acctproxy** is replaced with the **acctproxy** object generated for this sample, as well as the interface header files and the client DLL generated by the Object Builder for the persistent sample.

Run the client by typing the following command from the *VisualC++* directory:

```
AccountTX.exe
```

14.4.2 The Java client

All Java source and class files for this sample are located on the CD in the directory *Samples\14-Transactional-Object\Client\Java*.

14.4.2.1 Generate client proxies

For this sample, you only need to generate the proxies for the **Account**. Generate the Java client proxies for these IDL files as described in 6.7.2, “Java client programming model” on page 76.

If you have already done this for the previous transient and persistent \CBConnector\14-Transactional-Object\Client\ActiveX, you can use those. The **Account** Interface has not changed for any of these samples, although the Managed Objects created with the server applications are treated quite differently.

14.4.2.2 The GUI

This GUI is at first glance quite different from the persistent sample GUI, but at a closer look, you will see that the Transactional GUI basically is two persistent GUIs (GUIviews.AccountViews) with some buttons in the middle.

The functionality of the GUI is to find two **Accounts** and to transfer an amount of money from one account to the other. In addition, we are showing off the Transactional Services, by letting the user begin a transaction, perform updates on the **Accounts**, and choose to either commit or rollback the changes.

The GUI has one **AccountProxy**, that both **AccountViews** use. To be able to do any changes, the user clicks the **Begin** button, which in turn calls the **beginTX** method on the **AccountProxy**. The user may then find, create, or update **Accounts** as in the persistent sample. When two accounts are active, the user may type in an amount in the middle entry field and choose to transfer the amount from one account to the other (the arrows indicate the flow of money). The transfer is done by calling the **changeBalance** method to the two **Account** objects, with the negated amount for the source **Account**. After updates have been performed, the user can end the transaction by choosing to commit or rollback the changes. To perform further changes, the user must start a new transaction by pressing the **Begin** button.

14.4.2.3 Run the client

You can run this client as an application and as an applet, as you have in the previous samples.

To run the client application, execute the command:

```
java AccountApp <hostname> <port> AccountTXServer-server-scope
```

To run the client as an applet, embed the applet in an HTML page (*Transactional.html*, and open that page in the applet viewer with the command below:

```
appletviewer Transactional.html
```

If you use the sample command (CMD) files provided on the CD, remember to change the `hostname` and `port` parameters (in *runApplet.cmd* and *Transactional.html*) to reflect your bootstrap server.

14.4.3 The ActiveX client

Refer back to 10.4, “ActiveX implementation” on page 157, for an explanation on how implement your client and how to find the Business Object and invoke its methods. This section contains only the specific changes that have to be made to the general case. The differences come from the addition of a transactional part.

\CBConnector\14-Transactional-Object\Client\ActiveX

14.4.3.1 Initializing

Initialization of the client is similar to the general ActiveX client. The table below summarizes the changes to the variables in order to get the transactional sample working.

Variable	Value
scope	"AccountTXServer-server-scope"
interfaceName	"Account.object interface"
appName	"Account"

14.4.3.2 Initialize a transaction

Before starting a transaction, you need to initialize a Transaction Object.

```
Dim CosTransaction_Current As Object
Dim CCurrent As Object
Set CosTransaction_Current=
            CreateObject("IDL:costransactions.Current")
Set CCurrent=orb.get_current("CosTransactions::current")
CosTransaction_Current.narrow CCurrent
```

14.4.3.3 Begin the transaction

The **begin** starts the transaction and makes it possible to do a rollback on the changes you did on the data of the object.

```
Dim e As Variant

CosTransactions_Current.begin e
If e.EX_majorCode > 0 Then
  ...
End If
```

14.4.3.4 Commit the transaction

The **commit** ends the transaction and makes the changes on the data permanent. Call your Business Object methods between the **begin** and **commit** statements in the same way as for the general ActiveX client.

```
Dim e As Variant

CosTransactions_Current.commit e
If e.EX_majorCode > 0 Then
  ...
End If
```

14.4.3.5 Rollback the transaction

The **rollback** sets the data to the initial state at the **begin** of the transaction.

```
Dim e As Variant

CosTransactions_Current.rollback e
If e.EX_majorCode > 0 Then
  ...
End If
```

14.4.3.6 Building a client install image

Build and register your install image in the same way as for the Transient ActiveX client sample.

14.4.3.7 File cross reference

The table below references functionalities of the files used in the transactional client scenario:

File	Description
Account.dll	Dynamic Link Library containing Client runtime bindings
AccountKey.dll	Dynamic Link Library containing runtime bindings for Key Proxy
AccountCopy.dll	Dynamic Link Library containing runtime bindings for Key Proxy
register.bat	Batch file used to register the Account, AccountKey, and AccountCopy DLLs.
unregister.bat	Batch file used to unregister the Account, AccountKey, and AccountCopy DLLs.

14.4.3.8 Using the ActiveX CD sample

When you run the ActiveX transaction sample (AccountTXX.exe) a window with two Account panels (Account 1 and Account 2) pops up. Since this is a transaction sample, you begin, commit, and rollback the transactions manually.

Type in an account number and a name for Account 1. Press **Begin**, press **Create** for Account 1 and press **Commit**. You have now a new account in your database. Repeat the above steps for another account number and name.

Let's find an account. Type the first of the previous account numbers in Account 2, press **Begin**, **Find** for Account 2 and **Commit**. Voila'. "Account 2" is updated.

Let's put some money in Account 1 (you should have an account displayed there). Type in a balance (make it large enough); press **Begin** and **Commit**. Account 1's balance is updated.

You can now transfer some money to Account 2 as follows: Type in an amount in the middle box below the **>>** button. Press **Begin**, **>>** (transfer) and **Commit**. The amount has now been transferred from Account 1 to Account 2 — likewise for the **<<** button (transfer in the other direction). If you decide not to commit a transaction, you can always do a rollback.

14.5 What did you learn?

This chapter demonstrated the use of the Transaction Service. This service provides simple interfaces that allow an application to determine the scope of a transaction and resources to take part in a transaction.

To change the Business Objects into Transactional Objects is only a matter of configuration to the right Business Object Application Adaptor.

Chapter 15. Object with UUID Key and model dependency

This chapter introduces the use of a UUID Key and the *linking* together of Object Builder models to create a server application.

You can find the Object Builder model, code and client applications for this sample on the CD-ROM for this book in the directory *\CBConnector\15-UUID-Key-Model-Dependency*.

15.1 What you will learn

You will learn how to use the UUID Key (Universally Unique Identifier) as a Primary Key to an object, as well as how to *link* to another Object Builder object model; so you can use the Managed Objects defined there in your own model.

You will learn how to create an **AccountManager** which itself acts as a client to the transactional **Account** defined in the previous chapter. The **AccountManager** will handle the transferring of some amount of money from one **Account** to another, inside of a transaction. In the previous sample, you saw how client programs started a transaction, changed the balances of two Accounts, then either committed or rolled back the transaction. By using an **AccountManager**, you will see how the client program only has to call a single method to perform the same actions in the server process. This lightens the work of the client programmer and enhances runtime performance by making only one call over the ORB.

15.2 CBConnector componentry

Figure 55 shows the main components used in our Event Service Account scenario.

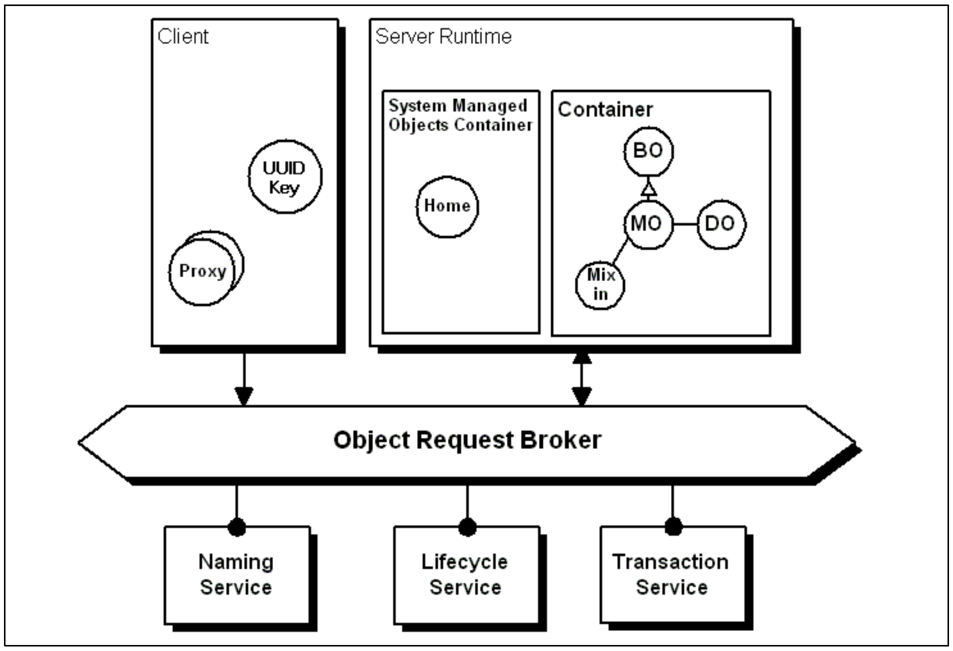


Figure 55. UUID key sample components

15.2.1 UUID Key

In the previous samples, you have created Managed Objects that are unique from one another depending on their Primary Keys. The attribute(s) contained in the Primary Key are usually part of the essential state of the Business Object, which are made persistent by a back-end data store. These Primary Keys are logically needed, for example, to distinguish one **Account** from another. But what if you need to create a Transient Object that has no essential state? What do you use for a Primary Key? This is where something called a UUID Key comes in.

A UUID (Universally Unique Identifier) is defined as "a value constructed with an algorithm that provides a reasonable assurance that the identity value is unique within the known universe." So a UUID Key is a Primary Key that is generated for you and guaranteed to be unique.

Using a UUID Key makes the most sense for Transient Objects that really have no logical reason for being unique, but each client program that creates one must have its own instantiation of that object on the server. In this sample, we will demonstrate this with the **AccountManager**.

15.3 Server

This section takes you through all the steps you need to perform to build, package, and install a server application that uses a UUID Key for its Primary Key.

Note

For your convenience, all code snippets used in this sample are on the CD located in the directory *\CBConnector\15-UUID-Key-Model-Dependency\Server\code_snippets.cp*

15.3.1 Build

Implementing the server application consists of two major steps:

1. Specify the interface and implementation of the Business Object, Data Object, and the Key and Copy Helpers.
2. Generate and build the application.

Through the Object Builder walkthrough, we provide you with the correct parameter settings for the SmartGuides. If no suggestions are made by us for a SmartGuide page, use the default settings.

We strongly recommend that you open help for each field of the SmartGuide pages in order to understand more of the dialectic behind the input fields.

15.3.1.1 Create the model

1. Create a new Object Builder model in a separate directory. In the **Project Dependencies** page of the *Open Project-SmartGuide*, specify the model you created for the sample in Chapter 14, “Transactional Object sample” on page 217, or specify the model provided for that sample on the CD in the directory *\CBConnector\15-UUID-Key-Model-dependency\depends on\Transactional\Server*.

   You will see under **User-Defined Business Objects** folder, the *Account* file and interface defined in the "depended on" model. These definitions are visible as read-only.

2. From **User-Defined Business Objects**, add a new file, *AccountManager*, using the following parameters:

Parameter	Value
Name	AccountManager
Constructs: Exception	TransferFailed Members: message string <30>
Files to Include	IManagedClient (default) Account AccountKey AccountCopy

3. For this file, add an interface, **AccountManager**, with the following parameters:

Parameter	Value
Name	AccountManager
Constructs: Exception	TransferFailed Members: message string <30>
Interface Inheritance	IManagedClient IManagedClient::IManageable

Add a method, **transferAmount** with the following definition:

Method	Return Type	Parameters	Exception
transferAmount	void	amount double in sourceAccountNumber string <10> In destAccountNumber	TransferFailed

4. Add an implementation of the BO. Use the following parameters:

Parameter	Value
Pattern for handling state data	Caching
Object Refercence	Use lazy evaluation
Data Object Interface	Create a new one now
Implementation Inheritance	IManagedClient IManagedClient::IManageable
Implementation Language	C + +
Key Selection	none
Handle	none
Attributes to Override	none
Methods to Override	none
Data Object File Name	AccountManagerDO
Data Object Interface Name	AccountManagerDO
State data	none

5. Implement the **transferAmount** method.
   The code for this method is on the CD in the file \CBConnector\15-UUID-Key-Model-Dependency\Server\code_snippets.cp p. The following code snippet is only an outline of the logic implemented by this method. It does not show the necessary variable definitions, or error-checking and handling:

```
//*** resolve the orb ***
 orb = CBSeriesGlobal::orb();

 //*** resolve the Name Service ***
 nameService = CBSeriesGlobal::nameService();

 //*** resolve the Factory Finder ***
 obj = nameService->resolve_with_string(
          "/host/resources/factory-finders/AccountTXServer-server-scope");

 factoryFinder = IExtendedLifeCycle::FactoryFinder::_narrow(obj);

 //***  resolve the Account home ***
 obj = factoryFinder->find_factory_from_string("Account.object interface");
 home = IManagedClient::IHome::_narrow(obj);

 //*** get the current transaction ***
 orbCurrentPtr = orb->get_current("CosTransactions::Current");
 currentTransaction = CosTransactions::Current::_narrow( orbCurrentPtr );
 currentTransaction->set_timeout(360);

 //*** Begin the transaction ***
 currentTransaction->begin();

 //*** Find the source Account to transfer the amount from ***
 key = AccountKey::_create( );
 key->accountNumber( sourceAccountNumber );
 keyString = key->toString( );
 mo = home->findByPrimaryKeyString(*keyString);
 sourceAccount = Account::_narrow( mo );

 //*** Find the destination Account to transfer the amount to ***
 key = AccountKey::_create( );
 key->accountNumber( destAccountNumber );
 keyString = key->toString( );
 mo = home->findByPrimaryKeyString(*keyString);
 destAccount = Account::_narrow( mo );

 //*** subtract amount from the source Account ***
 sourceAccount->changeBalance( amount * -1 );

 //*** add amount to the desination Account ***
 destAccount->changeBalance( amount );

 //*** commit the transaction ***
 currentTransaction->commit(1);
```

6. For the AccountManagerDO, add a Data Object implementation with the following parameters:

Parameter	Value
Environment	BOIM with UUID Key
Form of Persistent Behavior and Implementation	Transient
Handle for Storing Pointers	Home name and Key
Key Helper	none
Copy Helper	none

7. Add a Managed Object to the **AccountManagerBO**. Accept all default settings.
8. Generate the code for all files from **User-Defined Business Objects**.

15.3.1.2 Build the server application

In the **Build Configuration folder**, execute the following steps:

1. Add a client DLL named AccountManagerUUIDC with the following parameters:

Parameter	Value
Name	AccountManagerUUIDC
Client Source Files	AccountManager
Libraries to Link With	AccountTXC (depended on library)

2. Add a server DLL named AccountManagerUUIDS with the following parameters:

Parameter	Value
Name	AccountManagerUUIDS
Server Source Files	all AccountManager files
Libraries to Link With	AccountManagerUUIDC (the client library) AccountTXC (depended on library)

3. Generate the makefiles for all targets.
4. Copy the following files from the *Working* directory of the "depended on" sample into your current *Working* directory

   Because the **AccountManager** acts as a client to the **Account** object, you need the normal client files associated with **Account**:

   - Account.hh
   - AccountKey.hh
   - AccountCopy.hh
   - AccountTXC.lib

In CBConnector Release 1.2, the linking of Object Builder models doesn't give you everything you need from the model. In this sample, when you generated the code for your Business Objects, the following two files were not generated:

- AccountKey.idl
- AccountCopy.idl

5. Build all C++ out-of-date targets of the server application.

15.3.2 Package

In this section, we describe how to package the server application. This includes the following steps:

1. Definition of the application using the Object Builder.
2. Creation of an installation image using InstallShield.

15.3.2.1 Define the Application

1. From the **Application Configuration** folder, add an application family named **AccountManagerUUIDAppFam**.
2. For this application family, add a server application named **AccountManagerUUIDAppS** with parameters as follows:

Parameter	**Value**
Application Name	AccountManagerUUIDAppS
Initial state of application	stopped
Additional Executables	AccountTXC.dll (depended on DLL) AccountManagerUUIDC.dll (client DLL)

You don't normally need to include the client DLL for the server application because it is included by default to the installation script. However, in CBConnector Release 1.2, you need to specifically include the client DLL when you have a Managed Object that uses a UUID Key.

3. Add a Managed Object for the server application with the following parameters:

Parameter	**Value**
Managed Object	AccountManagerMO AccountManagerMO
Primary Key	none
Copy Helper	none
Data Object Implementation	AccountManagerDOImpl AccountManagerDOImpl
Container	CachedTransientObjects
Default Home	Default Home
Home Name	BOIMHOmeOfRegHomes
Name in Factory Finding Service Registry	AccountManagerMOFactory
Name in Naming Service Registry	AccountManagerMOFactory

15.3.2.2 Create an installation image

1. From the AccountManagerUUIDAppFam, generate the installation scripts.
2. Build the installation image.

15.3.3 Install

1. Install the server application on your server machine.
   Configure an **AccountManagerUUIDServer** in a new Management Zone, **AccountManagerUUID Management Zone**. Use a new configuration, **AccountManagerUUID Configuration**, for this server.

   Note: For this sample, you must use exactly this server name for the application server to enable the execution of the client programs because the single Location Object you installed refers to it.

   Alternately, you could configure the application on the **AccountTXServer** used for the transactional Account application. This would be a more logical configuration since the **AccountManager** uses this application.

2. Activate the configuration.

15.4 Client

These sections describe what is different in the client applications, as related to the general descriptions given in Chapter 10, “Sample client development approach” on page 117.

15.4.1 The C++ client

We remember that the **acctproxy** class is the application-dependent class, a subclass of the **cbcproxy** class, which hides the CBConnector specifics from the client application.

15.4.1.1 The acctproxy

In this sample, we only provide a single method, **createBO**, in our . **acctproxy** class (in addition to the usual **initCBC** method, with its hardcoded scope and interface information). In Figure 56 we show the implementation of the **createBO** method.

```
/*
;------------------------------------------------------------------;
;       create the business object                                 ;
;------------------------------------------------------------------;
*/
AccountManager_ptr _Export custproxy::createBO()
  {
IBOIMExtLocal::IUUIDPrimaryKey_ptr UUIDKey;
IManagedClient::IManageable_ptr    mo = NULL;
ByteString                         *keyString;
/*
;------- create key from UUID --------------------------------------;
*/
   UUIDKey = IBOIMExtLocal::IUUIDPrimaryKey::_create();
   try
     {
      UUIDKey->generate();
      keyString = UUIDKey->getUuid();
     }
   catch(IManagedClient::IInvalidKey)
     {
      cout << "Creating AccountKey failed - invalid key" << endl;
     }
   catch( ... )
     {
      cout << "Creating AccountKey failed" << endl;
      return NULL;
     }
   cout << "Key object OK" << endl;
/*
;------- create Managed Object -------------------------------------;
*/
   mo = createMO(*keyString);
   if (CORBA::is_nil(mo))
     return NULL;
/*
;------- narrow to business object ---------------------------------;
*/
   try
     {
      return AccountManager::_narrow(mo);
     }
   catch( ... )
     {
      cout << "narrow Managed Object failed" << endl;
      return NULL;
     }
  }
```

Figure 56. createBO

15.4.1.2 The GUI

The GUI looks just like the transactional sample GUI, without the buttons to begin, commit, and rollback. In this sample, there is a new Business Object, the **AccountManager**, that manages the transfer of money for you.

The **AccountManager** implements the method **transferAmount**, which takes the amount to transfer, the source account number and the target account number as inputs. When the source and target accounts have been found, one method call to the **AccountManager** is all you need. Within this method, a transactional scope is started, and the **changeBalance** method is invoked on both the accounts with the amount of money to be transferred. All this had to be done explicitly in the GUI for the transactional sample.

We do need two new wrapper classes, though, the **AccountMgrWrapper**, and the **AcctMgrCBCProxyWrapper**

The **AcctMgrCBCProxyWrapper** finds the home, and creates an **AccountMgrWrapper** object. We only need one such object since it only handles the one **transferAmount**.

As before, we include the **AcctCBCProxyWrapper** and a variable for an **AccountWrapper**. In our AccountView, we also needed to wrap the **createBO**, **findBO** and update methods within a transaction scope, since these methods are not available from the client in this sample.

Run the client by typing the following command in the *VisualC++* directory:

```
UuidC.exe
```

15.4.2 The Java client

All Java source and class files for this sample are located on the CD in the directory *\CBConnector\15-UUID-Key-Model-dependency\depends on\Transactional\Server\Client\Java*.

15.4.2.1 Generate Client Proxies

For this sample, you need to generate the proxies for the **AccountManager** Interface. The **AccountManager** uses the **Account** Interface; so the *Account.idl* file must be in the same directory as *AccountManager.idl* when you generate the proxies.

Generate the Java client proxies for the file *AccountManager.idl* as described in 6.7.2, “Java client programming model” on page 76.

15.4.2.2 The AccountManagerProxy Class

The **AccountManagerProxy** class is similar to **AccountProxy** class in the way it is initialized, except that it, obviously, creates **AccountManager** objects. **AccountManager** objects are created with a UUID Key as you know from implementing the server application for this sample. So, this class provides the method **createFromUUIDKey**, which generates a UUID Key, then creates a Managed Object as if it were using a normal Primary Key. See Figure 57.

```
/**
 * createFromUUIDKey - creates an AccountManager
 *                   from a generated UUID Key
 */
public AccountManager createFromUUIDKey ( ) {

  AccountManager accountManager = null;

  try {
    // create the UUID Key
    IUUIDPrimaryKey UUIDKey = IUUIDPrimaryKeyHelper._create();
    UUIDKey.generate();
    byteçŸ UUIDKeyString = UUIDKey._toString();

    org.omg.CORBA.Object o = null;

    // createMO of CBProxy calls createFromPrimaryKeyString()
    try {
      o = createMO(UUIDKeyString);
    }
    catch (IInvalidKey e) {
      o = null;
    }
    catch (com.ibm.IManagedClient.IDuplicateKey e) {
      // this should never happen
      o = null;
    }
    catch (NullPointerException e) {
      o = null;
    }

    if (o == null) {
      throw new NullPointerException();
    }

    // narrow to an AccountManager
    accountManager = AccountManagerHelper.narrow(o);

    if (accountManager == null) {
      throw new NullPointerException();
    }
  }
  catch(Throwable e) {
    accountManager = null;
  }

  return accountManager;
}
```

Figure 57. Creating an AccountManager from a UUID Key

15.4.2.3 The GUI

The GUI for this sample is similar to the Transactional sample, except in this one, it isn't up to the user to control the transaction. The transaction is taken care of by an **AccountManager**. When the user transfers money from one **Account** to another, the **transferAmount** method of the **AccountManager** is called.

15.4.2.4 Run the client

You can run this client as an application and as an applet, as you have in the previous samples.

To run the client application, execute the command:

```
java AccountManagerApp <hostname> <port> AccountTXServer-server-scope
AccountManagerUUIDServer-server-scope
```

To run the client as an applet, embed the applet in an HTML page (*Transactional.html*), and open that page in the applet viewer with the command:

```
appletviewer UUIDKey.html
```

If you use the sample command (CMD) files provided on the CD, remember to change the `hostname` and `port` parameters (in *runApplet.cmd* and *Transactional.html*) to reflect your bootstrap server.

15.5 What did you learn?

In this chapter, you learned how to use a UUID Key and were shown a case where it makes sense to use one. You also learned how to "link" or "depend on" another Object Builder object model to use its objects' definitions in your model.

Chapter 16. Event Service

This chapter enhances the transient Account Object with the use of the Event Service.

You can find the Object Builder model, code, and client applications for this sample on the CD-ROM for this book in the directory *\CBConnector\16-Event-Service*.

16.1 What you will learn

In this chapter, you will learn how to use the Event Service to create and send events from a server application and how to receive events from a client.

16.2 CBConnector componentry

Figure 58 shows the main components used in our Event Service Account scenario.

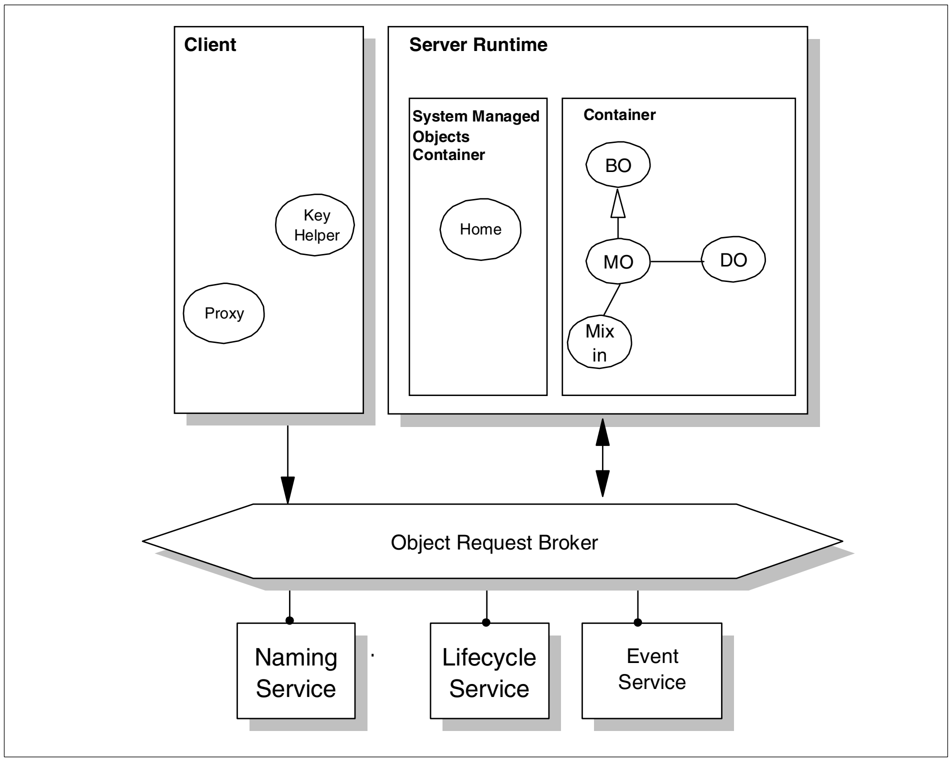


Figure 58. Event sample components

16.2.1 Event Service

Event channels provide a loosely coupled way to communicate with other objects sharing the same event channel. Event channels support two delivery models of events:

- Push model
- Pull model
- Event channels actually consist of several components:
- EventChannel
- ProxyPushSupplier (a counterpart for push consumer)
- ProxyPushConsumer (a counterpart for push supplier)
- ProxyPullSupplier (a counterpart for pull consumer)
- ProxyPullConsumer (a counterpart for pull supplier)
- SupplierAdmin object
- ConsumerAdmin object

See Figure 59 below.

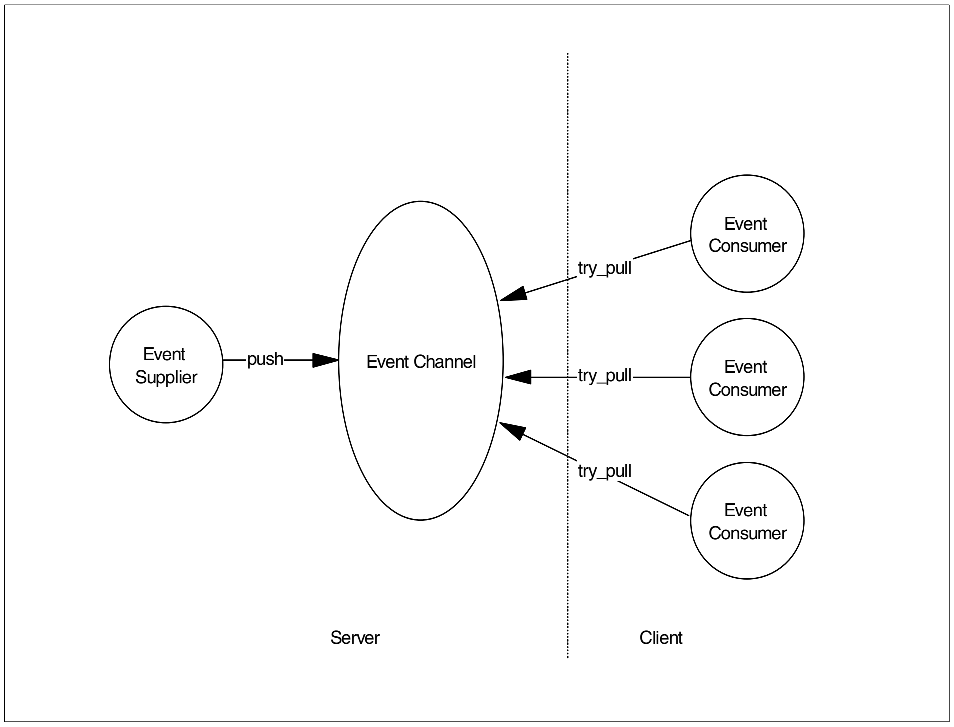


Figure 59. Event channel

When you connect to an event channel, you must decide the delivery policy (push or pull) of events and whether you want to consume and/or supply events. The event channel provides the Administration Object to obtain the counterpart with which to communicate. All the events sent to the event channel are multifaceted or queued (pull model) to all participants of a given channel. Delivery to actual consumers depends on the chosen model. If the consumer uses the push model, the events are delivered automatically to the consumer through a Callback Object. If the pull model is used, the events must be pulled from the channel.

In this chapter, we use the cell default event channel to push notifications to consumers. When the balance of the account is changed, the object (such as the supplier) pushes an event to the channel containing the number of the changed account, the name of the account holder, and the new balance. Clients can then pull the event from the channel and process the event accordingly.

The clients in this sample use the **try_pull** method to peek at the channel to see if there are events in the queue. If there are no events in the channel, the in/out Boolean parameter passed into this method is set to `false` by the channel and the event returned is empty. If the **pull** method is used, the channel will block the client until there is an event available.

16.3 Server

This section takes you through all the steps you need to perform to build, package, and install a Transient Object server application that places events on the event channel.

16.3.1 Build

Implementing the server application consists of two major steps:

1. Specify the interface and implementation of the Business Object, Data Object, and the Key and Copy Helpers.
2. Generate and build the application.

Through the Object Builder walkthrough, we provide you with the correct parameter settings for the SmartGuides. If no suggestions are made by us for a SmartGuide page, use the default settings.

We strongly recommend you open help for each field of the SmartGuide pages in order to understand more of the dialectic behind the input fields.

16.3.1.1 Create the model

1. Create a new Object Builder model in a separate directory.
2. From **User-Defined Business Objects**, import the IDL files *CosEventComm.idl* and *CosEventChannelAdmin.idl* provided with the Component Broker Toolkit. These files will be located in the directory *C:\CBroker\include* (where *C:\CBroker* is the directory where you installed CBConnector).

   These files define the interfaces you need to implement an event push supplier. In CBConnector Release 1.2, the Object Builder does not have tool support for implementing the Event Service; so we must import these files ourselves.

3. From **User-Defined Business Objects**, add a new file, *Account*, using the following parameters:

Parameter	Value
Name	Account
Constructs: Exception	NotEnough
Files to Include	IManagedClient (default) CosEventComm CosEventChannelAdmin

4. For this file, add an interface, **Account**, with the following parameters:

Parameter	Value
Name	Account
Constructs: Exception	NotEnough
Interface Inheritance	IManagedClient IManagedClient::IManageable (default) CosEventComm CosEventComm::PushSupplier

Add attributes to the Account Interface as described in the following table:

Parameter	Type	Implementation
accountNumber	string<10>	Public
AccountHolder	string<30>	Public
balance	double	Private
pushConsumer	CosEventComm:: PushConsumer	Private

Add methods, **changeBalance** and **getBalance** with the following definitions:

Method	Return Type	Parameter	Exception
changeBalance	void	anAmount double in	NotEnough
getBalance	double		

5. For the newly created Account Interface, add a Key Helper using the accountNumber as the Primary Key.
6. Add a Copy Helper with the attributes:
   - `accountNumber`
   - `accountHolder`

   The `balance` attribute is not added because it is a private attribute of the object.
7. Now add an implementation of the BO. Use the following parameters:

Parameter	Value
Pattern for handling state data	Caching
Object Reference	Use lazy evaluation
Data Object Interface	Create a new one now
Implementation Inheritance	IManagedClient IManagedClient::IManageable
Implementation Language	C + +
Key Selection	AccountKey
Handle	none
Attributes to Override	none
Methods to Override	PushSupplier:: disconnect_push_supplier
Data Object File Name	AccountDO
Data Object Interface Name	AccountDO

Parameter	Value
State data	accountNumber accountHolder balance

8. Implement the **changeBalance** method. The implementation of the BO methods in this sample will written in C++. Include the following logic:

```
if ( ( iBalance + anAmount ) < 0 )
   throw NotEnough ( ) ;

iBalance += anAmount ;

if ( ! CORBA::is_nil ( iPushConsumer ) )
{
    char message [100];
    sprintf(message, "%s:%s:%.2f", accountNumber(), accountHolder(), iBalance) ;

    CORBA::Any event ;
    event <<= CORBA::string_dup (message ) ;
    iPushConsumer->push ( event ) ;
}
```

9. Implement the **getBalance** method:

```
return iBalance;
```

10. Implement the **disconnect_push_supplier** method that you have overridden:

```
if( ! CORBA::is_nil ( iPushConsumer ) )
    iPushConsumer->disconnect_push_consumer ( ) ;
```

11. Now you need to enhance the implementations of the framework methods **initForCreation**, **uninitForDestruction**, **externalize_to_stream**, and **internalize_from_stream**. In order to change these implementations, you must first open the properties of these methods under the **Framework Methods** folder and select **Use the implementation defined in the editor pane**.

12. Enhance the implementation of the framework method, **initForCreation**, by appending the following code snippet:

```
IExtenderNaming::NamingContext_var rootNC ;"
CosEventChannelAdmin::EventChannel_var channel ;"
CosEventChannelAdmin::SupplierAdmin_var sadm ;"
CosEventChannelAdmin::ProxyPushConsumer_var ppc ;"
CORBA::Object_var tmp ;"
const char* channelName ="
  "/.:/resources/event-channels/cell-default" ;"

rootNC = CBSeriesGlobal::nameService ( ) ;
if( ! CORBA::is_nil ( rootNC ) )
{
  try
  {
      tmp = rootNC->resolve_with_string( channelName ) ;
      channel = CosEventChannelAdmin::EventChannel::_narrow(tmp) ;
      sadm = channel->for_suppliers ( ) ;
      iPushConsumer = sadm->obtain_push_consumer( ) ;
    ppc = CosEventChannelAdmin::ProxyPushConsumer::_narrow (iPushConsumer)
;
     ppc->connect_push_supplier( this ) ;
     }
     -
     catch( CosNaming::NamingContext::NotFound& e)
     {
         // We Do not want initializeState to throw exception
    }
    catch( CosNaming::NamingContext::CannotProceed& e )
    {
        // We do not want initializeState to throw exception
        }
}
```

13. Enhance the implementation of the framework method, **uninitForDestruction**, with the following code:

```
if( ! CORBA::is_nil ( iPushConsumer ) )
    iPushConsumer->disconnect_push_consumer( );
```

14. Remove the following code from the framework method, **externalize_to_stream**:

```
CosEventComm::PushConsumer_var PushConsumerAsObject =
            iDataObject->pushConsumer( ) ;
CORBA::String_var PushConsumerAsString =
            CBSeriesGlobal::orb( ) ->object_to_string(PushConsumerAsObject) ;
targetStreamI0->write_string(PushConsumerAsString) ;
```

15. Remove the following code from the framework method, **internalize_from_stream**:

```
CORBA: :String_var PushConsumerAsString = sourceStreamI0->read_string( ) ;
CORBA: :Object_var PushConsumerAsObject =
  CBSeriesGlobal: :orb ( ) - >string_to_object (PushConsumerAsString) ;
CosEventComm: :PushConsumer_var PushConsumerAsObjectTemp (
   CosEventComm: :PushConsumer: :_narrow ( PushConsumerAsObject) ) ;
 iDataObject->pushConsumer (PushConsumerAsObjectTemp ) ;
```

16. Under the **File Adornments** folder in the Methods panel, select **AccountBOPrologue**. This is where you can specify any include statements your methods may need to resolve exterior class definitions. Append the following include statement:

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

17. For the AccountDO, add a Data Object implementation with the following parameters:

Parameter	**Value**
Environment	BOIM with any Key
Form of Persistent Behavior and Implementation	Transient
Handle for Storing Pointers	Home name and Key
Key Helper	AccountKey
Copy Helper	AccountCopy

18. Add a Managed Object to the **AccountBO**. Accept all default settings.

19. Generate the code for all files from **Account** under **User-Defined Business Objects**. *Do not* generate the **CosEventComm** and **CosEventChannelAdmin** code because the implementation and header files for these two interfaces are provided by the Component Broker Toolkit. If you generate these from this model, you will have compile errors because local header files will be found during the build rather than the correct ones provided.

16.3.1.2 Build the server application

In the **Build Configuration folder**, execute the following steps:

1. Add a client DLL named AccountEVC with the following parameters:

Parameter	Value
Name	AccountEVC
Client Source Files	Account AccountKey AccountCopy
Libraries to Link With	none

2. Add a server DLL named AccountEVS with the following parameters:

Parameter	Value
Name	AccountEVS
Server Source Files	all
Libraries to Link With	AccountEVC (the client library)

3. Generate the makefiles for all targets.
4. In CBConnector Release 1.2, because of the way Object Builder generates the server DLL makefile for this sample, you need to edit the *AccountEVS.mak* file located in your *Working* directory. Open this file in an editor, and delete all references to *CosEventComm.idd* and *CosEventComm.hhd*.
5. Build all C++ out-of-date targets of the server application.

16.3.2 Package

In this section, we describe how to package the server application. This includes the following steps:

1. Definition of the application using the Object Builder.
2. Creation of an installation image using InstallShield.

16.3.2.1 Define the application

1. From the **Application Configuration** folder, add an application family named **AccountEVAppFam**.

2. For this application family, add a server application named AccountEVAppS, with parameters as follows:

Parameter	Value
Application Name	AccountEVAppS
Initial state of application	stopped
Additional Executables	none

3. Add a Managed Object for the server application with the following parameters:

Parameter	Value
Managed Object	AccountMO AccountMO
Primary Key	AccountKey AccountKey
Copy Helper	AccountCopy AccountCopy
Data Object Implementation	AccountDOImpl AccountDOImpl
Container	Default Home
Home Name	BOIMHomeOfRegHomes
Name in Factory Finding Service Registry	AccountMOFactory
Name in Naming Service Registry	AccountMOHome

16.3.2.2 Create an installation image

From the AccountEVAppFam, generate the installation scripts. This step also creates the System Management data in form of a DDL file.

16.3.3 Install

To install and configure the server application, follow these steps:

1. Install the server application on your server machine.
2. Configure an **AccountEVServer** in a new Management Zone, **AccountEV Management Zone**. Use a new configuration, **AccountEV Configuration**, for this server.

   You also need to add the following applications to the AccountEV Configuration:

   iDefaultCellEVentChannel and **iEventService**

   Note: For this sample, you must use exactly this server name for the application server to enable the execution of the client programs because the single Location Object you installed refers to it.
3. Activate the configuration.

16.4 Client

These sections describe what is different in the client applications, as related to the general descriptions given in Chapter 10, “Sample client development approach” on page 117.

16.4.1 The C++ client

This is a simple C++ client that pulls from the event channel. To run this client, run the executable \CBConnector\16-Event-Service\Client\SimpleCpp\EventApp.exe.

You need to run the Java client at the same time so you can update balances of **Accounts** and see events being logged.

16.4.2 The Java client

All Java source and class files for this sample are located on the CD-ROM in the directory *\CBConnector\16-Event-Service\Client\Java*.

16.4.2.1 Generate client proxies

Generate the **Account** proxies as described in 6.7.2, “Java client programming model” on page 76.

16.4.2.2 Event Service helper classes

To use the Event Service in this sample, we introduce three new classes:

1. EventHelper
2. EventThread
3. AccountEventHelper

First, we have two generic classes **EventHelper** and **EventThread**, which can be used in any situation where the client is going to act as a consumer of events in a pull model.

The **AccountEventHelper** is a subclass of the **EventHelper** class that customizes **EventHelper**. It overrides the **handleEvent** method in order to notify the GUI view that the AccountBalance has been modified.

The **EventHelper** is a new class whose role is to connect the client as a consumer for a given channel.

```
import org.omg.CosEventComm.PullSupplier;
import org.omg.CosEventChannelAdmin.EventChannel;
import org.omg.CosEventChannelAdmin.ProxyPullSupplier;
import com.ibm.IExtendedNaming.NamingContext;

import Proxy.CBCBase;

class EventHelper extends Object
{
  NamingContext rootNameService = null;
  private EventThread runner = null;
  private String channelName;
  PullSupplier pullSupplier = null;
  boolean shouldStop = false;

}
```

Figure 60. EventHelper class

The constructor requires two parameters:

- Name Service
- Channel Name

CBConnector automatically creates a single default event channel that is registered in the system name space in:

```
/.:/resources/event-channels/cell-default
```

We could have created our own event channel, but chose in this sample to use the default.

The application/applet, AccountPullApp, that we implemented to pull for events, creates an **AccountEventHelper** as part of its initialization; then it calls **run** to start pulling for events.

```
// initialize Event Helper
String eventChannel = "/.:/resources/event-channels/cell-default";
accountEventHelper = new AccountEventHelper(CBCBase.nameService,
eventChannel);
accountEventHelper.run( 100 );
CBCBase.traceIt("pulling for events....");
```

Figure 61. Creating an AccountEventHelper

The **run** method connects to the channel and registers the client as a **pull_consumer**. Then it creates an instance of **EventThread** that implements a thread and starts it. See Figure 62.

```
public void run( int interval ) {

  try{
   EventChannel channel;
   org.omg.CosEventChannelAdmin.ConsumerAdmin cadm;
   ProxyPullSupplier pps;
   org.omg.CORBA.Object tmp;
   shouldStop = false;

   tmp = rootNameService.resolve_with_string( channelName );
   channel = org.omg.CosEventChannelAdmin.EventChannelHelper.narrow( tmp );
   cadm = channel.for_consumers( );
   pullSupplier = cadm.obtain_pull_supplier( );
   pps = org.omg.CosEventChannelAdmin.ProxyPullSupplierHelper.narrow( pullSupplier );

   pps.connect_pull_consumer( null );
   runner = new EventThread( this,interval );
   runner.start( );
  }
  catch (org.omg.CosNaming.NamingContextPackage.NotFound e) {
    CBCBase.traceIt("Exception occurred in EventHelper: " + e);
  }
  catch (org.omg.CosNaming.NamingContextPackage.CannotProceed e) {
    CBCBase.traceIt("Exception occurred in EventHelper: " + e);
  }
  catch (org.omg.CosNaming.NamingContextPackage.InvalidName e) {
    CBCBase.traceIt("Exception occurred in EventHelper: " + e);
  }
  catch (org.omg.CosEventChannelAdmin.AlreadyConnected e) {
    CBCBase.traceIt("Exception occurred in EventHelper:" + e);
  }
  catch (Exception e) {
    CBCBase.traceIt("Exception occurred in EventHelper: " + e );
  }
}
```

Figure 62. EventHelper run method

The **EventThread.run** method (Figure 63), enters a loop where it checks for available events by calling **try_pull** on the helper object, which is an instance of **EventHelper** passed as a parameter during the creation of the **EventThread** instance, and then goes to sleep for an amount of time (100 ms).

```
public void run( ) {

  org.omg.CORBA.Any event;
  org.omg.CORBA.BooleanHolder hasEvent;

  hasEvent = new org.omg.CORBA.BooleanHolder( false );
  try      {
    while( helper.shouldStop == false ) {
    event = helper.pullSupplier.try_pull( hasEvent );
    if( hasEvent.value == true )
      helper.handleEvent( event );
      sleep( interval );
    }
  }
  catch( org.omg.CosEventComm.Disconnected e ) {  }
  catch( java.lang.InterruptedException e ) {  }
}
```

Figure 63. EventThread run method

If an event is available, it invokes the **handleEvent** method on the **EventHelper** instance and passes it the event.

In the **handleEvent** method (Figure 64), we need to extract from the event object the expected content. The format of the event is defined by the application. In our case, it consists of a single string that we parse to obtain the accountHolder, accountNumber and balance of the **Account** that has been changed. We then log the event in the trace window. This is done in the overridden **handleEvent** method **AccountEventHelper** class.

```
public void handleEvent( org.omg.CORBA.Any event ) {

  com.ibm.CORBA.iiop.AnyImpl e;
  e = ( com.ibm.CORBA.iiop.AnyImpl ) event;
  String eventString = e.extract_string();

  java.util.StringTokenizer tok = new java.util.StringTokenizer(eventString, ":");
  String accountNumber = tok.nextToken();
  String accountHolder = tok.nextToken();
  String newBalance = tok.nextToken();

  CBCBase.traceIt("A balance has been updated  ");
  CBCBase.traceIt("   Account        : " + accountNumber);
  CBCBase.traceIt("   Account Holder : " + accountHolder);
  CBCBase.traceIt("   new Balance    : " + newBalance);
}
```

Figure 64. HandleEvent method

16.4.2.3 Run the client

To see the "pull" client for this sample do anything, you need to run it at the same time you run another simple client which will allow you to make changes to **Account** balances. This simple client is called **EventService.AccountApp**.

You can use the transient or persistent sample clients to do the same thing, but make sure you check them against the AccountEVScope so that you can update balances to the right **Account** objects.

To run the client applications, execute the commands:

```
java AccountApp <hostname> <port> AccountEVServer-server-scope

java AccountPullApp <hostname> <port>
```

To run the clients as applets, embed the applets in HTML pages (*Account.html* and *EventPull.html*) with the same parameters as the applications, and open those pages in the applet viewer with the commands:

```
appletviewer Account.html

appletviewer EventPull.html
```

If you use the sample command (CMD) files provided on the CD-ROM, remember to change the `hostname` and `port` parameters (in *runApplet.cmd*, *Account.html*, and *EventPull.html*) to reflect your bootstrap server.

16.5 What did you learn?

In this chapter, you learned how to implement a server application that acts as an event push supplier, and how to implement clients to pull events from the event channel.

Chapter 17. Notification Service

This chapter takes the sample from the previous chapter and replaces the use of the Event Service with the Notification Service.

You can find the Object Builder model, code, and client applications for this sample on the CD-ROM for this book in the directory *\Samples\Notification-Service*.

17.1 What you will learn

In the previous chapter, you learned how to use the Event Service to send and receive events. In this chapter, you will learn how to use the Notification Service, a superset of the Event Service, to create and send structured "typed" events, and how to filter the events that clients receive.

17.2 CBConnector componentry

Figure 65 shows the main components used in our Notification Service Account scenario.

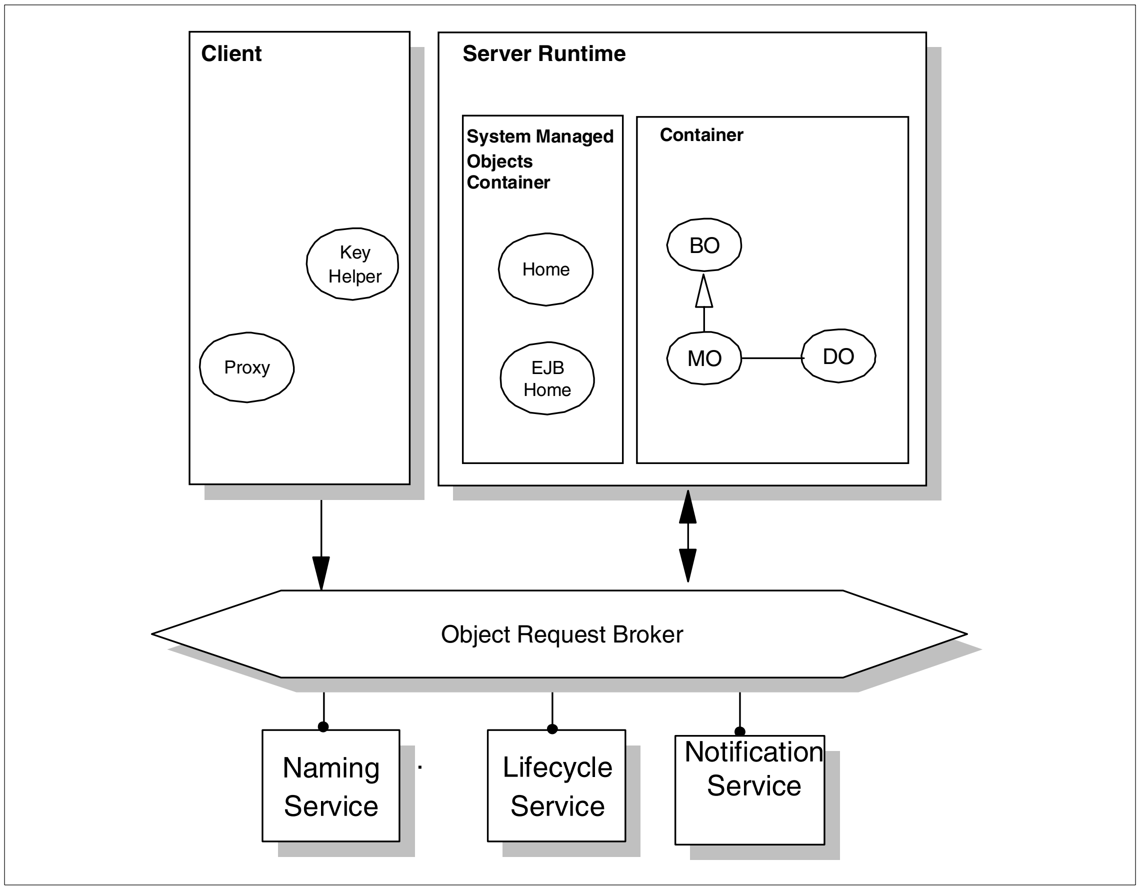


Figure 65. Notification Service sample components

17.2.1 Notification Service

The purpose of the Notification Service, like the Event Service, is to enable objects to freely register or unregister their interest in certain events. A Notification Service decouples communication between objects by defining two roles for objects: supplier objects and consumer objects. Suppliers produce events, while consumers process events.

The Notification Service provides additional functionality beyond simple queuing and retrieving of data which the Event Service provides. With the Notification Service, the messages you place on event channels have a defined structure that allows you to create message types, set qualities of service, and add "attributes" to your messages in the form of name/value properties. This service also provides the use of filters that can be applied when clients are pulling for events. Clients can register interest in events of a certain type and apply conditional logic to events through "filters", which will only hand back to clients those events in the queue that fulfill the clients' requirements.

For more information about the Notification Service, see Chapter 3, "Notification Service", in the *WebSphere Application Server Enterprise Edition Component Broker Advanced Programming Guide*, SC09-4443.

17.2.1.1 Structured events

The Notification Service defines an event message as a structured event. Each structured event consists of two main components: a header and a body. The header part mainly consists of information about the name of the event, the type of the event, and various (optional) Quality of Services (QoS) that are applicable to the event message. The body consists of the actual contents of the event instance upon which consumers are most likely to base filtering decisions.

Component Broker supports three QoS properties: Priority, StopTime, and TimeOut. *Priority* indicates the relative priority of the event compared to other events (1 - 10, with 10 being the greatest importance). *StopTime* is an absolute time (for example, 12/12/1999 at 23:59) when the channel should discard the event. *TimeOut* is a relative time (for example, 10 minutes from time received) when the channel should discard the event.

In this chapter, we use the cell default notification channel to push notifications to consumers. When the balance of an account is changed, the object (the supplier) pushes an event to the channel containing the number of the changed account, the name of the account holder, and the new balance.

In the sample, three types of events are generated: AccountEmpty, AccountCredited, AccountDebited. When the balance of an account reaches zero, an AccountEmpty event is thrown. AccountCredited and AccountDebited events are generated when amounts are credited or debited from the account balance.

17.2.1.2 Filters

Filters encapsulate constraints which will be used by a proxy object associated with a notification channel in order to make decisions about which events to forward, and which to discard. Each event must satisfy at least one of the constraints associated with one of the filters applied in order to be forwarded to the interested party.

Each constraint encapsulated by a filter object is a structure comprised of two main components. The first component is a sequence of data structures, each of which indicates an event type comprised of a domain and a type name. The second component is a boolean expression over the properties of an event, expressed in the constraint grammar (see "Appendix A. Default Filter Constraint Language" of the *WebSphere Application Server Enterprise Edition Component Broker Programming Reference*). For a given constraint, the sequence in the event type structures in the first component nominates a set of event types to which the constraint expression in the second component applies.

In the sample for this chapter, the client filters for events only of type AccountEmpty, AccountCredited, and AccountDebited. It also applies constraints around these events, to further discriminate which events it is interested in. For example, a bank might want to be notified when a customer's account balance goes above a certain amount, or falls below a minimum limit for an account. So for the constraints of our filter, we add the conditions *$balance > 1000* and *$balance < 1000*. When the client pulls for events, it will only receive events of these types, and only if the balance attribute of the event is within the specified range.

17.3 Server

There are very few differences between the model in this sample and the one you built previously in the Event Service chapter. Instead of the Account object inheriting CosEventComm::PushSupplier, it inherits from CosNotifyComm::StructuredPushSupplier. Other than that, the only difference lies in the implementation of pushing structured events onto a notification channel.

17.3.1 Build

In this section, we walk you through building an Account object which supplies events to a notification channel. You can implement the BO for this component, as with any component, using C++ or Java. In the following steps, we document the Java implementation. The steps are the same in any case, and we provide you with the source for both implementations.

The object model we provide for you on the CD in the directory *\Samples\Notification-Service\Server* contains BO implementations written in both C++ and Java. In this model, you will also see how to have two implementations of the same interface in the same model.

Note: There is a Notification sample that is shipped with Component Broker that is almost identical to the sample provided in this sample. You can find it in the Component Broker install directory on your machine in *\<CBroker install>\samples\Tutorial\Notification*. The sample in this chapter documents the steps for using the Notification Service in more detail, and provides Java implementation source to use as reference. The sample shipped with the product and described in the *Advanced Programming Guide* only has code samples written in C++.

We strongly recommend that you open help for each field of the SmartGuide pages in order to understand more of the dialectic behind the input fields.

17.3.1.1 Create the model

1. Create a new Object Builder model.
2. From **User-Defined Business Objects**, add a new file, *Account*.
3. For this file, add an interface, **Account**, with the following parameters:

Parameter	Value
Name	Account
Constructs: Exception	NotEnough
Interface Inheritance	IManagedClient IManagedClient::IManageable (default) CosNotifyComm CosNotifyComm::StructuredPushSupplier

Add attributes to the Account Interface as described in the following table:

Parameter	Type	Implementation
accountNumber	string<10>	Public
AccountHolder	string<30>	Public

Add methods, **changeBalance** and **getBalance** with the following definitions:

Method	Return Type	Parameter	Exception
changeBalance	void	anAmount double in	NotEnough

Method	Return Type	Parameter	Exception
getBalance	double		

4. For the Account Interface, add a Key using the accountNumber as the Primary Key.
5. Add a Copy Helper with all the attributes.
6. Add an implementation of the BO. Use the following parameters:

Parameter	Value
Pattern for handling state data	Caching
Implementation Language	Java
Attributes	balance (double, Private) pushConsumer (CosNotifyChannelAdmin::StructuredProxy PushConsumer, Private)
Methods to Override	StructuredPushSupplier:: disconnect_structured_push_supplier
State data	accountNumber accountHolder balance

Note

In the following steps, as an alternative to typing in the method implementations in the editor pane, you can link to an external file. To do this, open the properties of the method by right-clicking on the method, and select "properties". Then select "Use an external file," and browse to select the file which has the method implementation. The code for this sample is located on the CD-ROM for this book in the directory *\Samples\Notification-Service\Server\Source*. The files that contain the method implementations have the extension ".TDE". For Java implementations, we use the convention of appending "_java" to the file name, and "_cpp" for C++ implementations.

7. Implement the **changeBalance** method:

```
double newBalance = iBalance + anAmount;
if (newBalance < 0) throw new AccountPackage.NotEnough();
iBalance = newBalance;

if (iPushConsumer == null) return;

// create structured_event
org.omg.CosNotification.StructuredEvent structuredEvent = null;

try {
System.out.println("creating structured_event");

// fixed_header
String domain_name = "Samples";

String event_type = null;
if (iBalance == 0) event_type = "AccountEmpty";
else if (anAmount > 0) event_type = "AccountCredited";
else if (anAmount < 0) event_type = "AccountDebited";

String event_name = "Account:" + accountNumber();

org.omg.CosNotification.FixedEventHeader fixed_header =
        new   org.omg.CosNotification.FixedEventHeader(domain_name,   event_type,
event_name);

  // variable_header - set Qualities of Service
int numberOfQoS = 3;
org.omg.CosTrading.Property[]              variable_header              =              new
org.omg.CosTrading.Property[numberOfQoS];

for (int i = 0; i < numberOfQoS; i++) {
   variable_header[i] = new org.omg.CosTrading.Property();
   variable_header[i].value = com.ibm.CBCUtil.CBSeriesGlobal.orb().create_any();
}
short priority = (short) 6;
variable_header[0].name = "Priority";
variable_header[0].value.insert_short(priority);

int timeOut = (2*24*60*60*10000000); // 2 days in 100 nanoseconds
variable_header[1].name = "Timeout";
variable_header[1].value.insert_long( timeOut  );
int stopTime = 944326860;  // 1999 December 4, 12.01
variable_header[2].name = "StopTime";
variable_header[2].value.insert_long( stopTime );

// header
org.omg.CosNotification.EventHeader header =
   new org.omg.CosNotification.EventHeader(fixed_header, variable_header);

     // filterable_data
int numberOfAccountAttributes = 3;
org.omg.CosTrading.Property[]                filterable_data                =                new
org.omg.CosTrading.Property[numberOfAccountAttributes];

for (int i = 0; i < numberOfAccountAttributes; i++) {
   filterable_data[i] = new org.omg.CosTrading.Property();
   filterable_data[i].value = com.ibm.CBCUtil.CBSeriesGlobal.orb().create_any();
}

filterable_data[0].name= new String("accountNumber");
filterable_data[0].value.insert_string( accountNumber() );
filterable_data[1].name = new String("accountHolder");
filterable_data[1].value.insert_string( accountholder() );
filterable_data[2].name = new String("balance");
filterable_data[2].value.insert_double( getBalance() );

// remainder_of_body
org.omg.CORBA.Any                         remainder_of_body                         =
com.ibm.CBCUtil.CBSeriesGlobal.orb().create_any();
remainder_of_body.insert_string ("PUSH_DATA");

// structured_event
structuredEvent =
new             org.omg.CosNotification.StructuredEvent(header,             filterable_data,
remainder_of_body);

System.out.println("structured_event created");
}
catch(Throwable e) {
System.out.println("Exception creating structured_event");
return;
}
```

```
// push structured_event
try {
System.out.println(">>sending event");
iPushConsumer.push_structured_event(structuredEvent);
System.out.println(">>event sent");
}
catch(org.omg.CosEventComm.Disconnected e) {
System.out.println("Exception sending event: " + e);
}
catch(Throwable e) {
System.out.println("Exception sending event: " + e);
}
```

8. Implement the **getBalance** method:

```
return iBalance;
```

9. Implement the **disconnect_structured_push_supplier** method that you have overridden:

```
iPushConsumer.disconnect_structured_push_consumer();
```

10. Now you need to enhance the implementations of the framework methods **initializeState** and **uninitForDestruction**. In order to change these implementations, you must first open the properties of these methods under the **Framework Methods** folder, and select either **Use the implementation defined in the editor pane** or **Use an external file**.

11. Implement the framework method, **initializeState**:

```
System.out.println(">>Account::initializeState");

org.omg.CosNotifyChannelAdmin.EventChannel channel = null;

try {
System.out.println("Obtaining notification event channel");
String channelName = ".:/resources/notify-channels/cell-default"; // the default
Notification channel
org.omg.CORBA.Object obj =
com.ibm.CBCUtil.CBSeriesGlobal.nameService().resolve_with_string( channelName );
 channel = org.omg.CosNotifyChannelAdmin.EventChannelHelper.narrow(obj);
 System.out.println("event channel obtained");
}
catch (Throwable e) {
System.out.println("Unable to obtain Notification channel. Exception: " + e);
return;
}

try {
System.out.println("Obtaining StructuredProxyPushConsumer");
org.omg.CosNotifyChannelAdmin.SupplierAdmin sa =
channel.default_supplier_admin();
org.omg.CosNotifyChannelAdmin.ProxyConsumer pushConsumer =
sa.obtain_notification_push_consumer(org.omg.CosNotifyChannelAdmin.ClientType.S
TRUCTURED_EVENT, new org.omg.CORBA.IntHolder());
iPushConsumer =
org.omg.CosNotifyChannelAdmin.StructuredProxyPushConsumerHelper.narrow(push
Consumer);
System.out.println("StructuredProxyPushConsumer obtained");
}
catch (Throwable e) {
System.out.println("Unable to obtain StructuredProxyPushConsumer. Exception: " + e);
return;
}
```

```
try {
  // connect to the ProxyPushConsumer
  System.out.println("Connecting to notification channel");
  iPushConsumer.connect_structured_push_supplier(this);
  System.out.println("connected to channel");

  System.out.println(">>Account::initializeState successful");
}
catch(org.omg.CosEventChannelAdmin.AlreadyConnected e) {
  System.out.println("Account::initializeState: " + e); // benign
  return;
  }
  catch(Throwable e) {
  System.out.println("Account::initializeState: " + e);
  return;
  }
```

12. Implement the framework method, **uninitForDestruction**:

```
iPushConsumer.disconnect_structured_push_consumer();
```

13. For the AccountDO, add a Data Object implementation with the following parameters:

Parameter	**Value**
Environment	BOIM with any Key
Form of Persistent Behavior and Implementation	Transient

14. Add a Managed Object to the **AccountBO**. Accept all default settings.
15. Generate the code for all files from **Account** under **User-Defined Business Objects**.

17.3.1.2 Build the Server Application

In the **Build Configuration folder**:

1. Add a client DLL named AccountNSC with the following parameters:

Parameter	Value
Name	AccountNSC
Client Source Files	Account AccountKey AccountCopy
Libraries to Link With	none

2. Add a server DLL named AccountNSS with the following parameters:

Parameter	Value
Name	AccountEVS
Server Source Files	all
Libraries to Link With	AccountNSC

3. Generate the makefiles for "All Targets".
4. Build "All Targets".

17.3.2 Package

1. From the **Application Configuration** folder, add an application family named **AccountNSAppFam**.
2. For this application family, add a server application named AccountNSApp:
3. Add a Managed Object for the server application with the following parameters:

Parameter	Value
Managed Object	AccountMO AccountMO
Primary Key	AccountKey AccountKey
Copy Helper	AccountCopy AccountCopy
Data Object Implementation	AccountDOImpl AccountDOImpl
Container	CachedTransientObjects (or create a new one)

4. Generate the AccountNSAppFam.

17.3.3 Install

To install and configure the server application, follow these steps:

1. Install the server application on your server machine.
2. Configure an **AccountNSServer** in a new Management Zone, **AccountNS Management Zone**. Use a new configuration, **AccountNS Configuration**, for this serverActivate the configuration.
3. Add the following applications to the configuration, and configure them into the **AccountNSServer** server:
   AccountNSApp
   iDefaultCellNotifyChannel
   iNotificationService
4. Activate the configuration.

17.4 Client

These sections describe what is different in the client applications, as related to the general descriptions given in Chapter 10, “Sample client development approach” on page 117.

17.4.1 The C++ client

This is a simple C++ client that pulls from the event channel. To run this client, run the executable:
\Samples\Event-Service\Client\SimpleCpp\EventApp.exe.

You need to run the Java client at the same time so you can update balances of **Accounts** and see events being logged.

17.4.2 The Java client

All Java source and class files for this sample are located on the CD-ROM in the directory *\Samples\Notification-Service\Client\Java*. We have provided the VisualAge for Java repository file named "NotificationClient.dat" so that you can play with the code.

17.4.2.1 Notification Service Helper Classes

To use the Notification Service in this sample, we introduce three new classes (similar to the Event Service helper classes used in the previous chapter):

1. NotificationEventHelper
2. NotificationEventThread
3. AccountNotificationEventHelper

First, we have two generic classes, **NotificationEventHelper** and **NotificationEventThread**, which can be used in any situation where the client is going to act as a consumer of structured events in a pull model.

The **AccountNotificationEventHelper** is a subclass of the **NotificationEventHelper** class. It overrides the **handleEvent** method in order to notify the GUI view that the Account balance has been modified.

The **NotificationEventHelper** is a new class whose role is to connect the client as a consumer for a given notification channel and filter events.

```
import com.ibm.IManagedClient.*;
import com.ibm.IExtendedLifeCycle.*;

import com.ibm.IExtendedNaming.NamingContext;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyFilter.*;

public class NotificationEventHelper
{
   protected String channelName;
   protected Filter filter;
   private NotificationEventThread runner;
   StructuredProxyPullSupplier pullSupplier;
   boolean shouldStop = false;

public NotificationEventHelper(String channelName) { ... }
public NotificationEventHelper(String channelName, Filter filter) { ... }
public void handleEvent( org.omg.CosNotification.StructuredEvent event) { ... }
public void run( int interval ) { ... }
public void stop( ) { ... }
}
```

Component Broker automatically creates a single default notification channel that is registered in the system name space in:

```
.:/resources/notify-channels/cell-default
```

We could have created our own event channel, but chose in this sample to use the default.

The application, AccountNotificationPullApp, that we implemented to pull for events, creates an **AccountNotificationEventHelper** as part of its initialization; then it calls **run** to start pulling for events.

```
// initialize Event Helper
String eventChannel = ".:/resources/notify-channels/cell-default"; // the default
Notification channel
accountEventHelper = new AccountNotificationHelper( eventChannel );
accountEventHelper.run( 100 );

CBCBase.traceIt("pulling for structured events....");
```

The **AccountNotificationEventHelper** creates a filter appropriate for the types of events we want to receive:

```
org.omg.CosNotifyFilter.Filter filter = null;

try {
CBCBase.traceIt("Creating Filter");

// find FilterFactory
com.ibm.IExtendedLifeCycle.FactoryFinder factoryFinder =
CBCBase.resolveDefaultFactoryFinder();
com.ibm.IManagedClient.IHome home = CBCBase.resolveHome(factoryFinder,
"INotifyFilterManagedClient::Filter.object interface");
com.ibm.INotifyFilterManagedClient.FilterFactory filterFactory =
com.ibm.INotifyFilterManagedClient.FilterFactoryHelper.narrow(home);

// create Filter
filter = filterFactory.create_filter("IBM_NTF_CTG");

// create Constraints

org.omg.CosNotifyFilter.EventType[] eventTypes = new
org.omg.CosNotifyFilter.EventType[3];
String domain_name = "Samples";
String event_type0 = "AccountCredited";
String event_type1 = "AccountDebited";
String event_type2 = "AccountEmpty";
eventTypes[0] = new org.omg.CosNotifyFilter.EventType(domain_name, event_type0);
eventTypes[1] = new org.omg.CosNotifyFilter.EventType(domain_name, event_type1);
eventTypes[2] = new org.omg.CosNotifyFilter.EventType(domain_name, event_type2);
org.omg.CosNotifyFilter.ConstraintExp[]             constraints             =             new
org.omg.CosNotifyFilter.ConstraintExp[3];
String contraint0 = "$balance > 1000";
String contraint1 = "$balance < 500";
String contraint2 = "$balance == 0";
constraints[0] = new org.omg.CosNotifyFilter.ConstraintExp(eventTypes, contraint0);
constraints[1] = new org.omg.CosNotifyFilter.ConstraintExp(eventTypes, contraint1);
constraints[2] = new org.omg.CosNotifyFilter.ConstraintExp(eventTypes, contraint2);

// add Constraints to Filter
org.omg.CosNotifyFilter.ConstraintInfo[] constraintInfo = null;
constraintInfo = filter.add_constraints(constraints);
}
catch(Throwable e) {
  CBCBase.traceIt("Exception occurred in AccountEventHelper: " + e );
}
```

The **run** method connects to the channel and registers the client as a **pull_consumer** and attaches the filter to the **pullSupplier**. Then it creates an instance of **NotificationEventThread** that implements a thread and starts it.

```
 try{
if (runner == null) {

shouldStop = false;

CBCBase.traceIt("Obtaining notification event channel: " + this.channelName);
NamingContext rootNameService = CBCBase.nameService;
org.omg.CORBA.Object          obj          =          rootNameService.resolve_with_string(
this.channelName );
               org.omg.CosNotifyChannelAdmin.EventChannel               channel               =
org.omg.CosNotifyChannelAdmin.EventChannelHelper.narrow(obj);
 CBCBase.traceIt("event channel obtained");

CBCBase.traceIt("Obtaining StructuredProxyPullSupplier");
org.omg.CosNotifyChannelAdmin.ConsumerAdmin               consumerAdmin               =
channel.default_consumer_admin();
org.omg.CosNotifyChannelAdmin.ProxySupplier proxySupplier =

consumerAdmin.obtain_notification_pull_supplier(org.omg.CosNotifyChannelAdmin.Cl
ientType.STRUCTURED_EVENT, new org.omg.CORBA.IntHolder());
                                                            pullSupplier                              =
org.omg.CosNotifyChannelAdmin.StructuredProxyPullSupplierHelper.narrow(proxySu
pplier);
CBCBase.traceIt("StructuredProxyPullSupplier obtained");

// add Filter to StructuredProxyPullSupplier
if (filter != null) {
pullSupplier.add_filter(filter);
CBCBase.traceIt("Filter attached to StructuredProxyPullSupplier");
}

pullSupplier.connect_structured_pull_consumer(null);
runner = new NotificationEventThread( this, interval );
runner.start();
}

  }
  catch (Exception e) {
  CBCBase.traceIt("Exception occurred in NotificationEventHelper: " + e );
  }
```

The **NotificationEventThread.run** method enters a loop where it checks for available events by calling **try_pull** on the helper object, which is an instance of **NotificationEventHelper** passed as a parameter during the creation of the **NotificationEventThread** instance. It then goes to sleep for an amount of time (100 ms)..

```
while (true) {

try {
        org.omg.CORBA.BooleanHolder         hasEvent         =         new
org.omg.CORBA.BooleanHolder(false);
          org.omg.CosNotification.StructuredEvent            event            =
helper.pullSupplier.try_pull_structured_event(hasEvent);

 if ((hasEvent.value == true) && (event != null)) helper.handleEvent(event);
  }
  catch( org.omg.CosEventComm.Disconnected e ) {
  System.out.println("NotificationEventThread : " + e);
  }
  catch( Throwable e ) {
  System.out.println("NotificationEventThread : " + e);
  e.printStackTrace(System.out);
  }

  // sleep
  try { java.lang.Thread.sleep(interval);}
  catch( java.lang.InterruptedException e ) { }
}
```

If an event is available, it invokes the **handleEvent** method on the **NotificationEventHelper** instance and passes it the event.

In the **handleEvent** method, we can extract from the structured event object the content we are interested in. In our case, we pull out the accountNumber, accountholder, and balance of the **Account** that has been changed. We then log the event in the trace window. This is done in the overridden **handleEvent** method in the **AccountEventHelper** class:

```
System.out.println(">>Account notification event received");

org.omg.CosNotification.EventHeader header = event.header;

org.omg.CosNotification.FixedEventHeader fixed_header = header.fixed_header;
String domain_name = fixed_header.domain_type;
String event_name = fixed_header.event_name;
String event_type = fixed_header.event_type;

org.omg.CosTrading.Property[] variable_header = header.variable_header;
System.out.println("\t variable_header:");
String name = null;

// Priority
name = variable_header[0].name;
short value = variable_header[0].value.extract_short();
System.out.println("\t\t name: " + name + "  value: " + value);

org.omg.CosTrading.Property[] filterable_data = event.filterable_data;

String accountNumber = filterable_data[0].value.extract_string();
String accountHolder = filterable_data[1].value.extract_string();
double newBalance = filterable_data[2].value.extract_double();

CBCBase.traceIt(" Notification Event received:");
CBCBase.traceIt("   domain_name    : " + domain_name);
CBCBase.traceIt("   event_name     : " + event_name);
CBCBase.traceIt("   event_type     : " + event_type);
CBCBase.traceIt("   Account        : " + accountNumber);
CBCBase.traceIt("   Account Holder : " + accountHolder);
CBCBase.traceIt("   new Balance    : " + newBalance);
```

17.4.2.2 Run the client

All of the clients listed here have sample command (CMD) files provided on the CD which show how to execute them. If you use these, remember to change the `<host name> and <port number>` parameters to reflect your bootstrap server.

Pull Client

The "pull" client simply connects to the default notification channel and pulls for structured events. To run this client, execute the following command:

```
java AccountNotificationPullApp <host name> <port number>
```

Account client

To see the "pull" client for this sample do anything, you need to run it at the same time you run another simple client which will allow you to make changes to **Account** balances.

You can use the transient or persistent sample clients to do this, but make sure you specify the scope of your AccountNSServer, so that you will be able to update balances to the right **Account** objects, in case you have other **Account** objects installed from other samples. The transient sample client is included in the *\Client* directory for this chapter for your convenience. To run this client, execute the command:

```
java AccountApp -classpath ".;jcbAccountEVC.jar;%CLASSPATH%"
<host name> <port> AccountNSServer-server-scope
```

Another client you can use is the generated QuickTest client for this sample. You should be able to run this client from your project directory by issuing the following command::

```
\Working\NT\PRODUCTION\qt.bat
```

If you'd like, you can run the QuickTest client from the CD, by executing the following command:

```
\Samples\Notification-Service\Server\Working\NT\PRODUCTION\qt.bat
```

Note: If you install the server application for this sample from the object model that is provided on the CD, you will be installing two managed objects: one that is implemented in C++ (named AccountMO), and one implemented in Java (named AccountJMO). What this means is, in order to find or create Accounts with a specific implementation, you need to use the specific home for that implementation. You do this by finding the home for the Account interface and specifying the home's interface as it is registered in the name space.

So, for example, to find the Account which is implemented in C++,
you would look for an Account home with the string
"Account.object interface/AccountMOFactory.object home".

To find the home of the Account implemented in Java, use:
"Account.object interface/AccountJMOFactory.object home".
QuickTest has a menu pull-down option that allows you to set this.

PushClient
In the sample, you created a server object which acts as an event supplier. You can also have clients be suppliers of events, as well. We provide a client that has identical code as the **Account** BO that pushes Account events onto the default notification channel. This client is called **AccountNotificationPushClient**, and to run it, you would execute the following command::

```
java AccountNotificationPushClient <host name> <port>
```

17.5 What did you learn?

In this chapter, you learned how to implement a server application that acts as an structured event push supplier (through the use of the Notification Service), and how to implement clients to pull and filter structured events from the notification event channel.

Chapter 18. LifeCycle Service

This chapter describes and implements Location Objects and scopes. We have written a short sample application deploying the scope definitions to DCE Common Directory Services.

18.1 What you will learn

You will learn how to create your own Factory Finders and Location Objects.

18.2 CBConnector componentry

In the following sections, we explain the relationship between Location Objects and Factory Finders.

18.2.1 Location Objects

Location Objects can be used to resolve the ambiguity of different interface implementations and locations where those objects are created (proximity). The location scope is defined in a Location Object registered with the Factory Finder when it was created. The Location Object is like a set of criterions bound into the Factory Finders, which are then evaluated at runtime to select only factories that satisfy those criterions.

Since we have many implementations of **Account**, all client programs in this book use Location Objects to resolve the ambiguity between specific locations and object implementations. You ask the Name Service for a specific Factory Finder which interprets predefined criterions at runtime, thus returning the correct factory.

18.2.2 Factory Finders

You can find a factory that supports creating a particular kind of object with a Factory Finder. Factory Finders are intended to find factories. They do so based on two criteria: the kind of object for which a factory is needed, and a location scope in which to look. The kind of object is specified in the **factory_key** supplied to the Factory Finder on a **find_factory** or **find_factories request**.

CBConnector automatically creates a number of Factory Finders and binds them in the system name space for immediate use. These are referred to as the default Factory Finders. One default Factory Finder is produced and assigned a corresponding scope for each of the most commonly used scopes. This includes host-scope, workgroup-scope and cell-scope.

18.3 Do it

In the following sections, we explain the implementation of the LifeCycle Service.

Note

All code snippets have enumerated lines for easier orientation in the code.

18.3.1 Obtain a Factory Finder

Typically, once a Factory Finder (specifically, a managed Factory Finder) has been created, it will be bound in the system name space for further use by other users or applications. Thus, obtaining a Factory Finder is a matter of looking it up by its name in the system name space. To find a Factory Finder, you normally execute code like this:

```
01 factoryFinder = rootNC->resolve_with_string(
"/host/resources/factory-finders/host-scope" );
```

18.3.2 Find a factory of Factory Finders

To create a Factory Finder, you need a factory (such as the home). You can find the home of Factory Finders by using the default Factory Finder:

```
01 home = ffDef->find_factory_from_string(
        ILifeCycleManagedClient::factoryfinder.object interface" );
```

Here, `factory` is actually the home returned by the Factory Finder. This can be narrowed to **ILifeCycleManagedClient::factoryfinderhome**.

18.3.3 Find a factory of Location Objects

Typically, once a location scope (specifically, a Managed **SingleLocation** Object) has been created, it will be bound in the system name space for further use by other users or applications. Thus, obtaining a location scope is a matter of looking it up by its name in the system name space. To find the factory of Location Objects:

```
locHome = ffDef->find_factory_from_string(
      "ILifeCycleManagedClient::singlelocation.object interface" );
```

Here, `factory` is actually the home returned by the Factory Finder. This can be narrowed to **ILifeCycleManagedClient::singlelocationhome**.

18.3.4 Create a Location Object

A managed Factory Finder is one that is a Managed Object, and therefore one that can be retained persistently. To associate a Location Object with a Factory Finder, you have to create a new Factory Finder and attach the scope properties to it by creating a Location Object. To create the Location Object with the scope structure:

```
IExtendedLifeCycle::Scope scope;

scope.cell = CORBA::string_dup( "*LOCAL" );
scope.workgroup = CORBA::string_dup( "*LOCAL" );
scope.host = CORBA::string_dup( "*LOCAL" );
scope.server = CORBA::string_dup( "AccountServer" );
scope.container = CORBA::string_dup( "*ANY" );
scope.home = CORBA::string_dup( "*ANY" );

location = locHome->createWithScope( scope,"Account",1,1,0 );
```

The second parameter of **createWithScope**, the relative name, is the used to bind the Location Object to the name tree(s).

The Location Object can also be created using XFN (X/Open Federated Naming) standard syntax model string. The syntax follows the pattern <id>.<kind>. In each name-component, the <kind> field is used to identify the boundary-element, and the <id> field is used to specify the value to be used for that boundary element.

18.3.5 Create a Factory Finder

To associate a Location Object with a Factory Finder, you have to create a new Factory Finder and attach the scope properties to it by creating a Location Object.

```
01 ffHome->createWithLocation( location,"AccountScope",1,1,0 );
```

The second parameter of **createWithScope**, the relative name, is the used to bind the Factory Finder object to the name tree(s).

18.4 Run it

Under the *CBConnector\12-CBC-Services-Lifecycle-Naming\Client\* subdirectory on the CD-ROM, you can find the source code and executable for this sample. In the *Utility* subdirectory, we placed the *.CMD* files. The generic syntax to run this sample to create or remove scopes is as follows:

```
loccp /CREATE or (/REMOVE) scope_name  server_name
```

In order to make a correct update of DCE's CDS, you have to be logged in to DCE with your administrator ID, and the CBConnector Name Server has to be running.

We describe this procedure in 4.1, “Get ready” on page 30.

18.5 What did you learn?

In this chapter, you learned how to create your own Factory Finders and Location Objects.

Chapter 19. Query Service — reuse of an existing table

In the previous chapters, we were dealing with a top-down development paradigm, defining and implementing your object model from scratch. In this sample, you will be developing a server application using a meet-in-the-middle approach, where you will re-use existing data stored in a database.

You can find the Object Builder model, code, and client applications for this sample on the CD for this book in the directory *\CBConnector\18-Reuse-Table*.

19.1 What you will learn

You will learn how to re-use an existing DB2 table, define a Business Object as queryable, and use the Query Service by performing queries for Managed Objects from client applications.

In the previous chapter, you defined the database table you needed to create, based on the attributes of the **Account** Business Object. In this sample, you will import the definition of an existing DB2 table (CUST) into Object Builder, then map attributes of a **Customer** Business Object to the appropriate columns in the table using the schema mapper.

The **Customer** object will be queryable, which means that it will reside in a queryable home. From client applications, we show you how to query the home for Customers using four different methods. You will see how to iterate over a home and how to perform "push-down" queries, which execute SQL queries directly in the database, rather in the CBConnector middle tier.

19.2 CBConnector componentry

Figure 66 shows the CBConnector componentry involved in this sample. The components in this sample are exactly the same as the sample in Chapter 14, “Transactional Object sample” on page 217. The only real difference is the way you go about creating the two samples.

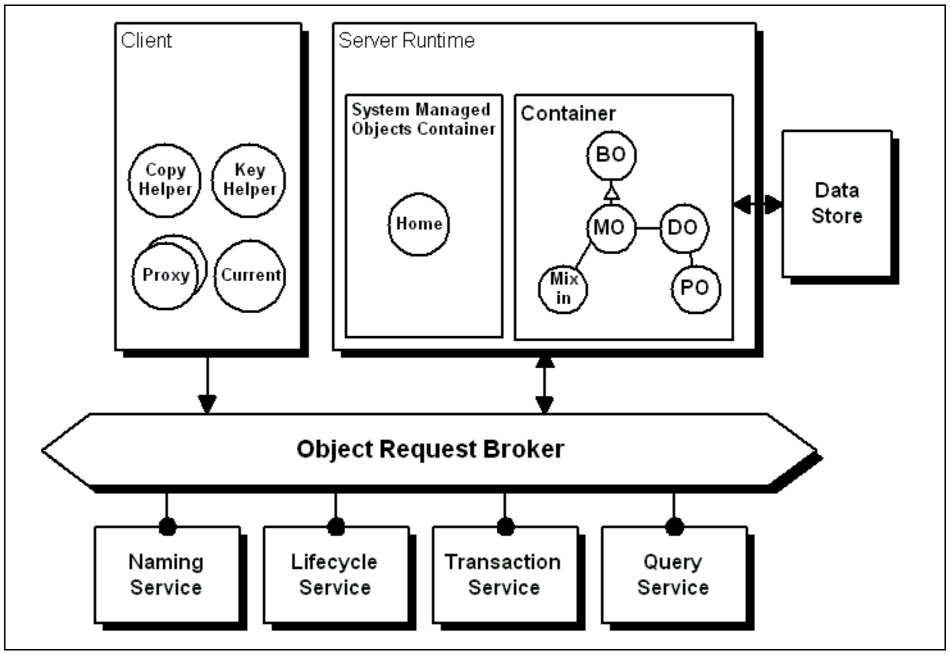


Figure 66. Persistent, transactional Business Object with Query Service

19.2.1 Delegating access pattern

In this sample, you will use a *delegating* pattern for handling how the state attributes of a Business Object are populated and made persistent. When a delegating pattern is used, the following things happen when you find an object, a **Customer**, for example. The duty of populating the `firstName` and `lastName` attributes is "delegated" to the Data Object. The Persistent Object (PO) retrieves the values stored in a database and places them in the PO. The **get/set** method retrieves and updates the copy of the data in the PO. At commit time, if the PO was updated, it is written back to the database.

19.2.2 Query Service

The Query Service gives you the ability to access collections of objects using OO-SQL. An OO-SQL query `SELECT` statement operates on one or more *collections* of objects. The `WHERE` clause of the statement is similar to relational SQL, except that the conditions go against object attributes, not table columns.

The Query Service takes the OO-SQL query statement and information from the home (called metadata) that describes how the objects' attributes are stored in database tables and columns, and it translates the OO-SQL query into an SQL query that runs against the database system (in this case, DB2). The rows returned from the SQL query are then used to construct objects.

In CBConnector, a queryable collection is an aggregation of objects that can be one of the following:

- **Home** — the birthplace and logical owner of objects of a given type
- **View** — a subset of some other collection, based on a predicate
- **Reference Collection** — a collection that holds references to potentially heterogeneous objects

In this sample, we show you how to query collections using the following methods:

- Iterate over a home
- Iterate over a view of a home
- Query over a home
- Query over a collection of data arrays

We don't cover *Reference Collections* in this sample.

19.2.2.1 Iterate over a home

By specifying an interface to be queryable, the Managed Object created for that interface will be placed in a *Queryable Iterable Home*. You can find all the objects in a home by creating an iterator over that home collection, then iterating over the home for each object. You create an iterator by calling the **createIterator** method of the home.

19.2.2.2 Iterate over a subset collection of a home

You can also create an iterator over a specific *subset collection* of a home, by calling the method **evaluate** of the home. With **evaluate**, you specify the predicate of an OO-SQL query statement (the WHERE clause), which creates a subset collection of a home. You are returned an iterator over the Managed Objects in this collection. For example, if you would like to find all Customers that have a first name of 'Jack', you would make the following method call on the Customer home:

evaluate("firstName = 'Jack';");

19.2.2.3 Query over a home

The query evaluator is a System Object on the application server that is a direct interface to the Query Service. It is registered in the name server under the name "host/resources/servers/<SEVER NAME>/query-evaluators/default." So, to get the query evaluator for the application server `MyServer`, you would call on the Name Service:

```
nameService->resolve_with_string ("/host/resources/servers/MyServer/
                                   query-evaluators/default")
```

The query evaluator provides the method **evaluate_to_iterator** that is similar to **evaluate** on the home, except here you provide a full OO-SQL query statement. What is returned to you is an array of references to all the objects that match that query and/or an iterator over this collection. For example, if you want to find all Customers that have a first name of 'Jack', the query statement would look like this:

```
select e from CustomerMOHome e where e.firstname=' Jack' ;
```

19.2.2.4 Query over a collection of data arrays

The query evaluator provides another method **evaluate_to_data_array** that returns a collection of data arrays and/or an iterator over a collection of data arrays. A data array is a sequence (array) of **CORBA::Any**. An **Any** can contain, as its name implies, "any" type of object. In this case, each **Any** will contain an attribute value.

Using the other query methods we have described above, you are returned only references to Managed Objects, not attributes. If you want the attributes for those objects, you must make a call across the ORB to retrieve each attribute. **evaluate_to_data_array** allows you to query for attribute values of a collection of objects with one trip across the ORB. For example, to query for the first and last names of all Customers, your query statement would look like this:

```
select e.firstName, e.lastName from CustomerMOHome e ;
```

19.3 Server

This section takes you through all the steps you need to perform to build, package, and install a server application that uses an existing DB2 table for persistence.

19.3.1 Build

Implementing this server application consists of three major steps:

1. Import the DB2 table definition and create the Persistent Object.
2. Specify the interface and implementation of the Business Object, Data Object, and the Key and Copy Helpers.
3. Generate and build the application.

Through the Object Builder walkthrough, we provide you with the correct parameter settings for the SmartGuides. If no suggestions are made by us for a SmartGuide page, use the default settings.

We strongly recommend that you open help for each field of the SmartGuide pages in order to understand more of the dialectic behind the input fields.

19.3.1.1 Create the database

This sample uses an existing DB2 table named CUST in a database called CustDB, which looks like this:

Column name	***Type***	***Length***	***Description***
CUSTNO	CHARACTER	10	Customer number
FNAM	CHARACTER	30	First name
LNAM	CHARACTER	30	Last name

Actually, since this is a sample, you will have to create the table. We provide an SQL script on the CD with this book that will create a CustDB database, create a CUST table, and load the table with data. The script is located on the CD in *\CBConnector\18-Reuse-Table\Server\DBscript\creCust.db2*. If you already have a database called CustDB that you do not want to destroy, use another database name in the steps described below. In that case, you must also remember to change the script we provided, before you run it.

To install the CustDB database and the CUST table, do the following:

1. Open a DB2 Command Window and change directory to the *\CBConnector\18-Reuse-Table\Server\DBscript\* directory on the CD.
2. Issue the command:

```
db2 -f creCust.db2
```

19.3.1.2 Create an SQL script

The Object Builder needs a definition of the table to re-use. This definition needs to be in an SQL script file containing a `CREATE TABLE` statement. In our case, the statement looks like this:

```
CREATE TABLE CUST
(
 CUSTNO CHARACTER(10) NOT NULL,
 FNAM CHARACTER(30) NOT NULL
 LNAM CHARACTER(30) NOT NULL
 PRIMARY KEY ( CUSTNO )
 );
```

For this sample, we have provided you with this table definition in a file on the CD named *\CBConnector\18-Reuse-Table\Server\DBscript\Cust.sql*.

It is possible to have more than one `CREATE TABLE` statement in a file, as long as each statement is ended with a semicolon (;). When you create the model, you will "import" this file into Object Builder, where you select which statements to actually import.

19.3.1.3 Create the model

1. Create a new Object Builder model in a separate directory.
2. On the **DBA-Defined Schemas** folder, use the **Import SQL...** pop-up menu option to import the CUST table definition. Define the schema group with the parameters as described in the following table:

Parameter	Value
File Name	CBConnector18-Reuse-Table/Server/ DBScript/Cust.sql
Group Name	CustomerSchemaGroup
Database Name	CustDB (change if necessary)
Statements to Import	CREATE TABLE CUST

When finished, there will be a schema group called CustomerSchemaGroup and a schema called `CUSTDB.CUST` under it.

3. Open the properties of this new schema, and set the `User ID` to the user ID with which the CustDB database was created. When you have done this, the schema name will be changed to `CUSTDB.USERID.CUST`, where `USERID` is your specific user ID.

4. For the CUSTDB.USERID.CUST schema, add a Persistent Object. Use the following parameters:

Parameter	Value
Name	CustomerPO
Package File	CustRT (RT=Reuse Table)
Type of Persistence	Embedded SQL
Mapping of Column to PQ Attribute	Change **CUSTNO** to **customerNumber**
Mapping of Column to PQ Attribute	Change **FNAM** to **firstName**
Mapping of Column to PQ Attribute	Change **LNAM** to **lastName**
PO Key	customerNumber

The Package File is the name that holds the necessary statements together in the DB2 database.

When finished, there will be a **CustomerPO** Persistent Object under the `CustDB.USERID.CUST` schema.

5. From **User-Defined Business Objects**, add a new file named *Customer*.
6. For this file, add an interface, **Customer**, with the following parameters:

Parameter	Value
Name	Customer
The interface is queryable	Yes
Constructs	none

Add attributes to the Customer Interface as described in the following table:

Attribute	Type	Implementation
customerNumber	string <1 0>	Public, Read-only
firstName	string <3 0>	Public
lastName	string <3 0>	Public

7. For the newly created Customer Interface, add a Key Helper using the customerNumber as the Primary Key.
8. Add a Copy Helper with the all the attributes.
9. Now add an implementation of the BO. Use the following parameters:

Parameter	Value
File Name	CustomerBO
Name	CustomerBO
Pattern for handling state data	Delegating
Data Object Interface	Create a new one now
Implementation Inheritance	IManagedClient IManagedClient::IManageable
Implementation Language	C + +
Key Selection	CustomerKey
Handle	none
Attributes to Override	none
Methods to Override	none
Data Object File Name	CustomerDO
Data Object Interface Name	CustomerDO
State data	all

10. For the CustomerDO, add a Data Object implementation. This is the "meet-in-the-middle" step, where you connect the DO with the PO. Use the following parameters:

Parameter	Value
File Name	CustomerDOImpl
Name	CustomerDOImpl
Environment	BOIM with any Key
Form of Persistent Behavior and Implementation	Embedded SQL
Data Access Pattern	Delegating
Handle for Storing Pointers	Home name and Key
Key helper	CustomerKey
Copy helper	CustomerCopy
Persistent Object Instance Name	iCustomerPO
Persistent Object Type	CustomerPO

Add attribute and method mappings between the DO and the P0 using the following table:

DO Attribute/Method	PO Attribute/Method
customerNumber	iCustomerPO.customerNumber
firstName	iCustomerPO.firstName
lastName	iCustomerPO.lastName
insert	iCustomerPO.insert( )
update	iCustomerPo.update( )
retrieve	iCustomerPO.retrieve( )
del	iCustomerPO.del( )
setConnection	iCustomerPO.setConnection(dataBaseName)

11. Add a Managed Object to the **CustomerBO**. Accept all default settings.

12. Save your model and generate the code for all files from **User-Defined Business Objects**.

19.3.1.4 Build the server application

In the **Build Configuration** folder, execute the following steps:

1. Add a client DLL named CustomerRTC with the following parameters:

Parameter	Value
Name	CustomerRTC
Client Source Files	all
Libraries to Link With	none

2. Add a server DLL named CustomerRTS with the following parameters:

Parameter	Value
Name	CustomerRTS
Server Source Files	all
Libraries to Link With	none

3. Generate the makefiles for all targets.

   Before you build the server application, make sure you have created the CustDB database and CUST table needed for this sample. If you have not done so, please follow the steps in 19.3.1.1, “Create the database” on page 295.

4. Build all out-of-data C++ targets.

19.3.2 Package

In this section, we describe how to package the Customer application. This includes the following steps:

- Definition of the application using the Object Builder.
- Creation of an installation image using InstallShield.

19.3.2.1 Define the application

1. Under the **Container Definition** folder, add a container instance with the following parameters:

Parameter	Value
Name	ContainerOfCustomerRT
Use Transaction Services	yes
Behavior for Methods Called Outside a Transaction	Throw an exception and abandon the call
Passivate a Component at End of Transaction	yes
Data Access Pattern Business Object	Delegating
Data Access Pattern Data Object	Delegating

2. From the **Application Configuration** folder, add an application family named **CustomerRTAppFam**.
3. For this application family, add a server application named CustomerRTAppS with parameters as follows:

Parameter	Value
Application Name	CustomerRTAppS
Initial state of application	stopped
Additional Executables	none

4. Add a Managed Object for the server application with the following parameters:

Parameter	Value
Managed Object	CustomerMO CustomerMO
Primary Key	CustomerKey CustomerKey
Copy Helper	CustomerCopy CustomerCopy
Data Object Implementation	CustomerDOImpl CustomerDOImpl

Parameter	Value
Container	ContainerOfCustomerRT
Default Home	Default Home
Home Name	BOIMHomeOfQIHomes
Name in Factory Finding Service Registry	CustomerMOFactory
Name in Naming Service Registry	CustomerMOHome

19.3.2.2 Create an installation image

From the CustomerRTAppFam, generate the installation scripts. This step also creates the System Management data in form of a DDL file.

19.3.3 Install

To install and configure the server application, follow these steps:

1. Bind the Customer server application to the CustDB database. The bind file created in your *Working* directory is named *CustomerPO.bnd*. Issue the following set of commands from your *Working* directory within a DB2 command window:

```
DB2 connect to CustDB
DB2 bind CustomerPO.bnd
DB2 connect reset
```

2. Bind the two CBConnector bind files needed for the use of the Query Service. These files are named *db2cntcs.bnd* and *db2cntrr.bnd* and are located in the directory *C:\CBroker\etc* (where *C:\CBroker* is the directory where you installed CBConnector). **This is a very important step. You must do this to perform queries.**

   Issue the following set of commands from this directory within a DB2 Command Window:

```
DB2 connect to CustDB
DB2 bind db2cntcs.bnd
DB2 bind db2cntrr.bnd
DB2 connect reset
```

3. Install the server application on your CBConnector server machine.

4. Configure a **CustomerRTServer** in a new Management Zone, **CustomerRT Management Zone**. Use a new configuration, **CustomerRT Configuration**, for this server. In this server, configure the applications **CustomerRTAppS**, **Specific CustomerRTAppS**, and the default DB2 application named **iDB2IMServices**.

   Note: For this sample, you must use exactly this server name for the application server to enable the execution of the client programs because the single Location Object you installed refers to it.

5. Activate the configuration.
6. Stop the **CustomerRTServer**, and set the **open string** with the database administrator user ID and password for the databases in the **XA Resource Manager images** of this server.
7. Start the server using `Run Immediate`.

19.4 Client

These sections describe what is different in the client applications, as related to the general descriptions given in Chapter 10, “Sample client development approach” on page 117.

19.4.1 The C++ client

The base class **cbcproxy** is enhanced with some helper methods to support OO-SQL query functions used in this sample. These helper methods are described in the following section.

19.4.1.1 The cbcproxy

The Query Services provided by the **cbcproxy** class are the following:

- createIterator
- homeEvaluator
- getEvaluator
- iterate

The **createIterator** method allows us to create an iterator over all the objects in a queryable home, while the **homeEvaluator** method allows us to narrow the number of objects in the iterator returned according to a selection criterion given as parameters to the **homeEvaluator** method. Both methods need a pointer to the queryable home as a parameter.

A helper method, **getEvaluator**, gives us a query evaluator based on the name of our CBConnector server. The **iterate** method enables us to do one iteration through the iterator acquired by the **createIterator** method or through the **homeEvaluator** method. Code snippets showing the implementation of our Query Services are shown in Figure 67:

```
/*
;--------------------------------------------------------------------;
;       create iterator                                              ;
;--------------------------------------------------------------------;
*/
ICollectionsBase::IIterator_ptr _Export cbcproxy::createIterator(
          IManagedAdvancedClient::IQueryableIterableHome_ptr home)
 {
ICollectionsBase::IIterator_ptr iterator;

  if (CORBA::is_nil(home))
    {
     cout << "Invalid home" << endl;
     return NULL;
    }
  try
    {
     cout << "About to create iterator" << endl;
     iterator = home->createIterator();
    }
  catch(CORBA::UserException &e)
    {
     cout << "Couldn't create iterator - exception: "
         << e.id() << endl;
     return NULL;
    }
  catch(CORBA::SystemException &e)
    {
     cout << "Couldn't create iterator - exception: "
         << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
     return NULL;
    }
  catch( ... )
    {
     cout << "Couldn't create iterator"  << endl;
     return NULL;
    }
  cout << "Iterator OK" << endl;
  return iterator;
 }
```

Figure 67. Create iterator

The pointer to the queryable home, passed as a parameter, is typically created in the application layer by using the overloaded **init** method of the **cbcproxy** class to get the factory. Then the factory is narrowed to the queryable home by using the **_narrow** method of the **IManagedAdvancedClient::IQueryableIterableHome** class.

The method shown above essentially executes only the statement:

```
iterator = home->createIterator();
```

while the rest of the code handles errors. See Figure 68.

```
/*
;--------------------------------------------------------------------;
;       create home evaluator                                        ;
;--------------------------------------------------------------------;
*/
ICollectionsBase::IIterator_ptr _Export cbcproxy::homeEvaluator(
   IManagedAdvancedClient::IQueryableIterableHome_ptr home, char *qStr)
  {
ICollectionsBase::IIterator_ptr iterator;

   if (CORBA::is_nil(home))
      {
       cout << "Invalid home" << endl;
       return NULL;
      }
   try
      {
       cout << "About to create home evaluatator by: " << qStr << endl;
       iterator = home->evaluate(qStr);
      }
   catch (ICollectionsBase::IQueryInvalid)
      {
       cout << "Couldn't create home evaluator - Invalid query" << endl;
       return NULL;
      }
   catch (ICollectionsBase::IQueryTypeInvalid)
      {
       cout << "Couldn't create home evaluator - Invalid query type" << endl;
       return NULL;
      }
   catch(CORBA::UserException &e)
      {
       cout << "Couldn't create home evaluator - exception: "
            << e.id() << endl;
       return NULL;
      }
   catch(CORBA::SystemException &e)
      {
       cout << "Couldn't create home evaluator - exception: "
            << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
       return NULL;
      }
   catch(...)
      {
       cout << "Couldn't create home evaluator" << endl;
       return NULL;
      }
   return iterator;
  }
```

Figure 68. Create home evaluator

The pointer to the queryable home passed as a parameter, is typically created in the application layer, using the overloaded **init** method of the **cbcproxy** class to get the factory. Then the factory is narrowed to the queryable home by using the **_narrow** method of the **IManagedAdvancedClient::IQueryableIterableHome** class.

The method shown above, essentially executes only the statement:

```
iterator = home->evaluate(qStr);
```

while the rest of the code handles errors. See Figure 69.

```
/*
;-------------------------------------------------------------------;
;       get evaluator from servername                               ;
;-------------------------------------------------------------------;
*/
IExtendedQuery::QueryEvaluator_ptr _Export cbcproxy::getEvaluator(char *server)
  {
IString qEvalStr("/host/resources/servers/");
CORBA::Object_var  qe;
IExtendedQuery::QueryEvaluator_ptr eval;

   qEvalStr += server;
   qEvalStr += "/query-evaluators/default";

   try
     {
      qe = CBSeriesGlobal::nameService()->resolve_with_string(qEvalStr);
      eval = IExtendedQuery::QueryEvaluator::_narrow(qe);
     }
   catch(CORBA::UserException &e)
     {
      cout << "Couldn't get query evaluator - exception: "
           << e.id() << endl;
      return NULL;
     }
   catch(CORBA::SystemException &e)
     {
      cout << "Couldn't get query evaluator - exception: "
           << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
      return NULL;
     }
   catch(...)
     {
      cout << "Couldn't get query evaluator" << endl;
      return NULL;
     }
   cout << "Found the query evaluator" << endl;
   return eval;
  }
```

Figure 69. Get evaluator from server

We see that our server name is placed in the appropriate location of the naming tree, and the whole tree structure is passed to the Name Service, which returns an object to be narrowed by the **_narrow** method of the **IExtendedQuery::QueryEvaluator** class. The returned query evaluator pointer would then be used in the application layer to evaluate to an iterator. See Figure 70.

```
/*
;--------------------------------------------------------------------;
;       iterate                                                      ;
;--------------------------------------------------------------------;
*/
CORBA::Object_ptr _Export cbcproxy::iterate(
                    ICollectionsBase::IIterator_ptr iterator)
 {
CORBA::Object_ptr aBO;

  try
    {
     aBO = iterator->next();
    }
  catch (ICollectionsBase::IInvalidIterator &ii)
    {
     cout << "Couldn't  iterate - exception: "
          << ii.id() << endl;
     return NULL;
    }
  catch(CORBA::UserException &e)
    {
     cout << "Couldn't  iterate - exception: "
          << e.id() << endl;
     return NULL;
    }
  catch(CORBA::SystemException &e)
    {
     cout << "Couldn't  iterate - exception: "
          << e.id() << " minor code 0x" << hex << e.minor( ) << endl;
     return NULL;
    }
  catch( ... )
    {
     cout << "Couldn't iterate"  << endl;
     return NULL;
    }
  return aBO;
 }
```

Figure 70. Iterate

This utility method takes your iterator, as obtained by **createIterator** or **makeEvaluator**, and returns the next object from it, or NULL if there are no more objects or if an error is encountered.

We remember that the **custproxy** class is the application-dependent class, a subclass of the **cbcproxy** class, which hides the CBConnector specifics from the client application.

19.4.1.2 The custproxy

To support the needs of the visual layer, the following methods are implemented in the **custproxy** class:

- makeIterator
- makeEvaluator
- setObjIterator
- getCustomer
- getDataIterator

The sample shows four different ways of obtaining customer collections: one by iteration of all customer objects in the home, one by making an evaluator on the home that meets specific criteria, one that takes a complete OO-SQL statement as a parameter, and creates an iterator based on that, and finally one that also takes a complete OO-SQL statement as parameter, but returns a pointer to an array of arrays, of which the client has to know the layout. All the generated iterators are executed through the **getCustomer** method. Before we can do our queries, we need to initialize CBConnector as shown in Figure 71:

```
/*
;-------------------------------------------------------------------;
;        initialize Component Broker                                ;
;-------------------------------------------------------------------;
*/
int  _Export custproxy::initCBC()
  {
CORBA::Object_ptr factory;
/*
;------- calling init with dummy argument - to call overload method -;
*/
   factory = init("CustomerRTScope","Customer",0);
   if (factory == NULL)
      return FALSE;
/*
;------- narrow to queryable home -----------------------------------;
*/
   try
      {
       home = IManagedAdvancedClient::IQueryableIterableHome::_narrow(factory);
       CORBA::release(factory);
      }
   catch(CORBA::UserException &e)
      {
       cout << "narrow home failed - exception: "
            << e.id() << endl;
       return FALSE;
      }
   catch(CORBA::SystemException &e)
      {
       cout << "narrow home failed - exception: "
            << e.id() << endl;
       return FALSE;
      }
   catch( ... )
      {
       cout << "narrow home failed - exception" << endl;
       return FALSE;
      }
   iterator = NULL;

   cout << "home OK" << endl;
   return TRUE;
  }
```

Figure 71. Initialize component broker

We get the factory from the **cbcproxy** class and narrow it to a queryable home. The following code snippets in Figure 72 show the implementation of the methods:

```
/*
;---------------------------------------------------------------------;
;       make iterator                                                ;
;---------------------------------------------------------------------;
*/
int  _Export custproxy::makeIterator()
  {
   iterator = createIterator(home);
   if (iterator == NULL)
      return FALSE;

   cout << "iterator is OK" << endl;
   return TRUE;
  }
```

Figure 72. makeIterator

```
/*
;---------------------------------------------------------------------;
;       make evaluator                                               ;
;---------------------------------------------------------------------;
*/
int  _Export custproxy::makeEvaluator(char *where)
  {
   iterator = homeEvaluator(home,where);
   if (iterator == NULL)
      return FALSE;

   cout << "iterator is OK" << endl;
   return TRUE;
  }
```

Figure 73. makeEvaluator

The methods above in Figure 73 set the instance variable *iterator* (by calling the previously described helper methods in the parent class) of type *ICollectionsBase::IIterator_ptr* so it may be used by subsequent **getCustomer** invocations.

```
/*
;----------------------------------------------------------------------;
;       set object iterator                                           ;
;----------------------------------------------------------------------;
*/
int _Export custproxy::setObjIterator(char *qStr)
  {
IString outData("Attempting to create iterator from: \n[");
IExtendedQuery::MemberList* mlst;

   strcpy(select,qStr);
   outData += select;
   cout << outData << "]" << endl;
   eval = getEvaluator("CustomerRTServer");
   if (eval == NULL)
      {
       cout << "Couldn't create evaluator" << endl;
       return FALSE;
      }

   try
      {
       eval->evaluate_to_iterator(select,0,0,0,0,mlst,iterator);
      }
   catch(CORBA::SystemException &se)
      {
       cout << "evaluate to iterator failed - exception: "
            << se.id() << endl;
       return FALSE;
      }
   catch(CORBA::UserException ü)
      {
       cout << "evaluate to iterator failed - exception: "
            << ue.id() << endl;
       return FALSE;
      }
   catch( ... )
      {
       cout << "evaluate to iterator failed " << endl;
       return FALSE;
      }
   return TRUE;
  }
```

Figure 74. setObjIterator

The **setObjIterator** method in Figure 74 takes an OO-SQL query as an input parameter and then creates an evaluator from the server, based on the query. Finally, it uses the evaluator to create the usual iterator to be used with the **getCustomer** method.

```
/*
;--------------------------------------------------------------------;
;       get Customer                                                  ;
;--------------------------------------------------------------------;
*/
Customer_ptr _Export custproxy::getCustomer()
  {
CORBA::Object_ptr aBO;
Customer_ptr      obj;
/*
;------- check if a valid iterator is present -----------------------;
*/
   if (iterator == NULL)
      return NULL;
/*
;------- iterate ---------------------------------------------------;
*/
   aBO = iterate(iterator);
/*
;------- at end? ---------------------------------------------------;
*/
   if (aBO == NULL)
       return NULL;
/*
;------- no - narrow result ----------------------------------------;
*/
   obj = Customer::_narrow( aBO );
   CORBA::release( aBO );
   return obj;
  }
```

Figure 75. getCustomer

We use the **getCustomer** method shown in Figure 75 to do one iteration through our obtained iterator.

```
/*
;--------------------------------------------------------------------;
;       get data iterator                                    ;
;--------------------------------------------------------------------;
*/
IExtendedQuery::DataArrayIterator_ptr
             _Export custproxy::getDataIterator(char *qStr)
 {
IString outData("Attempting to create iterator from: \n[");
IExtendedQuery::QueryEvaluator_ptr eval;
IExtendedQuery::DataArrayIterator_ptr queryIt;
IExtendedQuery::DataArrayList* dal;

   outData += qStr;
   cout << outData << "]" << endl;
   try
     {
      eval = getEvaluator("CustomerRTServer");
      if (eval == NULL)
         return NULL;
      eval->evaluate_to_data_array(qStr,0,0,0,0,dal,queryIt);
     }
   catch(CORBA::SystemException &se)
     {
      cout << "evaluate to iterator failed - exception: "
          << se.id() << endl;
      return NULL;
     }
   catch( ... )
     {
      cout << "evaluate to iterator failed " << endl;
      return NULL;
     }
   return queryIt;
  }
```

Figure 76. getDataIterator

The **getDataIterator**, shown in Figure 76**,** gives us an iterator based on an OO-SQL query. It allows us to iterate through an array of arrays, where the inner array is reflected by the attributes specified in the `WHERE` clause of the OO-SQL query, and the outer array is the number of objects found.

19.4.1.3 The GUI

This is a typical demonstrational GUI, with four almost identical lists that use the four different iterators to populate the lists. Two lists have a filter that let you select customers by their first name.

The GUI uses two nonvisual parts. First, the **CustCBCProxyWrapper** that wraps the **custproxy** object described in the previous section. It defines the **init** method that finds a home and the three transactional methods: **begin**, **commit** and **rollback**.

The **custproxy** object is passed as a parameter to the other nonvisual part, the **IteratorHandler**. The **IteratorHandler** has an attribute for each of the collections shown in the GUI's listboxes. These attributes are collections of **CustomerWrapper** objects, apart from the evaluate to Data Array sample, that displays the objects in a string. The **IteratorHandler** also has actions for generating each of the collections.

The collections are generated by calling the iterator methods on the **custproxy** object. These methods must be called within a transactional scope.

First, an iterator type needs to be created. For some iterator types, an evaluation statement may or must be provided. After the iterator has been successfully created, a **getCustomer** method is called that returns the next Customer Business Object from the iterator. This, in turn, is used as the basis for a new **CustomerWrapper** object, which is added to the collection held in the **IteratorHandler** for this iterator type. We now see why we didn't need the **createBO** and **findBO** methods on the **CustCBCProxyWrapper**; it is because the Business Object is returned when iterating through the iterator.

Run the client by typing the following command in the *VisualC++* directory:

```
CustomerRTC.exe
```

19.4.2 The Java client

All Java source and class files for this sample are located on the CD-ROM in the directory *\CBConnector\18-Reuse-Table\Client\Java*.

19.4.2.1 Client proxies

Generate the Java client proxies as described in 6.7.2, “Java client programming model” on page 76.

19.4.2.2 The CBCBase class

The **CBCBase** class is enhanced with some utility methods to support the use of the Query Service.

First, the method **resolveQueryableHome** is added. See Figure 77.

```
/**
 * resolves the queryable-iterable home of a specific object

 * @return com.ibm.IManagedAdvancedClient.IQueryableIterableHome

 */
public static IQueryableIterableHome resolveQueryableHome(
                FactoryFinder factoryFinder, String objectInterface) {

  com.ibm.IManagedAdvancedClient.IQueryableIterableHome  home = null;
  org.omg.CORBA.Object obj = null;

  try {
    obj = factoryFinder.find_factory_from_string( objectInterface );
    home = IQueryableIterableHomeHelper.narrow(obj);
  }
  catch (Exception e) {
    return null;
  }

  return home;
}
```

Figure 77. Resolving a queryable-iterable home

In order to perform OO-SQL queries for objects, you must first create a query evaluator. We add the utility method **resolveQueryEvaluator** to handle this as shown in Figure 78.

```
/**
 * This method resolves a Query Evaluator for the specified server
 *
 * @return com.ibm.IExtendedQuery.QueryEvaluator
 * @param queryServer java.lang.String  - the server you want to query
 */
public static com.ibm.IExtendedQuery.QueryEvaluator resolveQueryEvaluator (
            String queryServer ) {

  try {
    // build queryEvalutorString -
    //      where the to look in the name tree for the query evaluator
    String queryEvalutorString = "/host/resources/servers/" + queryServer
                                + "/query-evaluators/default";

    // get query evaluator
    com.ibm.IExtendedQuery.QueryEvaluator  queryEvaluator;
    org.omg.CORBA.Object obj = null;
    obj = nameService.resolve_with_string( queryEvalutorString );
    queryEvaluator = com.ibm.IExtendedQuery.QueryEvaluatorHelper.narrow(obj);

    if (queryEvaluator == null) throw new NullPointerException();

    return queryEvaluator;
  }
  catch (Throwable e) {
    return null;
  }
}
```

Figure 78. Resolving a query evaluator for a specific server

19.4.2.3 The CBQueryableProxy class

In this sample, the **Customer** object is queryable; so it resides in a queryable-iterable home. The **CBQueryableProxy** class is similar to the **CBProxy** class, except that the home that it finds and stores is of type **com.ibm.IManagedAdvancedClient.IQueryableIterableHome**. Also, since the objects in this home use the Query Services, this class stores the name of the application server in order to create query evaluators.

19.4.2.4 The CustomerProxy class

The **CustomerProxy** class subclasses the **CBQueryableProxy** class and provides additional helper methods to support OO-SQL queries. These methods are:

- queryAll_iterateOverHome
- queryByFirstName_evaluateOverHome
- queryAll_evaluateToIterator
- queryByFirstName_evaluate_to_data_array

The **queryAll_iterateOverHome** method allows you to create an iterator over all the objects in a queryable home and returns a **Vector** containing those objects as shown in Figure 79.

```
/*
 * This method must be called in the context of a transaction
 *
 * This method uses a Iterator on a home to query for all customers
 * in a home.
 * Returns a Vector of Customer objects
 *
 * @return java.util.Vector
 */
public java.util.Vector queryAll_iterateOverHome( ) {

  java.util.Vector customers = new java.util.Vector();

  try{
    // Create Iterator over the home
    com.ibm.ICollectionsBase.IIterator  customerHomeIterator = null;
    customerHomeIterator = home.createIterator();

    if (customerHomeIterator == null) {
    throw new NullPointerException();
    }

    //now iterate through home for Customers
    com.ibm.IManagedClient.IManageable mo;
    Customer customer = null;

    while( customerHomeIterator.more() ) {
    mo = customerHomeIterator.next();
    customer = CustomerHelper.narrow( mo );

    if (customer != null)
      customers.addElement(customer);
    }

    return customers;
  }
  catch (Throwable e) {
     return null;
  }
}
```

Figure 79. Iterating over a home using createiterator()

The method **queryByFirstName_evaluateOverHome** demonstrates how to create an iterator over the objects in a home based on a selection criteria passed as a parameter to the **evaluate** method. The selection used here always creates an iterator over **Customers** with a specific first name. A **Vector** of **Customers** is returned. See Figure 80.

```
/*
 * This method must be called in the context of a transaction
 *
 * This method uses a Iterator on a home to query for all customers
 * in a home with a specific first name. The Iterator is created
 * by calling evaluate() on the IQueryableIterableHome. evaluate() creates
 * a reference collection of objects based on the predicate passed in.
 * The iterator returned is an iterator over this reference collection.
 *
 * Returns a Vector of Customer objects
 *
 * @return java.util.Vector
 */
public java.util.Vector queryByFirstName_evaluateOverHome(String name) {

  java.util.Vector customers = new java.util.Vector();

  try {
    //create iterator on Customer home using evaluate()
    String predicate = "firstName = '" + name + "';";
    com.ibm.ICollectionsBase.IIterator  customerHomeIterator = null;
    customerHomeIterator = home.evaluate( predicate );

    if (customerHomeIterator == null) {
    throw new NullPointerException();
    }

    //now iterate through home for Customers
    com.ibm.IManagedClient.IManageable mo;
    Customer customer = null;

    while( customerHomeIterator.more() ) {
      mo = customerHomeIterator.next();
    customer = CustomerHelper.narrow( mo );

    if (customer != null)
         customers.addElement(customer);
    }

    return customers;
   }
   catch (Throwable e) {
    return null;
   }
}
```

Figure 80. Iterating over a home using evaluate()

The method **queryAll_evaluate_to_iterator** shows how to use the query evaluator method **evaluate_to_iterator**. See Figure 81.

```
public java.util.Vector queryAll_evaluate_to_iterator ( ) {

  java.util.Vector customers = new java.util.Vector();

  try {
    // create query evaluator
    com.ibm.IExtendedQuery.QueryEvaluator queryEvaluator =
              CBCBase.resolveQueryEvaluator( this.queryServer );

    com.ibm.ICollectionsBase.IIterator queryIterator;
    com.ibm.ICollectionsBase.IIteratorHolder queryIteratorHolder =
              new com.ibm.ICollectionsBase.IIteratorHolder() ;

    com.ibm.IManagedClient.IManageable mo[] = null ;
    com.ibm.IExtendedQuery.MemberListHolder moHolder =
              new com.ibm.IExtendedQuery.MemberListHolder();
    org.omg.CosQueryCollection.NVPair nullPair[] =
              new org.omg.CosQueryCollection.NVPair[0];

    CBCBase.traceIt("Calling evaluate_to_iterator() ...");
    String query = "select e from CustomerMOHome e;";

    queryEvaluator.evaluate_to_iterator(
                              query,
                              null,
                              nullPair,
                              nullPair,
                              0,
                              moHolder,
                              queryIteratorHolder);

    // Now pull the values out of their respective Holders.
    mo = moHolder.value;
    queryIterator = queryIteratorHolder.value;

    if (queryIterator == null ) {
     throw new NullPointerException();
     }
     else {
     CBCBase.traceIt("Query results..." );
     com.ibm.IManagedClient.IManageable tup;
     com.ibm.IManagedClient.IManageableHolder tupHolder =
                new com.ibm.IManagedClient.IManageableHolder();

     while(queryIterator.nextOne(tupHolder))  {
            tup = tupHolder.value;
       Customer customer = null;
       customer = CustomerHelper.narrow(tup);
       customers.addElement(customer);
      }
    }
    queryIterator.remove();

    return customers;
  }
  catch (Throwable e) {
    return null;
  }
}
```

Figure 81. Using evaluate_to_iterator method of the query evaluator

The method **queryByFirstName_evaluate_to_data_array** shows how to use the query evaluator method **evaluate_to_data_array**. See Figure 82 and Figure 83.

```
- Part 1

/**
 * This method must be called in the context of a transaction
 *
 * This method uses a QueryEvaluator to query for all customers
 * in the queryServer using evaluate_to_data_array().
 * Returns a Vector of Strings, each String contains all
 *                      the attributes of a Customer
 *
 * @return java.util.Vector
 */
public java.util.Vector queryByFirstName_evaluate_to_data_array(String name){

  java.util.Vector customerData = new java.util.Vector();

  try {
    // create query evaluator
    com.ibm.IExtendedQuery.QueryEvaluator queryEvaluator =
             CBCBase.resolveQueryEvaluator( this.queryServer );

    com.ibm.ICollectionsBase.IIterator queryIterator;
    com.ibm.ICollectionsBase.IIteratorHolder queryIteratorHolder =
             new com.ibm.ICollectionsBase.IIteratorHolder() ;

    // now do a query using query data arrays
    com.ibm.IExtendedQuery.DataArrayIterator dataArrayIterator;
    com.ibm.IExtendedQuery.DataArrayIteratorHolder dataArrayIteratorHolder =
             new com.ibm.IExtendedQuery.DataArrayIteratorHolder();
    org.omg.CORBA.Any dataArrayList[][] = null;
    com.ibm.IExtendedQuery.DataArrayListHolder dataArrayListHolder =
             new com.ibm.IExtendedQuery.DataArrayListHolder() ;
    org.omg.CosQueryCollection.NVPair nullPair[] =
             new org.omg.CosQueryCollection.NVPair[0];

    String query = "select e.customerNumber, e.firstName, e.lastName " +
          "from CustomerMOHome e where e.firstName = \'" + name + "\';";

    CBCBase.traceIt("Calling evaluate_to_data_array() ...");
    queryEvaluator.evaluate_to_data_array(
                           query,
                           null,
                           nullPair,
                           nullPair,
                           0,
                           dataArrayListHolder,
                           dataArrayIteratorHolder);

     dataArrayIterator = dataArrayIteratorHolder.value;
     dataArrayList = dataArrayListHolder.value;

     if ( dataArrayIterator == null ) {
        return null;
    }
```

Figure 82. Using evaluate_to_data_array method of the query evaluator

```
- Part 2

CBCBase.traceIt("DataArray query result... " );
    org.omg.CORBA.Any tup[] = null;
    com.ibm.IExtendedQuery.DataArrayHolder tupHolder =
              new com.ibm.IExtendedQuery.DataArrayHolder();

    while (dataArrayIterator.next_one(tupHolder)) {
     tup = tupHolder.value;

     String customerNumber = tup[0].extract_string();;
     String firstName = tup[1].extract_string();
     String lastName = tup[2].extract_string();
     customerData.addElement( firstName.trim() + " " +
              "lastName.trim() + " " + customerNumber.trim());
     }

   return customerData;
  }
  catch (Throwable e) {
   return null;
  }
}
```

Figure 83. Using evaluate_to_data_array method of the query evaluator

19.4.2.5 The GUI

This is a typical demonstrational GUI, with four almost identical lists that use the four different iterators to populate the lists. Two lists have a filter that let you select customers by their first name.

The collections are generated by calling the iterator methods on the **CustomerProxy** object. These methods must be called within a transactional scope.

19.4.2.6 Run the client

You can run this client as an application and as an applet as you have in the previous samples.

To run the client application, execute the command:

```
java CustomerQueryApp <hostname> <port>
                        CustomerRTServer-server-scope CustomerRTServer
```

To run the client as an applet, embed the applet in an HTML page (*ReuseTable.html*) with the same parameters as the application, and open that page in the applet viewer with the command:

```
appletviewer ReuseTable.html
```

If you use the sample command (CMD) files provided on the CD, remember to change the `hostname` and `port` parameters (in *runQueryApplet.cmd* and *ReuseTable.html*) to reflect your bootstrap server.

19.5 What did you learn?

In this chapter, you learned how to re-use an existing database table by mapping Business Object attributes to the appropriate table columns. Also, you learned how to make an object queryable and learned various methods for querying for those objects. You also saw how to iterate over a home and how to perform "push-down" queries.

Chapter 20. Meet-in-the-middle paradigm with two datastores

In this chapter, we continue with our meet-in-the-middle approach by extending the previous sample to use two datastores. We demonstrate how a client program can use Managed Objects that have their essential state stored in different databases. We also show the use the Query Service to query both databases for collections of objects.

You can find the Object Builder model, code, and client applications for this sample on the CD for this book in the directory *\CBConnector\19-Two-Datastores*.

20.1 What you will learn

In this sample, you will learn how to create an object model containing two objects, **Customer** and **CustomerAccount**, which use two different databases for their backend stores.

You will re-use two simple tables: one CUST table containing customer data, and one ACCNT table containing each customer's accounts. The **Customer** object uses the CUST table, which resides in a DB2 database named CustDB; and the **CustomerAccount** uses the ACCNT table, which resides in a database named CActDB.

20.2 CBConnector componentry

Figure 84 shows the CBConnector componentry involved in this sample.

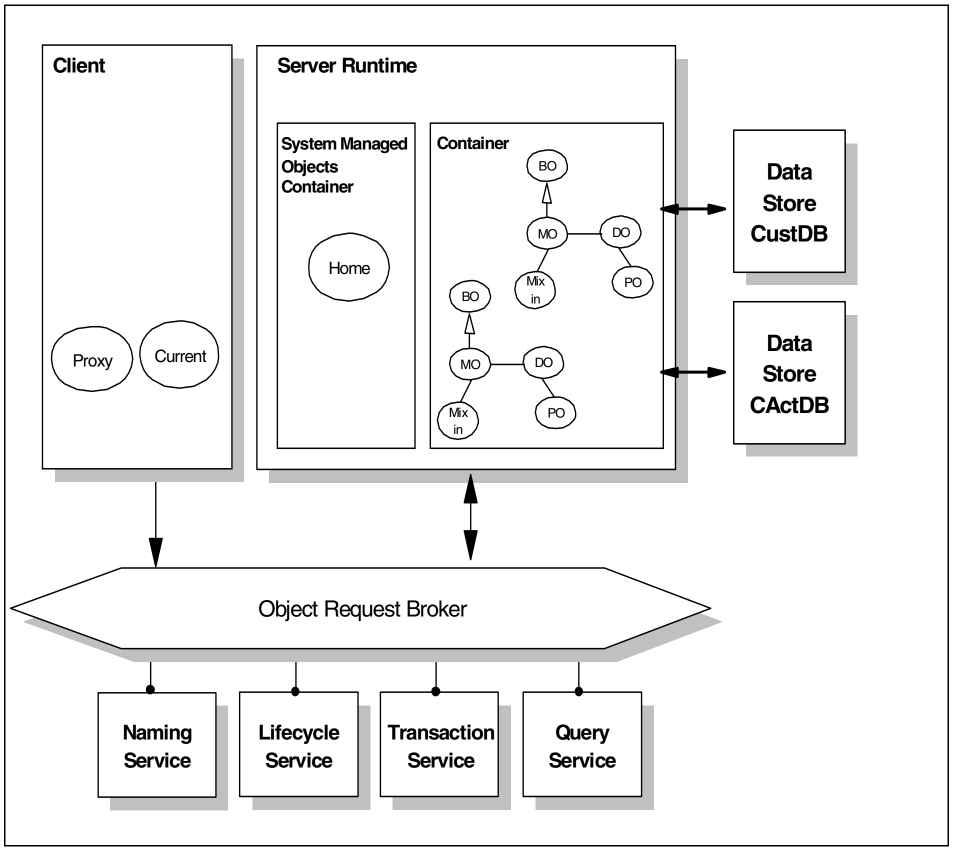


Figure 84. Two business objects in two different datastores

20.3 Server

This section takes you through all the steps you need to perform to build, package, and install a server application that uses an existing DB2 table for persistence.

20.3.1 Build

This sample assumes that you are familiar with the sample described in Chapter 19, “Query Service — reuse of an existing table” on page 291. All the details will not be pointed out. It is assumed that by now you know how to create Managed Objects in Object Builder.

Through the Object Builder walkthrough, we provide you with the correct parameter settings for the SmartGuides. If no suggestions are made by us for a SmartGuide page, use the default settings.

We strongly recommend you open help for each field of the SmartGuide pages in order to understand more of the dialectic behind the input fields.

20.3.1.1 Create the databases

The sample uses an existing DB2 table named CUST in a database called CustDB, which looks like this:

Column name	*Type*	*Length*	*Description*
CUSTNO	CHARACTER	10	Customer number
FNAM	CHARACTER	30	First name
LNAM	CHARACTER	30	Last name

It also uses an existing DB2 table named ACCNT in a database called CActDB, which looks like this:

Column name	*Type*	*Length*	*Description*
ACCTNO	CHARACTER	10	Account number
CUSTNO	CHARACTER	10	Account holder
BALANCE	DOUBLE		Balance

The `CUSTNO` column is a foreign key to the CUST table. We use it when searching for the accounts held by a customer.

Again, since this is a sample, you will have to create the tables. There are SQL scripts included on the CD that will create the databases, create the tables, and load them with data. If you already have a database called CustDB and/or CActDB that you do not want to destroy, use other database names in the steps described below. In that case, you must also remember to change the scripts before you run them.

To install these two databases and tables, do the following:

Open a DB2 Command Window and change directory to the *\CBConnector\19-Two-Datastores\Server\DBscript\* directory on the CD.

Issue the commands:

```
db2 -f creCust.db2
db2 -f creAccnt.db2
```

20.3.1.2 Create the SQL scripts

The Object Builder needs a definition of the each of these tables to create schemas. We have provided you with these table definitions in files named *creCust.db2* and *creAccnt.sql* on the CD-ROM and located in the directory *\CBConnector\19-Two-Datastores\Server\DBscript*.

20.3.1.3 Create the model

1. Create a new Object Builder model in a separate directory.
2. Create a **Customer** exactly as described in Chapter 19, “Query Service — reuse of an existing table” on page 291, except for the parameters in the following table:

Hint

Instead of re-creating the Customer from the previous chapter, you can export its model in XML format, then import it into the model for this sample and make the changes to reflect the settings in the table below. The XML files defining the Customer model can also be found on the CD-ROM in *\CBConnector\18-Reuse-Table\Server\Export*.

Parameter	**Value**
Add schema group: File Name	CBConnector\19-Two-Datastores\ Server\DBScript\Cust.sql
Add Persistent Object: Package File	CustTD (TD=Two Datastores)
Add BO Implementation: Pattern for Handling State Data	Caching
Add DO Implementation: Data Access Pattern	Local Copy

3. Create a **CustomerAccount** in the same way as you created the **Customer** above, except for the following differences (obvious name changes are not included).

- When importing the SQL statement, use the following parameters:

Parameter	Value
File Name	CBConnector\19-Two-Datastores\Server\DBScript\Accnt.sql
Group Name	CustomerAccountSchemaGroup
Database Name	CActDB (change if necessary)
Statements to Import	CREATE TABLE ACCNT

- When adding the Persistent Object to the schema, `CACTDB.USERID.ACCNT`, use the following parameters:

Parameter	Value
Name	CustomerAccountPO
Package File	CActTD (TD=Two Datastores)
Type of Persistence	Embedded SQL
Mapping of Column to PO Attribute	Change **ACCNTNO** to **accountNumber**
Mapping of Column to PO Attribute	Change **CUSTNO** to **accountHolder**
Mapping of Column to PO Attribute	Change **BALANCE** to **balance**
PO Key	accountNumber

- Add attributes to the queryable **CustomerAccount** interface as described in the following table:

Parameter	Type	Implementation
accountNumber	string <10>	Public, Read-Only
accountHolder	string <10>	Public
balance	double	Public

Note that, in this sample, we do not care about any extra methods on **CustomerAccount** to retrieve and change the balance. The purpose is only to get two queryable interfaces that are loosely related. In real life, there would probably be an object relation between **Customer** and **CustomerAccount**.

- For the **CustomerAccount** Key Helper, use `accountNumber` as the Primary Key attribute.

20.3.1.4 Build the server application

1. Add a client DLL named CustomerTDC with the following parameters:

Parameter	Value
Name	CustomerTDC
Client Source Files	all Customer files
LIbraries to Link With	none

2. Add a server DLL named CustomerTDS with the following parameters:

Parameter	Value
Name	CustomerTDS
Server Source Files	all Customer files
Libraries to Link With	CustomerTDC (the client library)

3. Add a client DLL named CustomerAccountTDC with the following parameters:

Parameter	Value
Name	CustomerAccountTDC
Client Source Files	all CustomerAccount files
Libraries to Link With	none

4. Add a server DLL named CustomerAccountTDS with the following parameters:

Parameter	Value
Name	CustomerAccountTDS
Server Source Files	all CustomerAccount files
Libraries to Link With	CustomerAccountTDC (the client library)

5. Generate the makefiles for all targets.

   Before you build the server application, make sure you have created the CustDB database, CUST table, CActDB database, and ACCNT table needed for this sample. If you have not done so, please follow the steps in 20.3.1.1, “Create the databases” on page 325.

6. Build all out-of-data C++ targets.

20.3.2 Package

This section describes how to package the distributed application.

20.3.2.1 Define the application

1. Add a two container instances, **ContainerOfCustomerTD** and **ContainerOfCustomerAccountTD**, with the following parameters:

Parameter	Value
Use Transaction Services	yes
Behavior for Methods Called Outside a Transaction	Throw an exception and abandon the call
Passivate a component at end of transaction	yes
Data Access Pattern Business Object	Caching
Data Access Pattern Data Object	Local Copy

2. Add an application family named **CustomerTDAppFam**.

3. For this application family, add a server application named **CustomerTDAppS**.

4. Add a Managed Object for the **CustomerTDAppS** with the following parameters:

Parameter	Value
Managed Object	CustomerMO CustomerMO
Primary Key	CustomerKey CustomerKey
Copy Helper	CustomerCopy CustomerCopy
Data Object Implementation	CustomerDOImpl CustomerDOImpl
Container	ContainerOfCustomerTD
Default Home	Default Home
Home Name	BOIMHomeOfRegHomes
Name in Factory Finding Service Registry	CustomerMOFactory
Name in Naming Service Registry	CustomerMOHome

5. Add another application family named **CustomerAccountTDAppFam**.
6. For this application family, add a server application named **CustomerAccountTDAppS** .
7. Add a Managed Object for the **CustomerAccountTDAppS** with the following parameters:

Parameter	Value
Managed Object	CustomerAccountMO CustomerAccountMO
Primary Key	CustomerAccountKey CustomerAccountKey
Copy Helper	CustomerAccountCopy CustomerAccountCopy
Data Object Implementation	CustomerAccountDOImpl CustomerAccountDOImpl
Container	ContainerOfCustomerAccountTD
Default Home	Default Home
Home Name	BOIMHomeOfRegHomes

Parameter	Value
Name in Factory Finding Service Registry	CustomerAccountMOFactory
Name in Naming Service Registry	CustomerAccountMOHome

20.3.2.2 Create an installation image

1. Generate the installation scripts for each application family.
2. Build the installation images for each application family.

20.3.3 Install

To install and configure the server applications, follow these steps:

1. Bind the Customer server application to the CustDB database and the CustomerAccount application to the CActDB database. The bind files created in your *Working* directory are named *CustPO.bnd* and *CusAccPO.bnd*. Issue the following set of commands from your *Working* directory within a DB2 command window:

```
DB2 connect to CustDB
DB2 bind CustPO.bnd
DB2 connect reset
DB2 connect to CActDB
DB2 bind CusAccPO.bnd
DB2 connect reset
```

2. Bind the two CBConnector bind files needed for the use of the Query Service to both databases. These files are named *db2cntcs.bnd* and *db2cntrr.bnd* and are located in the directory *C:\CBroker\etc* (where *C:\CBroker* is the directory where you installed CBConnector). **This is a very important step. You must do this to perform queries.**

   Issue the following set of commands from this directory within a DB2 command window (xxxxDB = CustDB, CActDB):

```
DB2 connect to xxxxDB
DB2 bind db2cntcs.bnd
DB2 bind db2cntrr.bnd
DB2 connect reset
```

3. Install both server applications on your CBConnector server machine.

4. Configure a **CustomerAccountTDServer** in a new Management Zone, **CustomerAccountTD Management Zone**. Use a new configuration, **CustomerAccountTD Configuration**, for this server. In this server, configure the applications CustomerTDAppS, Specific CustomerTDAppS, CustomerAccountTDAppS, Specific CustomerAccountTDAppS, and the default DB2 application named iDB2IMServices.
5. Activate the configuration.
6. Stop the **CustomerAccountTDServer**, and set the **open string** with the database administrator user ID and password for the databases in the **XA Resource Manager images** of this server.
7. Start the server using `Run Immediate`.

20.3.4 The C++ client

We remember that the **custproxy** class is the application-dependent class, a subclass of the **cbcproxy** class, which hides the CBConnector specifics from the client application.

20.3.4.1 The custproxy

In this sample, we re-use the **custproxy** class from Chapter 19, “Query Service — reuse of an existing table” on page 291. All the methods, except the **setObjIterator** are used in this sample.

20.3.4.2 The acctproxy

We remember that the **acctproxy** class is the application-dependent class, a subclass of the **cbcproxy** class, which hides the CBConnector specifics from the client application. The **acctproxy** class is constructed from the **custproxy** class by using the *CustomerAccount.hh* header file rather than the *Customer.hh* header file. We renamed the **getCustomer** method to **getAccount**.

20.3.4.3 The GUI

The GUI shows a list of customers and accounts and lets you see a list of accounts for a selected customer.

The two datastores sample incorporates two Business Objects and needs two Business Object wrappers, the **AccountWrapper** and the **CustomerWrapper**, and two CBCProxyWrappers, **AcctCBCProxyWrapper** and **CustCBCProxyWrapper**. Both proxy wrappers need to be initialized when the application starts. We also include an **IteratorHandler** similar to the one in the Re-use Table sample.

The **IteratorHandler** has four attributes, a holder for each proxy wrapper, a holder for the collection of **CustomerWrapper** objects and finally a holder for the collection of **AccountWrapper** objects. It implements three actions:

- Retrieves all customers by the **makeIterator** method.
- Retrieves all accounts by the **makeIterator** method.
- Retrieves accounts by the **makeEvaluator** method with the customerNumber of the selected customer as input parameter.

Run the client by typing the following command in the *VisualC++* directory:

```
TwoDataC.exe
```

20.3.5 The Java client

All Java source and class files for this sample are located on the CD-ROM in the directory *\CBConnector\19-Two-Datastores\Client\Java*.

20.3.5.1 Generate Client Proxies

For this sample, you need to generate the proxies for the **Customer** and **CustomerAccount** Interfaces as described in 6.7.2, “Java client programming model” on page 76.

20.3.5.2 The CustomerProxy class

In this sample, we re-use the **CustomerProxy** class from 19.4.2, “The Java client” on page 314.

20.3.5.3 The CustomerAccountProxy class

The **CustomerAccountProxy** class is implemented the same as the **CustomerProxy** class, except it, of course, handles **CustomerAccounts**. It provides two methods almost identical to those in **CustomerProxy** used to iterate over the home:

- **queryAll_iterateOverHome** — iterates over the home and returns a **Vector** of all the **CustomerAccounts** in the home.
- **queryByHolder_evaluateOverHome** — uses **evaluate** to iterate over the **CustomerAccounts** in the home which have a specified accountHolder.

20.3.5.4 The GUI

The GUI shows a list of **Customers** and **CustomerAccounts** and lets you see a list of **CustomerAccounts** for a selected **Customer**.

The collections are generated by calling the iterator methods on the **CustomerProxy** and **CustomerAccountProxy**. These methods must be called within a transactional scope.

20.3.5.5 Run the client

You can run this client as an application and as an applet, just as you have in the previous samples.

To run the client application, execute the command:

```
 java CustomerAccountQueryApp <hostname> <port>
CustomerAccountTDServer-server-scope CustomerAccountTDServer
```

To run the client as an applet, embed the applet in an HTML page (*TwoDatastores.html*) with the same parameters as the application, and open that page in the applet viewer with the command:

```
appletviewer TwoDatastores.html
```

If you use the sample command (CMD) files provided on the CD-ROM, remember to change the `hostname` and `port` parameters (in *runApplet.cmd* and *TwoDatastores.html*) to reflect your bootstrap server.

20.3.6 The ActiveX client

Refer back to 10.4, “ActiveX implementation” on page 157, for an explanation of how implement your client and how to find the Business Object and invoke its methods. This section contains only the specific changes that has to be made to the general case. The differences comes from the fact that we are dealing with two datastores.

You will find all the source files as shown in :tref refid=tabfsum14. used for the code snippets of this section in the following directory:

```
\CBConnector\19-Two-Datastores\Client\ActiveX
```

20.3.6.1 Initializing

Initialization of the client is similar to the general ActiveX client. The table below summarizes the changes to the variables when initializing the ORB and finding the Factory Finder.

Parameter	Value
scope	"AccountTXServer-server-scope"

20.3.6.2 Finding the homes

For two datastores, we have two homes for two different objects.

```
Dim CustomerHome As Object
Dim AccountHome As Object
' using IQueryableIterableHome
Set CustomerHome = CreateObject("IDL:IManagedAdvancedClient.IQueryableIterableHome")
Set AccountHome = CreateObject("IDL:IManagedAdvancedClient.IQueryableIterableHome")
...
' Use the factory finder to find the Customer Home
Set anObject = factoryFinder.find_factory_from_string("Customer.object interface")
' found the Home!
CustomerHome.narrow anObject
' Use the factory finder to find the Account Home
Set anObject = factoryFinder.find_factory_from_string("CustomerAccount.object interface")
' found the Home!
AccountHome.narrow anObject
```

20.3.6.3 Loading the accounts

Before being able to access the home, you have to begin a transaction and create the iterator for all accounts. Then go through the list of accounts using **next**.

```
CosTransactions_Current.begin e
...
Dim Iterator as Object
Set Iterator = CreateObject("IDL:ICollectionsbase.IIterator")
Set Iterator = AccountHome.CreateIterator(e)
Set anObject = Iterator.Next()
While (Not (anObject Is Nothing))
 AccountObject.narrow anObject
 ' do something with the object...
  Set anObject = Iterator.Next()
Wend
CosTransactions_Current.commit 1, e
```

For the customers, the code is exactly the same except for the variable names of the home and the object.

20.3.6.4 Getting the accounts by using a query

Another possibility to get a number of objects is to make a query. This is a better way if you have many objects and want to avoid searching the whole set. The account home is given a predicate to evaluate and returns an iterator object. Once you have the iterator, the code is the same as without the query.

```
CosTransactions_Current.begin e
...
Set Iterator = CreateObject("IDL:icollectionsbase.IIterator")
Set anObject = AccountHome.evaluate("accountHolder='" + num + "';")
Iterator.narrow anObject
...
Set anObject = Iterator.Next()
While (Not (anObject Is Nothing))
 AccountObject.narrow anObject
  ' do something with the object
  Set anObject = Iterator.Next()
Wend
CosTransactions_Current.commit 1, e
```

Here, `"accountHolder='" + num + "';"` indicates the attribute and its value to search for.

20.3.6.5 Building a client install image

In the following steps, you build the client install image:

Invoke `init customer` to generate the includes for the makefile (on the CD-ROM, we have placed the Customer and CustomerAccount files in separate directories).

Execute `nmake`.

Register the DLLs with *register.bat*.

Do likewise for CustomerAccount (`init customeraccount` and so on).

20.3.6.6 File cross reference

The following files are involved in two datastores client development.

File	Description
CustomerAccount.dll, Customer.dll	Dynamic Link Library containing Client runtime bindings
CustomerAccountKey.dll, CustomerKey.dll	Dymanic Link Library containing runtime bindings for Key Proxy
CustomerAccountCopy.dll, CustomerCopy.dll	Dynamic Link Library containing runtime bindings for Key Proxy
register.bat	Batch file used to register the Account, AccountKey and AccountCopy DLLs.
unregister.bat	batch file used to unregister the Account, AccountKey and AccountCopy DLLs.

20.3.6.7 Using the ActiveX CD sample

When you run the ActiveX Two Datastores sample (TDX.exe), a window with three text boxes (All Customers , All Accounts, Accounts for Selected Customer) pops up.

Press the **Update** button for the customers. The All customers box is updated. Do the same for the accounts. Select a customer and press the **Get Customer Accounts** button. The client matches the accounts with the customer and updates the list. You have just queried two databases for accounts that belong to a specific customer.

20.4 What did you learn?

In this chapter, you learned that a client program can use Managed Objects that have essential state data stored in different databases.

Chapter 21. Deploying Enterprise JavaBeans

This chapter walks you through deploying an EJB in the Component Broker runtime environment.

You can find the Object Builder model, code, and client applications for this sample on the CD-ROM for this book in the directory *\Samples\EJB*.

21.1 What you will learn

In this chapter, you will learn how to deploy container-managed entity EJBs, as well as some advantages of running EJBs in the Component Broker runtime.

In Chapter 19, “Query Service — reuse of an existing table” on page 291, and Chapter 20, “Meet-in-the-middle paradigm with two datastores” on page 323, you were introduced to the Query Service and the different ways to use it. In this chapter, you will learn how your EJBs can take advantage of the Query Service.

In Chapter 14, “Transactional Object sample” on page 217, you learned how to control transactions from CORBA clients. In this chapter, you will learn how to have client-demarcated transactions from EJB clients.

This sample walks you through deploying a simple Customer entity bean which has several "finder" methods defined in the home interface. This entity bean is persisted with a DB2 table and is very similar to the Customer CORBA object you created in Chapter 20, “Meet-in-the-middle paradigm with two datastores” on page 323; however, in this sample we use a slightly different table definition in order to illustrate additional features of the Query Service.

21.2 CBConnector componentry

Figure 85 shows the main components used in our Customer EJB scenario.

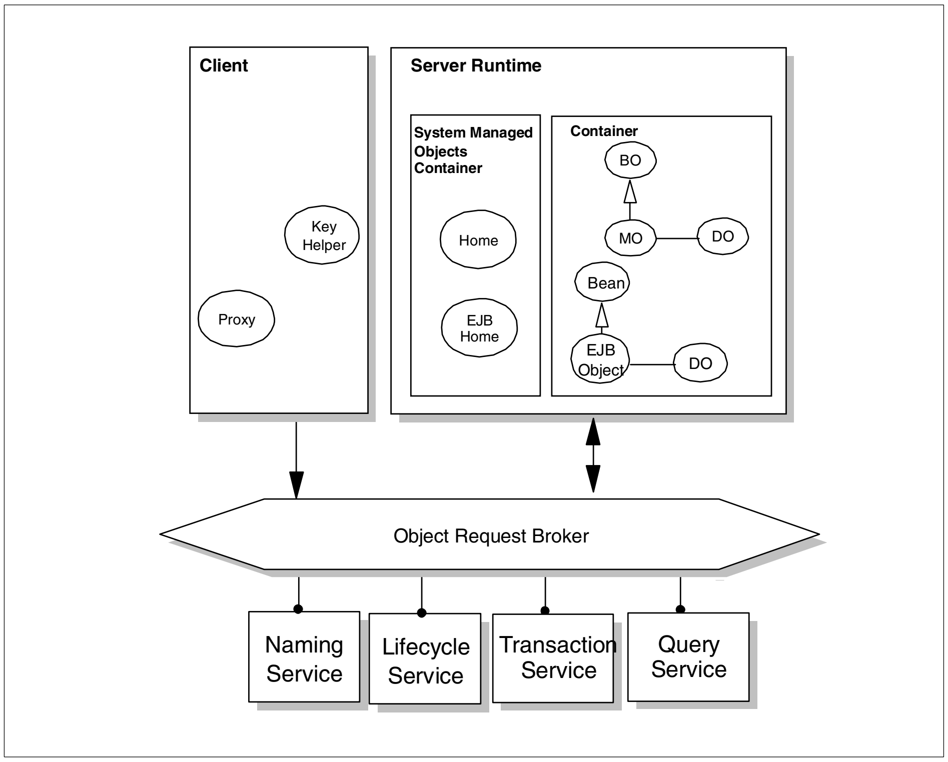


Figure 85. Enterprise JavaBeans in Component Broker

You can find a wealth of information about Enterprise JavaBeans support in Component Broker from the white paper "Writing Enterprise Beans in WebSphere" that comes with the documentation for the product. Most of the topics covered in this section can be found in more detail in this paper. In this sample, we present the basic ideas, and focus more on the actual steps you take to deploy an EJB in this environment.

21.2.1 EJBs as Managed Objects

If you have gone through the chapters in this book up to here, then you are familiar with what a Managed Object is. In Component Broker, a Managed Object is a complete component living in the runtime environment that is "managed" by the container in which it lives and the Managed Object Framework, that provides a number of qualities of services, such as persistence, transactionality, security, and workload management.

Each Managed Object simply represents a Business Object (BO) that has been configured with a Data Object Implementation (DOImpl) into a container that is specific to the type of the persistence the DOImpl implements. So far, the Managed Objects that you have been working with represent CORBA objects. This Managed Object Framework that provides Component Broker with the means of serving CORBA objects is so similar to the EJB architecture, that EJBs inherently fit into this infrastructure.

EJBs in Component Broker are, in fact, Managed Objects that can take advantage of the same qualities of service provided to CORBA components in this environment. Let's look at how an EJB maps almost directly to the Managed Object Framework. According to EJB architecture, an EJB consists of a remote interface, a bean class, a key class, a home interface, and the runtime environment to provide the EJBObject, whose job it is to delegate calls to the actual bean class, as well as a container for these beans to live in.

In Component Broker, the remote interface is represented as an IDL interface; the bean class is essentially the Business Object (BO); the key class is the business object Primary Key; the home interface is implemented as a specialized home that inherits from IHome or IQueryableIterableHome, depending on if the bean is queryable or not; the EJBObject *is* the Managed Object (MO), and each MO is configured into a container. When the EJB is a container-managed entity bean, all of the container-managed attributes of the bean are represented as a Data Object (DO).

So, CORBA objects and EJBs both live in the Component Broker runtime as Managed Objects and can take advantage of all the same services provided by the environment. The only real difference between the two is the client programming model that you use to access these components.

21.2.2 EJB query

As explained above, an EJB home interface is implemented in Component Broker as a specialized home that inherits from the framework interfaces, IHome or IQueryableIterableHome. If you specify your EJB to be queryable (you will see how to do this later), then the EJB home will inherit from IQueryableIterableHome, and you can perform the same types of queries over this home as you have learned how to do for CORBA object homes in previous chapters.

Where you perform queries over an EJB home is the only difference between querying over CORBA object homes. Instead of having clients directly query over an EJB home directly, you define special "finder" methods on the home interface that return a collection of beans. Since it is up to the vendor to provide the implementation of the home interface, Component Broker uses what are called finder helper classes that allow you implement these "finder" methods. The finder helper class contains a reference to the queryable EJB home, and it is in the "finder" method implementations where you perform the query over the home.

The normal means of querying over a home in the finder helper class is by calling the **evaluate** method that takes an OOSQL predicate and returns an enumeration of objects that meet the conditions of the query. The finder helper methods can actually return an enterprise bean key, a Component Broker business object key, an EJBObject, or a Component Broker Managed Object (IManageable) or an enumeration of any of these types.

The predicate that you pass to the **evaluate** method can use all of the standard conditional statements supported by OOSQL. For example, in this sample, the finder helper passes in predicate statements such as:
"lastName_ = 'Bob' AND firstName_ LIKE 'Dylan%'"
and "customerNumber_ >= 100 AND customerNumber_ <= 200"

Notice that the bean attribute being queried, such as "lastName_" has an underscore appended to the name of the attribute. The actual attribute name is "lastName", but when the object model for your EJB gets created by the **cbejb** tool, underscores are appended to attribute names in the Business Object (BO) for purposes of mapping these names to IDL. In Component Broker version 3.0, make sure you append these underscores to attribute names in the OOSQL expressions for querying over EJB homes. Why this is necessary is explained in further detail in *Writing Enterprise Beans in WebSphere*, SC09-4431.

Note: In future releases, beginning with version 3.5, due out later in 2000, these underscores will not be necessary, and you can query over the attribute names as you would expect.

The ability to query EJBs is just one example of how the Component Broker runtime environment provides additional services above what the EJB specification defines.

21.2.3 Client-managed transactions

In the EJB programming model, it is most common to control transactions from session beans or have the container handle the transactions for you. However, the Java Transaction API (JTA) specification defines a means of demarcating transactions from the client applications that Component Broker implements.

Using JNDI, the client program must obtain a reference to the interface **javax.transaction.UserTransaction**. JTA does not define the JNDI name for

UserTransaction, so Component Broker uses "jta/usertransaction" for the JNDI lookup name. The following code sample shows how to demarcate client-managed transactions:

```
javax.transaction.*;
...
// Use JNDI to locate the UserTransaction object
Context initialContext = new InitialContext();
UserTransaction transaction = (
UserTransaction)initialContext.lookup("jta/usertransaction");
// Set the transaction timeout to 30 seconds
transaction.setTransactionTimeout(30);
...
// Begin a transaction
transaction.begin();
// Perform transaction work invoking methods on enterprise bean references
...
// Call for the transaction to commit
transaction.commit();
```

The client provided for this sample shows you a complete example of managing transactions in this way.

For more information on JTA and the Java Transaction Services (JTS), see Chapter 22, “Mixing CORBA objects and Enterprise JavaBeans” on page 355.

21.3 Server

This section describes the steps you need to perform to deploy a container-managed EJB into the Component Broker runtime.

Deploying an EJB in Component Broker is somewhat simpler that creating a CORBA object. Essentially, you run your EJB through a deployment tool called **cbejb** which creates for you the Object Builder object model that represents your EJB as a Managed Object (as discussed above in section 21.2.1, “EJBs as Managed Objects” on page 340).

You then compile this object model (just as you would for CORBA objects); then install your application that is packaged to contain your EJB (MO). The only time you ever see the Object Builder is if your EJB is a container-managed entity bean, then Object Builder will open at the appropriate time during the **cbejb** deployment process for you to provide a mapping between the Data Object implementation (DOImpl) and a Persistent Object (PO) that represents the appropriate backend datastore. In the case of deploying session beans, you never see the Object Builder.

21.3.1 Build

The EJB that you are about to deploy has already been created for you. The EJB jar file containing this bean is located on the CD-ROM for this book in *\Samples\EJB\Server\Customer.jar*. There are also some .bat files in this same directory which will simplify some of the following steps for you.

You can find the source for this EJB in *\Samples\EJB\Server\Source*. The beans will be located in the package structure "cb\ejb\samples\customer". Also, we have provided the VisualAge for Java repository file named "FirstSteps EJB Sample.dat" so you can see the beans in the EJB development environment and use the generated test clients from inside of VisualAge for Java to test the bean once you have it deployed in Component Broker.

21.3.1.1 Create the model

1. Create a new project directory in which you want to put your Object Builder object model.
2. Copy the Customer.jar into your project directory (from *\Samples\EJB\Server\* on the CD).
3. Because the home interface for the Customer bean has special "finder" methods, you need to provide an implementation for these methods by creating a FinderHelper class.

   To create the CustomerFinderHelper class, execute the following command from a command line (you can find this command in the .bat file named "createCustomerFinderHelper.bat") :

```
java -classpath "%somcbase%\lib\developEJB.jar;%somcbase%\lib\developBEX.jar;
%somcbase%\lib\somojor.zip;%somcbase%\lib\BEXruntime.jar;.\Customer.jar;%classpath%"
com.ibm.ejb.cb.emit.cb.FinderHelperGenerator cb.ejb.samples.customer.CustomerHome
```

   "SOMCBASE" is an environment variable that is created when you install Component Broker, and it specifies the Component Broker install directory. Be sure to include your EJB jar file in the -classpath flag so that the FinderHelperGenerator will find the home interface.

This command will generate a Java file for a CustomerFinderHelper class, which extends **com.ibm.ejb.cb.runtime.FinderHelperBase** and defines method signatures equivalent to the finder methods found in the CustomerHome interface. You now need to implement these finder methods.

4. Implement the finder methods of the **CustomerFinderHelper**.

   findByLastName

```
return evaluate("lastName_ = '" + arg0 + "'");
```

   findByName

```
return evaluate("lastName_ = '" + arg0 + "' AND firstName_ LIKE '" + arg1
+ "%'");
```

   findRangeOfCustomers

```
return evaluate("customerNumber_ >= " + arg0 + "
AND customerNumber_ <= " + arg1);
```

   Notice that all you are doing here is providing an OOSQL predicate string to the **evaluate** method of the **FinderHelperBase** super class. This results in a call to the **evaluate** method on the queryable home that provides the implementation of the CustomerHome interface in the Component Broker Managed Object Framework.

   Also, remember that this **CustomerFinderHelper** class is just that — a normal Java class that extends a superclass. This means that you can perform any logic inside of these methods that you want. For example, the call to **evaluate** returns a **java.util.Enumeration**. You could easily perform some logic on this enumeration and the elements inside of it before returning it. You can also add other members and methods to this class to facilitate any logic you might need for the implementation of these finder methods. If you look at the CustomerFinderHelper for this sample on the CD, you will see that we changed the parameter names in the finder method signatures to make the code for the creation of the query predicate string easier to understand.

5. Compile the **CustomerFinderHelper** class and add it to the Customer.jar file. There are various ways of accomplishing this, but we've created a .bat file ("compileAndAddToJar.bat") to do this for you. This .bat file executes the following commands:

```
md .\temp
javac -d .\temp CustomerFinderHelper.java
cd .\temp
jar -xf ..\Customer.jar
jar -cvf0 CustomerWithFinderHelper.jar cb\ejb\samples\customer
copy CustomerWithFinderHelper.jar ..\
```

Executing these commands will package the CustomerFinderHelper with the contents of the original Customer.jar into a new file called CustomerWithFinderHelper.jar.

6. Create a deployment descriptor for your EJB. Component Broker comes with a tool for creating and editing deployment descriptors called **jetace**.

   Note: If you use VisualAge for Java for developing your EJBs, you can set deployment descriptors properties inside the IDE, and when you export the ejb-jar file, it will already contain the deployment descriptor for your bean. In this case, you can skip this step, unless you want to edit some properties such as the JNDI home name.

   a. To run jetace, execute the command "jetace" from a command line, with no parameters.

   b. Load the CustomerWithFinderHelper.jar file by selecting File->Load in the jetace window. Nothing shows up in the "Current Enterprise Beans" panel, because there is no existing deployment descriptor in this jar file yet.

   c. Press "New" to create a deployment descriptor. The default name for this descriptor will be "UNNAMED_BEAN_1.ser".

   d. Edit the deployment descriptor by selecting "UNNAMED_BEAN_1.ser" and pressing "Edit".

e. Provide the following values for the properties of the deployment descriptor in the ***Basic*** tab:

Property	Value
Deployed Name	Customer.ser (Press "Set" after updating the name)
Enterprise Bean Class	cb.ejb.samples.customer.CustomerBean
Home Interface	cb.ejb.samples.customer.CustomerHome
Remote Interface	cb.ejb.samples.customer.Customer
JNDI Home Name	cb/ejb/samples/customer/Customer

f. **jetace** knows that your EJB is an entity bean. Provide the following values in the ***Entity*** tab:

Property	Value
Primary Key Class	cb.ejb.samples.customer.CustomerKey (Press "Set" after entering the class name)
Container-Managed Fields	Select All

g. Provide the following property value in the ***Transactions*** tab:

Property	Value
Transaction Attribute	TX_REQUIRED

h. Accept all other defaults, and close the deployment descriptor editor.

i. Save updated jar file that now has deployment descriptor defined for it. Select File->Save As, and save this new jar file as "EJBCustomer.jar".

You now have an ejb-jar file ready for deployment into Component Broker.

7. Execute the following **cbejb** command from a command line from your project directory for this sample (you can find this command in the "buildCustomer.bat" file):

```
cbejb EJBCustomer.jar -bean cb.ejb.samples.customer.Customer -queryable
-cacheddb2 -finderHelper cb.ejb.samples.customer.CustomerFinderHelper
```

   This will generate the object model for the Customer bean. The "-queryable" flag indicates that this bean will be using a queryable home; "-cacheddb2" specifies that you want this bean persisted using the DB2 Cache Service; and "-findHelper" specifies the class name of the finder helper that is associated with this bean. Note: the finder helper class must be packaged inside of the ejb-jar file in order for this command to execute successfully.

8. Because this bean is container-managed, Object Builder will open for you to map the container-managed attributes to a backend datastore (in this sample it is DB2). When Object Builder comes up, open the your project (which will be the default project).

9. Map the attributes of the DOImpl, which has already been created for you by the **cbejb** tool, to a Persistent Object (PO). The DOImpl will have the name **cb_ejb_samples_customer_CustomerDOImpl**.

   This step is identical to mapping a DOImpl to a PO for CORBA objects; in fact, you can reuse the same POs for your EJBs as for CORBA objects.

   As you know from previous samples, you can perform the mapping between a DOImpl and PO by either selecting "Add a Persistent Object and Schema" from the DOImpl; or by importing an SQL file that contains a table definition into **DBA-Defined Schemas**, creating a PO from the schema, and then selecting "Select Persistent Object and Schema" from the DOImpl. Either way you want to accomplish this is fine. For this sample we used the second approach.

   a. From **DBA-Defined Schemas**, import the SQL file "Customer.sql". This file can be found in *\Samples\EJB\Server\DBScript* on the CD. Use the following parameters:

Parameter	Value
Group Name	DB2Group
Database	CustDB
Database Type	DB2

b. From **CUSTDB.CUSTOMER**, select "Add Persistent Object". Use the following parameter:

Parameter	Value
Type of Persistence	DB2 Cache Service

c. From **cb_ejb_samples_customer_CustomerDOImpl**, select "Select Persistent Object and Schema", and select the **CUSTOMPO**, which you created in the previous step.

d. Save your object model, then exit Object Builder. This will continue the **cbejb** processing

Once the **cbejb** processing finishes, you are ready to build the server application.

21.3.1.2 Build the server application

The **cbejb** tool creates a client and server DLL configuration for you, named according to your bean name. So, in this sample, you have DLLs named "CustomerClient" and "CustomerServer".

You do not have to open Object Builder again for deploying your bean; however, if you have selected the QuickTest to be included in the makefile for the "Generate All" option, you need to unselect this. By default, this option is not selected, so unless you've changed this setting, you're fine. QuickTest generates clients for CORBA objects, and will fail when trying to compile clients based on the IDL that gets generated for the managed objects that wrapper EJBs in the Component Broker environment.

To build your application, execute the following command from the command line in the "Working\NT" directory under your project directory.

```
nmake -f all.mak
```

21.3.2 Package

The **cbejb** tool creates an application family and defines an application for you that contains your EJB. There is nothing you need to do to package your EJB into an application.

The application family name will be based on the name of your EJB. In this case, the application family will be named "EJBCustomerFamily", and the application under it will be named "CustomerApp".

21.3.3 Install

To install and configure the server application, follow these steps:

1. Install the server application on your server machine, the same as you install applications that contain CORBA objects. Select "Load Application" from Host Images-><your host> in the System Management User Interface, and install the application family that is located in the "*Working\NT\PRODUCTION*" directory.
2. Create a **CustomerEJBServer** in a new Management Zone, **Customer EJB Zone**. Use a new configuration, **Customer EJB Configuration**, for this server.
3. Add the following applications to the configuration, and configure them into the **CustomerEJBServer**:
   CustomerApp
   Specific CustomerApp
   iDB2IMServices
4. Activate the configuration.
5. When the activation is complete, bind the **Customer** EJB to the JNDI name service by executing the following commands (you can find this command in the .bat file "bindCustomer.bat" located on the CD):

```
ejbbind EJBCustomer.jar cb.ejb.samples.customer.Customer
-ORBInitialHost cbdemo.sl.dfw.ibm.com -ORBInitialPort 900
```

   (**Note**: You will need to change the ORBInitialHost and ORBInitialPort values to reflect your host name and port number)

   There is another .bat file on the CD "unbindCustomer.bat" that shows you how to unbind EJBs from the JNDI name tree.

   Make sure you follow the steps in the next section "Create the Database", before running any clients against your EJB.

21.3.4 Create the database

This sample uses a DB2 table named CUSTOMER in a database called CustDB. In Chapter 19, “Query Service — reuse of an existing table” on page 291, you created the CustDB database with a table named CUST. The CUSTOMER table is slightly different.

You can create the table yourself using the SQL file that was generated in your "Working\NT" directory, or we provide an SQL script file on the CD in *\Samples\EJB\Server\DBscript\createCustomer.db2*, that creates this table, and loads some data into the table that is relevant to the sample, so that the finder methods on the CustomerHome interface return something. If you already have the CUSTDB database created, this script will report an error because it first tries to create this database that already exists. You can safely ignore this error.

To run the SQL script file:

1. Open a DB2 Command Window and change directory to the *\Samples\EJB\Server\DBscript\* directory on the CD.
2. Issue the command:

```
db2 -f createCustomer.db2
```

Bind the two Component Broker bind files needed for the use of the Query Service. These files are named *db2cntcs.bnd* and *db2cntrr.bnd* and are the directory %SOMCBASE%*etc*. **This is a very important step. You must do this to perform queries.**

Issue the following set of commands within a DB2 Command Window:

```
db2 connect to CustDB
db2 bind %SOMCBASE%\etc\db2cntcs.bnd
db2 bind %SOMCBASE%\etc\db2cntrr.bnd
db2 connect reset
```

21.4 Client

21.4.1 Customer client with client-managed transactions

We provide you with a simple EJB client that controls a transaction and tests all of the finder methods on the CustomerHome interface. You can find the source for this client on the CD in *\Samples\EJB\Client* and included in the VisualAge for Java project that is provided in the repository file in *\Samples\EJB\Server\Source\FirstSteps EJB Sample.dat*.

To run this client, execute the following command from *\Samples\EJB\Client* directory on the CD:

```
java -classpath ".;..\Server\EJBCustomer.jar;..\Server\Working\NT\PRODUCTION\
CustomerClient.jar;%CLASSPATH%"
cb.ejb.samples.customer.CustomerApp <hostname> <port number>
```

This client takes the bootstrap hostname and port number as parameters, so remember to provide the correct values for your bootstrap host.

Clients to EJBs running in Component Broker need to have the EJB jar file and the EJB client jar file that is created when you build your object model. In this sample, this client needs the EJBCustomer.jar and CustomerClient.jar.

In order to execute a query in Component Broker, the query needs to be inside of a transaction. This means that when you make calls to finder methods on the home interface that makes use of the Query Service, those calls must made within the boundary of a transaction. As in this sample, the client starts a transaction before making these calls.

21.4.2 VisualAge for Java test client

You can also use the generated test client for the Customer EJB from inside of VisualAge for Java to test this sample bean.

Follow these steps to set up VisualAge for Java in order to generate and run the test clients successfully:

1. Load the WebSphere Test Environment into the Workspace.
2. Create a new project in your Workspace to contain the Component Broker Java ORB.
3. Import the file *<cb install>\lib\somojor.zip* into your new project. This file contains the ORB. Select "No" when dialogs pop up saying that there are package conflicts, and asking if you want to create a new edition of these packages. These conflicts occur because some of the standard CORBA packages already exist in the WebSphere Test Environment. If you select "Yes" to allow new editions of these packages to be made, make sure that you set the WebSphere Test Environment project back to its original version, otherwise you will have problems running the test environment.

The CB ORB needs to be in the Workspace, because when the EJB test clients initialize the JNDI initial context, it needs to specify the CB-specific initial context factory. If you do not have the CB ORB present in the Workspace, a ClassNotFound exception will occur when the test client tries to initialize the JNDI context.

4. Import the repository file for this sample *"\Samples\EJB\Server\Source\FirstSteps EJB Sample.dat"* into the Workspace, if you haven't done this already. You will find the EJB for this sample under the EJB group named FirstSteps_CustomerEJBGroup.
5. Import the EJB client bindings that were generated and compiled during deployment into the "FirstSteps EJB Sample" project. These client stubs should already be there, but this is a step you will have to do when you want to test EJBs running in CB that you develop yourself.
6. Generate the test clients for the FirstSteps_CustomerEJBGroup.

To run the test client, simply right-click on the Customer bean and select "Run Test Client".

In the "Connect" page of the test client, use the following values:

- **Provider URL**: If you are running the test client on a machine other than the Component Broker host machine, you will need to change the Provider URL to point to the host name and port number of your server, in the format "IIOP://hostname:port".
- **JNDI name**: cb/ejb/samples/customer/Customer
  (Or, use whatever you specified for this in the deployment descriptor.)
- **Initial context factory**: com.ibm.ejb.cb.runtime.CBCtxFactory
  (The default provided in the test client is the initial context factory used for the WebSphere test environment.)

You will not be able to test the finder methods on the CustomerHome interface with this test client, because these method calls must be inside of a transaction. The test client does not manage transactions for you when making calls on the home interface, so if you attempt to test these methods with this test client, they will fail.

21.5 What did you learn?

In this chapter, you learned how to deploy container-managed Enterprise JavaBeans in the Component Broker runtime.

Chapter 22. Mixing CORBA objects and Enterprise JavaBeans

This chapter demonstrates interoperability between CORBA objects and Enterprise JavaBeans in the Component Broker runtime environment.

You can find the Object Builder model, code, and client applications for this sample on the CD-ROM for this book in the directory \Samples\CORBA-EJB.

22.1 What you will learn

In this chapter, you will learn how to access EJBs from CORBA objects, as well as how to access CORBA objects from EJBs. You will also learn how to coordinate transactions which span both CORBA objects and EJBs in the same unit of work, and take advantage of two-phase commit support across both programming models.

You have already seen some of the interfaces to the Transaction Service in previous samples (see Chapter 13, “Persistent Object sample” on page 203, and Chapter 14, “Transactional Object sample” on page 217). In those samples, you learned how to set container policies so that the server will initiate transactions for you, as well as how to control transactions from a client application. Then you saw how to encapsulate a client-demarcated unit of work inside of a transient "application object" in the Account Manager scenario in Chapter 15, “Object with UUID Key and model dependency” on page 233.

This sample uses a similar Account Manager scenario to demonstrate the similarities and interoperability between CORBA and EJB components. We show you how to create a transient **AccountManager** CORBA object and an equivalent **AccountManager** session EJB, each of which have the same interface as the **AccountManager** in the previous sample. What differs in this sample is the implementation behind the interface. The **AccountManager** CORBA object uses interfaces to the CORBA Transaction Service to coordinate a transaction around transferring money from one account to another, and the **AccountManager** EJB uses interfaces to the Java Transaction Service (JTS) to coordinate its transactions. It both cases, the accounts involved in the transactions can be CORBA objects, EJBs, or a combination of the two.

22.2 CBConnector componentry

Figure 86 shows the main components used in our CORBA and EJB Account Manager scenario.

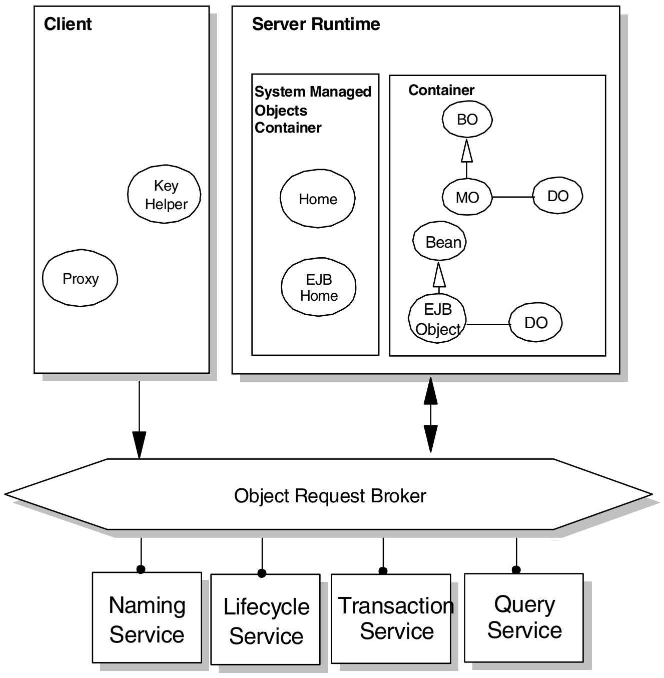


Figure 86. CORBA and EJB Components

22.2.1 Object Transaction Monitor (OTM)

Component Broker is an Object Transaction Monitor. The Transaction Service, like the rest of the services in Component Broker, conforms to OMG's CORBA specifications. We have already shown you how to use the Transaction Service; now, let's discuss a little bit about what this service provides.

One of the most important components of any distributed environment is the Transaction Service. To provide robustness in a distributed system, participants at different locations of the system should work together in a coordinated fashion. The Transaction Service provides this coordination.

22.2.1.1 What is a transaction?

What exactly is a transaction? A transaction sets the scope of a single, logical atomic action. When we say "atomic", we mean that the action is a complete set of events that either all take place, or none take place. For example, in our Account Manager scenario, the transfer of money from one account to another involves multiple events:

- Money should be removed from one account.
- The same amount of money should be added to a second account.

If for some reason these events cannot be processed successfully, either because of a physical problem (the database which is used to persist one of the accounts is down) or a logical problem (one of the accounts is temporarily frozen or has been closed), the system may be left in an inconsistent state. We don't want to have the money removed from one account without making sure that it reaches the other account. An application must be able to explicitly rollback a started transaction whenever it detects a problem in the completion of the transaction.

To ensure a consistent state at all times, a group of related operations can be executed within a single transaction. Before the transaction commits (ends successfully), none of the changes are visible to the other parts of the system. Just before the transaction ends, all of the participants in the transaction must agree on the success of the transaction, and only then do all of the changes take place. If, for some reason, one of the participants cannot process the operation correctly, the entire transaction is rolled back (cancelled), and none of the changes take place.

22.2.1.2 Two-phase commit

Operations in a distributed environment can take place in different processes running on different computers. The operations may span different types of resources, such as databases and file systems. This makes the successful completion of a transaction depend on the successful completion of each one of the resources in the distributed system.

For this reason, the transaction's commit (completion) has two phases:

Phase one The Transaction Service, after receiving a commit request from the application, sends a "prepare" message to all of the participating resources. Now each resource must vote on whether the transaction should be committed or rolled back. This vote will depend on whether the resource can complete its operation successfully.

Phase two The Transaction Service checks the votes to see if the transaction can be committed. Every resource has veto power. If all of the resources voted "commit", the Transaction Service will direct each resource to actually commit their operations. This brings the transaction to a successful end. If any of the resources voted" rollback", a rollback message will be sent to all of the participating resources, indicating that the operation should not take place.

By using a two-phase commit scheme, the Transaction Service can ensure that a transaction ends in a consistent state. It's an all or nothing deal, and each of the participants must agree on it.

22.2.2 Java Transaction API (JTA)

The Enterprise JavaBeans architecture requires a different set of interfaces, called the Java Transaction API (JTA) to be used for interacting with a transaction manager. In the EJB architecture, that transaction manager is commonly the Java Transaction Service (JTS).

Instead of working with a CORBA **Current** pseudo-object to control transactions, the JTA defines a similar interface with the **javax.transaction.UserTransaction**. Because Component Broker uses RMI/IIOP for communicating with remote EJBs, the **UserTransaction**, in the plumbing of the ORB, becomes a normal CORBA **Current** on the server.

22.2.3 Java Transaction Service (JTS)

The Java Transaction Service (JTS) is a Java binding of the CORBA Object Transaction Service (see http://java.sun.com/products/jts). Component Broker provides the JTS as a thin layer on top of its implementation of the CORBA Transaction Service.

What this means is, in the Component Broker runtime environment, whether you are demarcating transactions from a CORBA client (or application object) or an EJB client (or session bean), the underlying transaction monitor is the same. So, you can flow transactions across CORBA objects and EJBs, using either programming model for interfacing with the transaction services, and have full two-phase commit transactional coordination.

As it turns out, the JTS inside of WebSphere Advanced Edition has an identical implementation as that of Component Broker, which allows transactions to flow between components running in both environments. So, not only can you coordinate distributed transactions between CORBA components and EJBs inside of Component Broker, you can coordinate transactions between CORBA components (or EJBs) in Component Broker and EJBs inside of WebSphere Advanced Edition.

22.3 Server

This section describes the steps you need to perform to build, package, and install an application in which CORBA objects and Enterprise JavaBeans interoperate in the same server.

At this point in the book, we assume you have already worked through the previous samples and are familiar with the process of building components in Object Builder, as well as deploying Enterprise JavaBeans in Component Broker.

22.3.1 Build

This sample includes four components:

- **PersonalAccount** — a CORBA object persisted with DB2
- **CorporateAccount** — a container-managed entity EJB persisted with DB2
- **AccountManager** — a transient CORBA object (application object)
- **AccountManager** — a session EJB

We first walk you through creating the **PersonalAccount** and the **AccountManager** CORBA objects in Object Builder, then go through deploying the **CorporateAccount** and **AccountManager** EJBs.

Through the Object Builder walkthrough, we provide you with the correct parameter settings for the SmartGuides. If no suggestions are made by us for a SmartGuide page, use the default settings.

22.3.1.1 Create the model

1. Create a new Object Builder model.
2. From **User-Defined Business Objects**, add a new file, *Account*.
3. For this file, add a module, **account**, with the following parameters:

Parameter	Value
Name	account
Constructs: Exception	NotEnough AccountClosed (Members: accountNumber string <10>) TransferFailed (Members: message string <100>)

Create the PersonalAccount CORBA object

4. For this module, add an interface, **PersonalAccount**. Select "The interface is queryable."
5. Add attributes to the **PersonalAccount** Interface as described in the following table:

Parameter	Type
accountNumber	string <10> Read-only
accountholder	string<30>
address	string<100>
phoneNumber	string<12>
accountState	string<20> Read-only

Add methods with the following definitions:

Method	Return Type	Parameter	Exception
getBalance	double		
changeBalance	double	anAmount (double)	account::NotEnough account::AccountClosed
open	void		
close	void		

6. Add a Key with the accountNumber attribute.
7. Add a Copy Helper with all attributes.
8. Add an implementation of the BO. Use the following parameters:

Parameter	Value
Implementation Language	Java
Attributes	balance (double, Private)
Methods	isOpen (Return Type: boolean, Private) isClosed (Return Type: boolean, Private)
State data	all attributes

Note

In the following steps, as an alternative to typing in the method implementations in the editor pane, you can link to an external file. To do this, open the properties of the method by right-clicking on the method and select "properties". Then select "Use an external file," and browse to select the file which has the method implementation. The code for this sample is located on the CD-ROM for this book in the directory \Samples\CORBA-EJB\Server\Source. The files that contain the method implementations have the extension ".TDE".

9. Implement the **changeBalance** method:

```
if (isClosed()) throw new account.AccountClosed(iAccountNumber);
if ((iBalance + anAmount) < 0 ) throw new account.NotEnough();
iBalance += anAmount;
```

10. Implement the **getBalance** method:

```
return iBalance;
```

11. Implement the **open** method:

```
iAccountState = "OPEN";
```

12. Implement the **close** method:

```
iAccountState = "CLOSED";
```

13. Implement the **isOpen** method:

```
if (iAccountState.equalsIgnoreCase("OPEN")) return true;
else return false;
```

14. Implement the **isClosed** method:

```
if (iAccountState.equalsIgnoreCase("OPEN")) return true;
else return false;
```

15. For the PersonalAccountDO, add a Data Object implementation with the following parameters:

Parameter	Value
Environment	BOIM with any Key
Form of Persistent Behaivor and Implementation	DB2 Cache Service
Data Access Pattern	Delegating

16. For the PersonalAccountDOImpl, add a Pesistent Object and Schema with the following parameters:

Parameter	Value
Group Name	DB2Group
Database	ACCNTDB

17. Add a Managed Object to the **PersonalAccountBO**.

Create the AccountManager CORBA object

1. For the **account** module, add another interface, **AccountManager**.
2. Add the method with the following definition to the **AccountManager** Interface:

Method	Return Type	Parameters	Exception
transferAmount	void	anAmount (double) sourceAccountNumber (string<10>) destAccountNumber (string<10>)	account::TransferFailed

3. Add an implementation of the BO. Use the following parameters:

Parameter	Value
Implementation Language	Java

4. Implement the **transferAmount** method by linking to an external file. To do this, open the properties of the method by right-clicking on the method and select "properties". Then select "Use an external file," and browse to select the file which has the method implementation. This code is located on the the CD-ROM for this book in the directory \Samples\CORBA-EJB\Server\Source\doTransferAmount.tde.

Note

The transaction is being controlled through the interfaces from the CORBA programming model; and by having this component be a client to the PersonalAccount CORBA Object as well as the CorporateAccount EJB, we are using both the CORBA and EJB programming models in the same method.

5. For the AccountManagerDO, add a Data Object implementation with the following parameters:

Parameter	Value
Environment	BOIM with UUID key

6. Add a Managed Object to the **AccountManagerBO**.
7. Save your object model and exit Object Builder. Object Builder should be closed before going through the process of deploying the EJBs.

Deploy the CorporateAccount container-managed entity EJB

The EJBs used in this sample have already been created for you. The EJB jar file containing these beans is located on the the CD-ROM for this book in \Samples\CORBA-EJB\Server\EJBAccount.jar. There are also some .bat files in this same directory which will simplify some of the following steps for you.

If you'd like to look at the source for these beans, which we're sure you will, you can find it in \Samples\CORBA-EJB\Server\Source. The beans will be located in the package structure "cb\ejb\samples". Also, we have provided the VisualAge for Java repository file named "First Steps CORBA-EJB Sample.dat" so you can see the beans in the EJB development environment and use the generated test clients to test the beans once they are deployed in Component Broker.

1. Copy the EJBAccount.jar into the project directory you specified for your Object Builder object model.
2. Execute the following **cbejb** command from a command line from the same directory in which you copied the jar file in the previous step. This will generate the object model for the CorporateAccount bean and add it to the Application Family named CORBA_EJB_AccountAppFam. The "-cacheddb2" flag also specifies that you want this bean persisted using the DB2 Cache Service. (You can also copy the "buildCorporateAccount.bat" file from the CD and execute it in your project directory instead).

```
cbejb EJBAccount.jar -bean cb.ejb.samples.CorporateAccount -cacheddb2 -family
CORBA_EJB_AccountAppFam
```

3. Because this bean is container-managed, Object Builder will open for you to map the container-managed attributes to a backend datastore (in this sample it is DB2). When Object Builder comes up, open your project.
4. Add a Persistent Object and Schema to the DOImpl which has already been created for you by the **cbejb** tool. The DOImpl will have the name **cb_ejb_samples_CorporateAccountDOImpl**. Use the following parameters:

Parameter	**Value**
Group Name	DB2Group
Database	ACCNTDB
Table	CorporateAccount

5. The mappings from the Data Object attributes to the Persistent Object attributes should already be done for you. However, you'll need to provide a length for the *accountNumber* column that the Persistent Object's *accountNumber* attribute will be mapped to. The *accountNumber* column must have a specified size because it is the primary key of the table you are defining. The other attributes have default values set for you, but you might want to update these to match the definition that was defined for the PersonalAccount table in the previous section when you created the

PersonalAccount object. Use the following the values for the schema definition:

Schema Column	SQL Type	Length
accountNumber	VARCHAR	10
accountholder	VARCHAR	30
balance	DOUBLE	
businessPhoneNum	VARCHAR	12
companyName	VARCHAR	50
accountState	VARCHAR	20

6. Save your object model, then exit Object Builder. This will continue the **cbejb** processing.

Deploy the AccountManager session EJB

Execute the following **cbejb** command from a command line from your project directory. This will generate the object model for the AccountManager bean and add it to the Application Family named CORBA_EJB_AccountAppFam. (You can also copy the "buildAccountManager.bat" file from the CD and execute it your project directory instead).

```
cbejb EJBAccount.jar -bean cb.ejb.samples.AccountManager -family
CORBA_EJB_AccountAppFam
```

Once the **cbejb** processing finishes, you are ready to to build the server application.

22.3.1.2 Build the server application

Open to your project in Object Builder. In the **Build Configuration** folder, you find client and server DLLs defined for each of the EJBs you've deployed. You need to define the DLLs for the CORBA objects you created in the model.

1. Add a client DLL named AccountClient with the following parameters:

Parameter	Value
Name	AccountClient
Client Source Files	Account AccountKey AccountCopy
Libraries to Link With	none

2. Add a server DLL named AccountServer with the following parameters:

Parameter	Value
Name	AccountServer
Server Source Files	AccountMO AccountBO AccountDO AccountDOImpl
Libraries to Link With	AccountClient

3. Generate the makefile for "All Targets".
4. Execute the following command from the command line in the "Working\NT" directory under your project directory. **Note**: You need the EJB jar file in the classpath, because the CORBA AccountManager uses the CorporateAccount remote and home interfaces in its implementation.

```
set classpath="..\..\etc\EJBAccount.jar;%CLASSPATH%";
nmake -f all.mak
```

22.3.2 Package

In the **Application Configuration** folder, you will already have the application family named CORBA_EJB_AccountAppFam which was created when you ran the **cbejb** tool.

1. For this application family, add an application named AccountApp, with parameters as follows: (The jar files listed under Additional Executables are necessary, because the AccountManager CORBA object is a client to the CorporateAccount EJB, therefore it needs the EJB interfaces and client proxies available as any other client does. You can use the full path

to the files you are adding, but here we use relative paths so the model will be more portable):

Parameter	Value
Application Name	AccountApp
Additional Executables	..\etc\EJBAccount.jar ..\Working\NT\PRODUCTION\CorporateAccountClient.jar

2. Add the **PersonalAccount** Managed Object to the application with the following parameters:

Parameter	Value
Managed Object	accountMO::PersonalAccountMO
Primary Key	accountKey::PersonalAccountKey
Copy Helper	accountCopy::PersonalAccountCopy
Data Object Implementation	accountDOImpl::PersonalAccountDOImpl
Container	PersonalAccountContainer (create a new container, accept all defaults)

3. Add the **AccountManager** Managed Object to the application with the following parameters:

Parameter	Value
Managed Object	accountMO::AccountManagerMO
Data Object Implementation	accountDOImpl::AccountManagerDOImpl
Container	CachedTransientObjects

4. Generate the CORBA_EJB_AccountAppFam.

22.3.3 Install

To install and configure the server application, follow these steps:

1. Install the server application on your server machine.

2. Create a **CORBA EJB Server** in a new Management Zone, **CORBA EJB Zone**. Use a new configuration, **CORBA EJB Configuration**, for this server.
3. Add the following applications to the configuration, and configure them into the **CORBA EJB Server** server:
   AccountApp
   Specific AccountApp
   AccountManagerApp
   CorporateAccountApp
   Specific CorporateAccountApp
   iDB2IMServices
4. Activate the configuration.
5. When the activation is complete, bind the **CorporateAccount** and **AccountManager** EJBs to the JNDI name service by executing the following commands (**Note**: You will need to change the ORBInitialHost and ORBInitialPort values to reflect your host name and port number.):

```
ejbbind EJBAccount.jar cb.ejb.samples.CorporateAccount
-ORBInitialHost cbdemo.sl.dfw.ibm.com -ORBInitialPort 900

ejbbind EJBAccount.jar cb.ejb.samples.AccountManager
-ORBInitialHost cbdemo.sl.dfw.ibm.com -ORBInitialPort 900
```

22.3.4 Create the database

Before running the clients against your components, you need to create the CorporateAccount and PersonalAccount tables in DB2. In the steps below, we install these tables in the existing AccntDB database you used in previous samples. Also, make sure you bind the Component Broker-provided bind files to this database. These bind files are needed by the DB2 Cache Service.

To create the database tables, follow these steps:

1. Open a DB2 Command Window and change directory to the working directory for this sample's object model.
2. Issue the following commands:
   db2 connect to AccntDB
   db2 -tf CorporateAccount.sql
   db2 -tf PersonalAccount.sql
   db2 connect reset

Bind the two Component Broker bind files needed for the use of the DB2 Cache Service. These files are named *db2cntcs.bnd* and *db2cntrr.bnd* and are in the directory *%SOMCBASE%\etc*. **This is a very important step**.

Issue the following set of commands within a DB2 Command Window:

```
db2 connect to AccntDB
db2 bind %SOMCBASE%\etc\db2cntcs.bnd
db2 bind %SOMCBASE%\etc\db2cntrr.bnd
db2 connect reset
```

22.4 Client

We didn’t write any clients for this sample, because there is nothing unique about client programming related to the components built in this scenario. You have already seen the client programming that of interest in this sample in the implementations of the **AccountManager** application objects.

For testing your components for this sample, use the test clients that the tools generate for you.

22.4.1 CORBA clients

Use the generated QuickTest to test the **PersonalAccount** and **AccountManager** CORBA objects.

In Component Broker 3.0, QuickTest does not generate test clients for EJBs. Because we placed both CORBA objects and EJBs in the same model, the make file that gets generated for compiling QuickTest clients is incorrect. If you’d like to compile these clients yourself, we’ve modified the make file to only build test clients for the CORBA objects, and this make file is located on the CD in \Samples\CORBA-EJB\Server\Working\NT\qt.mak.

To run the QuickTest clients for this sample, execute the following command from the CD:

```
\Samples\CORBA-EJB\Server\Working\NT\PRODUCTION\qt.bat
```

22.4.2 EJB clients

Use the generated test client for the CorporateAccount and AccountManager EJBs from inside of VisualAge for Java to test these beans.

The following steps are described in 21.4.2, “VisualAge for Java test client” on page 352, but are listed here again for clarity.

Follow these steps to set up VisualAge for Java in order to generate and run the test clients successfully:

1. Load the WebSphere Test Environment into the Workspace.
2. Create a new project in your Workspace to contain the Component Broker Java ORB.
3. Import the file *<cb install>\lib\somojor.zip* into your new project. This file contains the ORB. Select "No" when dialogs pop up saying there are package conflicts, and asking if you want to create a new edition of these packages. These conflicts occur because some of the standard CORBA packages already exist in the WebSphere Test Environment. If you select "Yes" to allow new editions of these packages to be made, make sure that you set the WebSphere Test Environment project back to its original version, otherwise you will have problems running the test environment. The CB ORB needs to be in the Workspace, because when the EJB test clients initialize the JNDI initial context, it needs to specify the CB-specific initial context factory. If you do not have the CB ORB present in the Workspace, a ClassNotFound exception will occur when the test client tries to initialize the JNDI context.
4. Import the repository file for this sample *"\Samples\CORBA-EJB\Server\Source\First Steps CORBA-EJB Sample.dat"* into the Workspace, if you haven't done this already. You will find the EJBs for this sample under the EJB group named FirstSteps_AccountEJBGroup.
5. Import the EJB client bindings that were generated and compiled during deployment into the "FirstSteps CORBA-EJB Sample" project. These client stubs should already be there, but this is a step you will have to do when you want to test EJBs running in CB that you develop yourself.
6. Generate the test clients for the FirstSteps_AccountEJBGroup.

To run the test clients, simply right-click on the bean you want to test and select "Run Test Client".

In the "Connect" page of the test client, use the following values:

- **Provider URL**: If you are running the test client on a machine other than the Component Broker host machine, you will need to change the Provider URL to point to the host name and port number of your server, in the format "IIOP://hostname:port".
- **JNDI name**: cb/ejb/samples/AccountManager
- **Initial context factory**: com.ibm.ejb.cb.runtime.CBCtxFactory (The default provided in the test client is the initial context factory used for the WebSphere test environment.)
- **Home interface**: cb.ejb.samples.AccountManagerHome

22.5 What did you learn?

In this chapter, you learned how to implement a server application which has CORBA objects and Enterprise JavaBeans interoperating with each other, as well as coordinating transactions across these components using both programming models.

Chapter 23. IOM with two datastores

This chapter builds on all the previous samples in this book.

You can find the Object Builder model, code, and client applications for this sample on the CD for this book in the directory *\CBConnector\20-IOM-Two-Datastores*.

23.1 What you will learn

In the previous samples, you have learned how to implement a Java BO, use databases to make object attributes persistent, and how to perform queries for objects and their attributes. You have also learned how to incorporate definitions from other Object Builder object models into a new model. In this chapter, you will create a server application that involves everything that you have learned so far.

You will create a transient **CustomerManager** object that uses a UUID Key and has a Java BO implementation. This **CustomerManager** acts as a client to both objects you created in Chapter 19, “Query Service — reuse of an existing table” on page 291, and performs queries over all the **Customer** and **CustomerAccount** objects. This sample further illustrates the functionality provided by CBConnector by having a Java BO talking to two C++ BOs, each being backed by two different DB2 databases.

23.2 CBConnector componentry

In this section, we discuss the Interlanguage Object Model (IOM) to prepare for the sample in this chapter.

23.2.1 Interlanguage object model

In Chapter 11, “Transient Object sample” on page 161, we introduced the differences between the components involved in having a Java BO and a C++ BO. We briefly explained that when you have a Java BO, there will always be some cross-language communication going on, because all other parts of the total Managed Object assembly (the DO, Mixin, MO) are implemented in C++. Here, we will give more explanation of how this communication is handled between Java and C++.

In a distributed object system, cross-language communication is commonly done with CORBA using ORB-to-ORB communication, as is the case when Java (or ActiveX) client objects communicate with any Managed Object. This path *has* to be used if the two objects reside on different hosts or on the same host but in separate server processes, as shown in Figure 87.

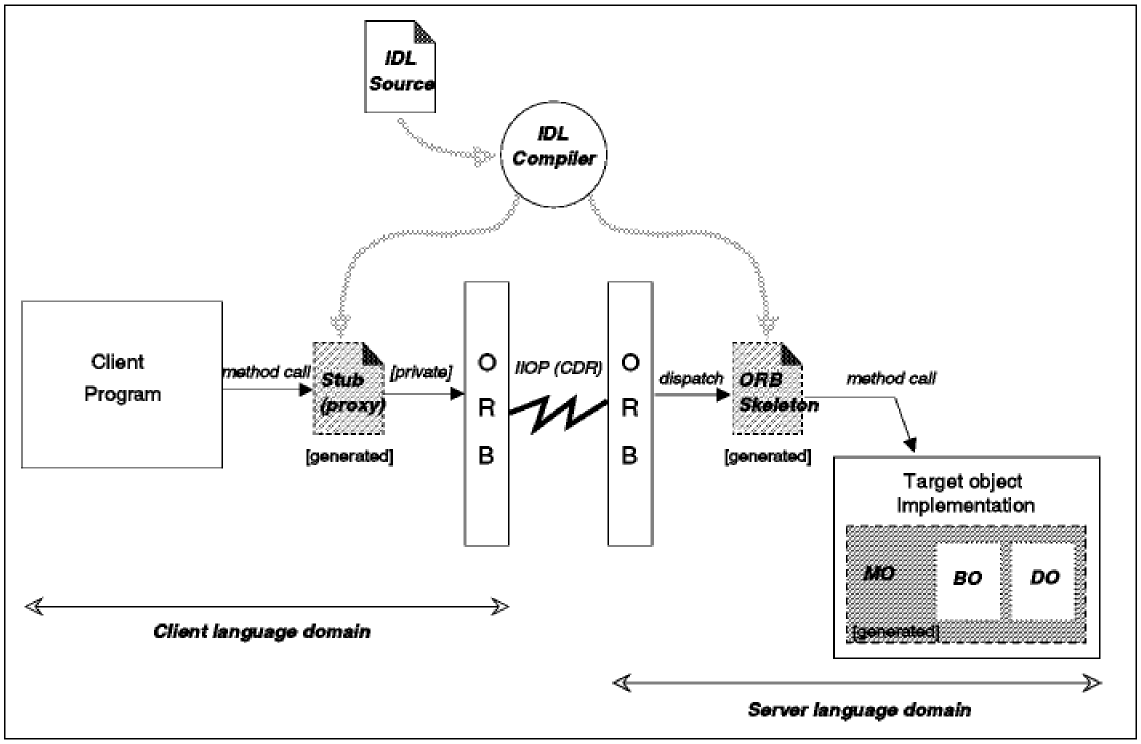


Figure 87. CORBA method invocation model

This is just fine when there is a "wire" between a client and a server; it does imply some overhead. What if the object interactions between a Java and a C++ object all occur within just one process? This is where the Interlanguage Object Model (IOM) capability comes in.

IOM implementations use IDL for object interfaces, but these interfaces are implemented in different languages, currently in C++ or Java, yet residing in the same process. Interlanguage communication occurs when an implementation in one language invokes a method on an implementation in a different language. The client object in Figure 88 represents any object that invokes a method on any other *local* object written in a different language. The conversion step between the languages is made in a similar way as the conversion in the ORB, using Common Data Representation. However, in this case there is no need for IIOP, since the interaction takes place within a single process.

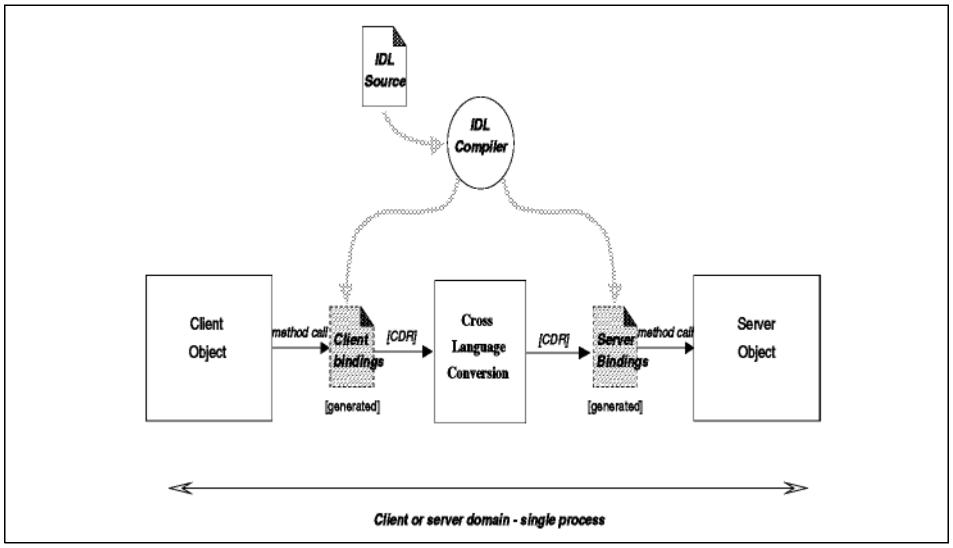


Figure 88. Local cross-language method invocation model

You can find a more in-depth explanation of language interoperability in the redbook *IBM Component Broker Connector Overview*, SG24-2022.

23.2.2 Query Service

In Chapter 19, “Query Service — reuse of an existing table” on page 291, we introduced various ways of using the Query Service. One of which was the use of the **evaluate_to_iterator** method, which you call on the query evaluator of an application server. In that sample, the query we performed with this method was logically the same as calling **evaluate** on the home. If you remember back to our examples of these queries, if you wanted to find all Customers that had the first name of 'Charles', using **evaluate** you would pass the predicate of the query statement:

```
firstName = ’Charles’ ;
```

Apart from the way you write the code, in each case, you get back an iterator over the same objects. So you might be wondering why you would choose one query method over the other.

One advantage to using **evaluate_to_iterator** is that you can query over multiple homes. For example, if you would like to find all Accounts for the Customers who have a first name of 'Charles' and last name of 'Babbage', your query statement would look like this:

```
select e from CustomerAccountMOHome e, CustomerMOHome d
   where d.firstName='Charles' and d.lastName='Babbage
   and d.customerNumber = e.accountHolder;
```

In the scenario for this chapter, we want to compute the total balance of all the **CustomerAccounts** that a **Customer** holds. We do this by first querying for the Customers with a specific first and last name. If more than one Customer is found for the given name, we take the first one returned by the query. We then query for all **CustomerAccounts** based on the customerNumber of that Customer. For each query, we use the **evaluate_to_data_array** method for performance reasons. Once you have gone through this chapter, you might want to substitute our implementation of the queries with this alternate implementation, using the **evaluate_to_iterator** method with the query as described above. Both implemenations are provided for you on the CD.

23.3 Server

This section takes you through all the steps you need to perform to build, package, and install a server application that uses a UUID Key for its Primary Key.

Note

For your convenience, all code snippets used in this sample are on the CD-ROM located in *\CBConnector\20-IOM-Two-Datastores\Server\code_snippets.java*.

23.3.1 Build

Remember, through the Object Builder walkthrough, we provide you with the correct parameter settings for the SmartGuides. If no suggestions are made by us for a SmartGuide page, use the default settings.

We strongly recommend that you open Help for each field of the SmartGuide pages in order to understand more of the dialectic behind the input fields.

23.3.1.1 Create the model

1. Create a new Object Builder model in a separate directory. Depend on the model you created for the sample in Chapter 20, “Meet-in-the-middle paradigm with two datastores” on page 323, or specify the model provided for that sample on the CD-ROM in the directory *\CBConnector\19-Two-Datastores\Server*.

   You will see under **User-Defined Business Objects** folder, the *Customer* and *CustomerAccount* files and interfaces defined in the "depended on" model. These definitions are visible as read-only.

2. From **User-Defined Business Objects**, add a new file, *CustomerManager*, using the following parameters:

Parameter	Value
Name	CustomerManager
Constructs: Exception	ComputingBalanceFailed Members: message string <50>
Fires to Include	IManagedClient (default) Customer CustomerAccount

3. For this file, add an interface, **CustomerManager**, with the following parameters:

Parameter	Value
Name	CustomerManager
Constructs: Exception	ComputingBalanceFailed Members: message string <50>
Interface Inheritance	IManagedClient IManagedClient::IM anageable

Add a method, **getTotalBalance** with the following definition:

Method	Return Type	Parameters	Exception
getTotal Balance	double	firstName string<30> In lastName string<30> In	Computing BalanceFalied

4. Add an implementation of the BO. Use the following parameters:

Parameter	Value
Pattern for handling state data	Caching
Object Reference	Use lazy evaluation
Data Object Interface	Create a new one now
Implementation Language	Java

5. Implement the **getTotalBalance** method, as shown in Figure 89 and Figure 90. The code for this method is on the CD-ROM in the file \CBConnector\20-IOM-Two-Datastores\Server\code_snippets.java. The following snippet is only an outline of the logic implemented by this method. In the *code_snippets.java* file, we provide two implementations of this method. Only one is shown here, and it does not show the necessary variable definitions or error-checking and handling:

```
//*** resolve the orb ***
orb = CBSeriesGlobal.orb();

//*** resolve the Name Service ***
nameService = CBSeriesGlobal.nameService();

//*** get the current transaction ***
orbCurrent = orb.get_current("CosTransactions::current");
currentTransaction = org.omg.CosTransactions.CurrentHelper.narrow(orbCurrent);
currentTransaction.set_timeout(1000);

//*** Begin the transaction ***
currentTransaction.begin();

String customerNumber = null;

//*** Get Query Evaluator ***
String queryEvalutorString = "host/resources/servers/CustomerAccountTDServer/query-evaluators
                              /default";
obj = nameService.resolve_with_string( queryEvalutorString );
queryEvaluator = com.ibm.IExtendedQuery.QueryEvaluatorHelper.narrow(obj);

//*** Query for Customer numbers that match the firstName and lastName ***
String query = "select e.customerNumber from CustomerMOHome
                e " + "where e.firstName = \'" + firstName + "\'
                and e.lastName = \'" +
                lastName + "\';";

queryEvaluator.evaluate_to_data_array(
                                  query,
                                  null,
                                  nullPair,
                                  nullPair,
                                  0,
                                  dataArrayListHolder,
                                  dataArrayIteratorHolder);

while (dataArrayIterator.next_one(tupHolder))
{
  tup = tupHolder.value;
  customerNumber = tup[0].extract_string();

  // first customer was found, so break
  break;
}
```

Figure 89. getTotalBalance() — Part 1

```
   //*** commit the transaction ***
   currentTransaction.commit(true);

   //*** get the current transaction ***
   orbCurrent = orb.get_current("CosTransactions::current");
   currentTransaction = org.omg.CosTransactions.CurrentHelper.narrow(orbCurrent);
   currentTransaction.set_timeout(1000);

   //*** Begin the transaction ***
   currentTransaction.begin();

   double total = 0;

   //*** Get Query Evaluator ***
   String queryEvalutorString = "host/resources/servers/CustomerAccountTDServer/query-evaluators
                              /default";
   obj = nameService.resolve_with_string( queryEvalutorString );
   queryEvaluator = com.ibm.IExtendedQuery.QueryEvaluatorHelper.narrow(obj);
 //*** Query for CustomerAccounts for the customerNumber ***
   String query = "select e.balance from CustomerAccountMOHome e " +
                  "where e.accountholder = \'" +
                  customerNumber + "\';";
   queryEvaluator.evaluate_to_data_array( query, null, nullPair, nullPair, 0, dataArrayListHolder,

                                                       dataArrayIteratorHolder);
 double balance = 0;
 // process the first account balance found
   tup = tupHolder.value;
   balance = tup[0].extract_double();
   total += balance;
 // process the rest of the balances
   while (dataArrayIterator.next_one(tupHolder))
   {
     tup = tupHolder.value;
     balance = tup[0].extract_double();
     total += balance;
   }
 //*** commit the transaction ***
   currentTransaction.commit(true);
 return total;
```

Figure 90. getTotalBalance() — Part 2

6. Under the **File Adornments** folder in the Methods panel, select **CustomerManagerBOPrologue**. This is where you can include any import statements your methods may need to resolve exterior class definitions. Include the following import statement:

```
import com.ibm.CBUtil.*;
import org.omg.CORBA.ORB;
import com.ibm.IExtenderLifeCycle.*;
import com.ibm.IManagedClient.*;
import com.ibm.IManagedAdvancedClient.*;
import org.omg.CosTransactions.Current;

import CustomerManagerPackage.*;
```

Note

The package named CustomerManagerPackage is generated when you build the DLLs for your server application in the following section.
As part of the build process, Java files are generated from the IDL files from the model. In this sample, *CustomerManager.idl* contains the exception `ComputingBalanceFailed`. When the Java files are generated, the `ComputingBalanceFailed` exception classes are placed into the `CustomerPackage`, and because of a bug in how Object Builder generates the files for your Java BO, you must import this package to resolve the reference to this exception.

7. For the CustomerManagerDO, add a Data Object implementation with the following parameters:

Parameter	**Value**
Environment	BOIM with any Key
Form of Persistent Behavior and Implementation	Transient
Handle for Storing Pointers	Home name and Key

8. Add a Managed Object to the **CustomerManagerBO**. Accept all default settings.
9. Generate the code for all files from **User-Defined Business Objects**.
10. Due to a bug in the code generation of the Java BO implementation, you must edit the generated Java file *_CustomerManagerBOBase.java* located in the working directory. Open this file in an editor, and replace all references to `ComputingBalanceFailed` with `CustomerManagerPackage.ComputingBalanceFailed`.

23.3.1.2 Build the server application

1. Add a client DLL named CustomerManagerTDC with the following parameters:

Parameter	Value
Name	CustomerManagerTDC
Client Source Files	CustomerManager
Libraries to Link With	none

2. Add a server DLL named CustomerManagerTDS with the following parameters:

Parameter	Value
Name	CustomerManagerTDS
Server Source Files	All CustomerManager files
LIbraries to Link With	CustomerManagerTDC (the client library) CustomerTDC (depended on library) CustomerAccountTDC (depended on library)

3. Generate the makefiles for all targets.
4. Copy the following files from the working directory of the "depended on" sample into your current working directory.

   Because the **CustomerManager** acts as a client to the **Customer** and **CustomerAccount** objects, you need the normal client files associated with them:

   - Customer.hh
   - CustomerAccount.hh
   - CustomerTDC.lib
   - CustomerAccountTDC.lib
5. Build all Java out-of-date targets.
6. Build all C++ out-of-data targets.

23.3.2 Package

In this section we package the application using Object Builder.

23.3.2.1 Define the application

1. Add an application family named **CustomerManagerTDAppFam**.
2. For this application family, add a server application named CustomerManagerTDAppS with parameters as follows:

Parameter	Value
Application Name	CustomerManagerTDApps
Initial State of Application	Stopped
Additional Executables	CustomerTDC.dll CustomerAccountTDC.dll CustomerManagerTDC.dll

   You don't normally need to include the client DLL for the server application because it is included by default to the install ation script. However, in CBConnector Release 1.2, you need to specifically include the client DLL when you have a Managed Object that uses a UUID Key.

3. Add a Managed Object for the server application with the following parameters:

Parameter	Value
Data Object Implementation	CustomerManagerDOImpl
Container	CachedTransientObjects
Managed Object	CustomerManagerMO

23.3.2.2 Create an installation image

1. From the CustomerManagerTDAppFam, generate the installation scripts.
2. Build the installation image.

23.3.3 Install

To install and configure the server application, follow these steps:

1. If you have not already done so, install the single Location Objects as described in 4.1, “Get ready” on page 30, as a recommended procedure. This will install the location, **CustomerManagerTDScope**, which specifies the **CustomerManagerTDServer** in its configuration.
2. Install the server application on your server machine.
3. Configure an **CustomerManagerTDServer** in a new Management Zone, **CustomerManagerTD Management Zone**. Use a new configuration, **CustomerManagerTDConfiguration**, for this server. For this sample, you must use exactly this server name for the application server to enable the execution of the client programs because the single Location Object you installed refers to it.

   Alternately, you could configure the application on the **CustomerAccountTDServer** used for the Customer and CustomerAccount applications. This would be a more logical configuration since the **CustomerManager** uses these applications.
4. Activate the configuration.

23.4 Client

These sections describe what is different in the client applications, as related to the general descriptions given in Chapter 10, “Sample client development approach” on page 117.

23.4.1 The C++ client

We remember that the **custproxy** class is the application-dependent class, a subclass of the **cbcproxy** class, which hides the CBConnector specifics from the client application.

23.4.1.1 The custproxy

In this sample, we re-use the **acctproxy** class from Chapter 11, “Transient Object sample” on page 161. We rename it to **custproxy** and use the *CustomerManager* header file instead of the *AccountManager* header file. In addition, we do the usual hardcoding of scope and interface information in the **initCBC** method.

23.4.1.2 The GUI

The GUI lets you add the total balance for a customer. The fields display the first and last name of the customer and the resulting balance when the balances for all this customers' accounts have been added.

The Business Object in this client is the **CustomerManager** object. It only has one method, **getTotalBalance**, that takes the first and last name of the customer as input and returns the amount. We wrap this Business Object with the **CustomerMgrWrapper**.

Finally, we need a **CustMgrCBCProxyWrapper** to provide us with CBConnector services and the CustomerManager Business Object. Run the client by typing the following command in the VisualC++ directory:

```
IomTDC.exe
```

23.4.2 The Java client

All Java source and class files for this sample are located on the CD in the directory *\CBConnector\20-IOM-Two-Datastores\Client\Java*.

23.4.2.1 Generate client proxies

For this sample, you need to generate the proxies for the **CustomerManager** interface as described in 6.7.2, “Java client programming model” on page 76. The **CustomerManager** uses the **Customer** and **CustomerAccount** interfaces; so their IDL files must be in the same directory as *CustomerManager.idl* when you generate the proxies.

23.4.2.2 The CustomerManagerProxy class

The **CustomerManagerProxy** class is similar to the **AccountManagerProxy** class used in 15.4.2, “The Java client” on page 244.

23.4.2.3 The GUI

The GUI lets you compute the total balance for a **Customer**. The fields display the first and last name of the **Customer** and the resulting total balance.

23.4.2.4 Run the client

You can run this client as an application and as an applet, just as you have in the previous samples.

To run the client application, execute the command:

```
java CustomerManagerApp <hostname> <port>
CustomerManagerTDServer-server-scope
```

To run the client as an applet, embed the applet in an HTML page (*IOMTwoDatastores.html*), and open that page in the applet viewer with the command

```
appletviewer IOMDatastores.html
```

If you use the sample command (CMD) files provided on the CD, remember to change the `hostname` and `port` parameters (in *runApplet.cmd* and *IOMTwoDatastores.html*) to reflect your bootstrap server.

23.5 What did you learn?

In this chapter, you learned how to implement a Java BO that uses the Query Service to access other Business Objects implemented in C++. By creating this sample, you have seen an example of the CBConnector IOM implementation in action.

Chapter 24. Security service

This chapter describes the use of the security service with Component Broker. We explain why we should make our network secure, the OMG Security Reference Model (SRM), the Component Broker's implementation of the security service, all the steps needed to set up the security properties using the System Management User Interface, and the security differencies between the C++ and Java clients.

24.1 Why we need to make our network secure

The Internet brought to us a new way to do business. E-commerce, Web banking, trade on-line, and other business services on the Net are few examples. Credit card numbers, personal information, and all confidential data should flow securely across the network. The Internet is a TCP-IP protocol based network, and due to the behavior of the TCP-IP protocol, it is totally unsecure. There is no authentification of the server and the client, and no integrity and confidentiality of the data.

With distributed applications, the users can access our application remotely, and without any security services, everybody can use our business objects, everybody can see, reuse or modify the information. To prevent such things, we need to make sure that the user is who he claims to be, the server which we are connected to is the authorized server, and the information between client and server are exchanged securely without any tampering.

24.2 OMG Security Reference Model (SRM)

OMG has defined a general Security Reference Model (SRM) to provide various security policies to make the distributed environment more secure. This architecture covers the following protection features:

- Access control
- Identification and authentification
- Transport message protection
- Delegation mechanisms
- Non-repudiation
- A security audit log mechanism

Access control provides a mechanism to limit the access to an interface, a subset of the implementation of an interface, or an interface operation. Even if the user has been authorized, we can have an Access Control List (ACL) which contains all the privileges the user has and all the interfaces or operations of it that the user is allowed to use. By providing this finer granularity security, we make the distributed environment more secure and flexible.

Identification and authentification allow the user to the CORBA system, the client to the server, and the server to the client to be authenticated. Using authentification protocols, the server ensures that the client is who it claims to be and that the server is the real server.

Transport message protection makes the data flowing across the network unreadable. Using cryptographic techniques, the users can exchange the information securely even if the trasport protocol used by the network is not secure.

Delegation provides a mechanism to how the credentials of a principal are passed between the objects.

SRM identifies five types of delegation policies:

- No delegation
- Impersonation
- Combined delegation
- Composite delegation
- Traced delegation

Non-repudiation means that once we have done an action, we cannot repudiate it.

A security audit log mechanism which means that the system is able to log all the activities and the accesses of the interfaces and methods.

24.3 Component Broker's implementation of the security service

To make the network more secure, Component Broker defines the following security mechanisms:

Authentication, which allows the servers and the clients to be authenticated to the CORBA system. Component Broker uses the DCE security server as a 3-party authentication scheme. DCE manages a user registry to store all the information of the users, and using the Kerberos/V protocol, it verifies the users with knowledge-based authentification. Component Broker supports the certificate based authentification using SSL protocol. In this version of CB, only a Java client can use SSL as the authentification mechanism.

Authorization, which controls the access to the business objects in application servers. When the application server receives a method request from a principal, it checks an ACL to see if the permission to invoke this method has been granted to the principal. By default, CB uses a coarse level of authentification, which means that all the authenticated users can access all the objects in the system. To use a fine level of authorization, we register the objects we would like to protect to a repository and set all the permissions of the principals for each protected object. In order to use a fine level of authorization, we need the IBM SecureWay Policy Director product, which is used by CB. Specifically, the CB uses the following Policy Director components for the fine level of authentification:

- An authorization server, which protects objects registered to the authorization repository by enforcing the authorization policies defined in the ACL.
- A management console, which is a GUI to set an ACL to objects.
- A NetSEAT client, which talks to an authorization server. The NetSEAT client is built into the CB server runtime.
- An authorization database manager, which maintains a secure authorization policy directory used by authorization servers.

Delegation, which is a mechanism to transfer the credentials between objects. Component Broker supports no delegation and simple delegation or impersonation.

Credential Mapping, which allows you to define a credential mapping between the CB security credential and those of an entity in the data system’s security mechanism. Credential mapping supports two different types of mapping. The simple type uses the information in the system management to map the credential. It is not flexible because you can set up only one userid and password for a server or a server group, and it is not secure because all the userids and passwords are stored in clear. SSO-enable allows CB to store the credential mapping in a secure database. This version of Component Broker uses GSO database as secure storage. Thus, you need to install the IBM Global Sign-On product in order to use this security feature.

24.4 Reuse the Persistent Object account sample, making it secure

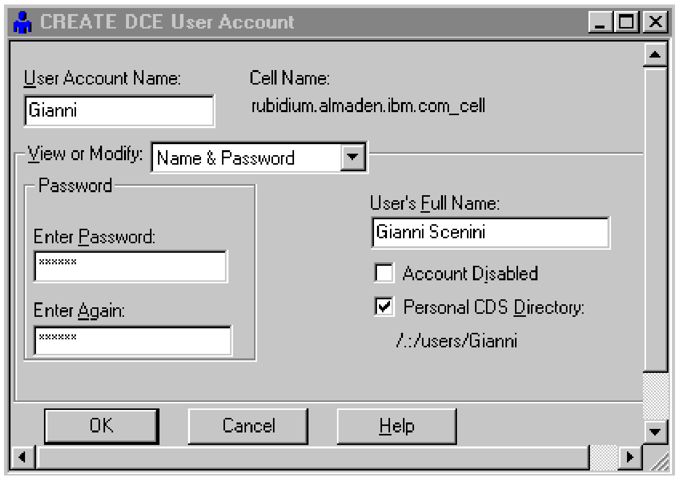
We use the existing sample Persistent Object to make it secure, using the security service provided by Component Broker. Using the security features of CB, we have to consider first what kind of security policy we would like to use for the servers and for the clients.

24.4.1 Using DCE

In order to use DCE as the authentification mechanism, we have to set up the properties of the security service of our servers and clients. Before that, any clients need to have a valid DCE account, no matter if it is a Java or C++ client. DCE uses Message Digest version 5 (MD5) algorithm for hash functions, and Commercial Data Masking Facility (CDMF) and Data Encryption Standard (DES) for encryption.

24.4.1.1 Create a new DCE account

We use the DCE director to create a new account. Select the cell and the kind of object, in our case users, where we want to create an account, and click the Create button of the list of the users of the selected cell window. In the create DCE user account panel, provide all the information needed to create a new account like the userid and the password. Make sure the Personal CDS Directory radio button is selected. In Figure 91 we can see the right settings to create an account in the DCE cell rubidium.almaden.ibm.com_cell.


CREATE DCE User Account

User Account Name: Gianni

Cell Name: rubidium.almaden.ibm.com_cell

View or Modify: Name & Password

Password

Enter Password: ******

Enter Again: ******

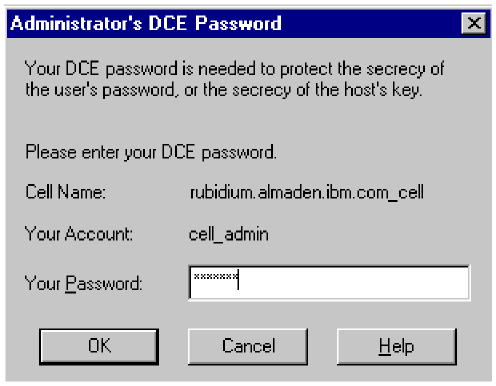
User's Full Name: Gianni Scenini

☐ Account Disabled

☑ Personal CDS Directory: /.:/users/Gianni

Figure 91. Using DCE director to create a new account

This is an account with userid Gianni. We also need to provide the cell_admin password in order to create the new account as shown in Figure 92.


Administrator's DCE Password

Your DCE password is needed to protect the secrecy of the user's password, or the secrecy of the host's key.

Please enter your DCE password.

Cell Name: rubidium.almaden.ibm.com_cell

Your Account: cell_admin

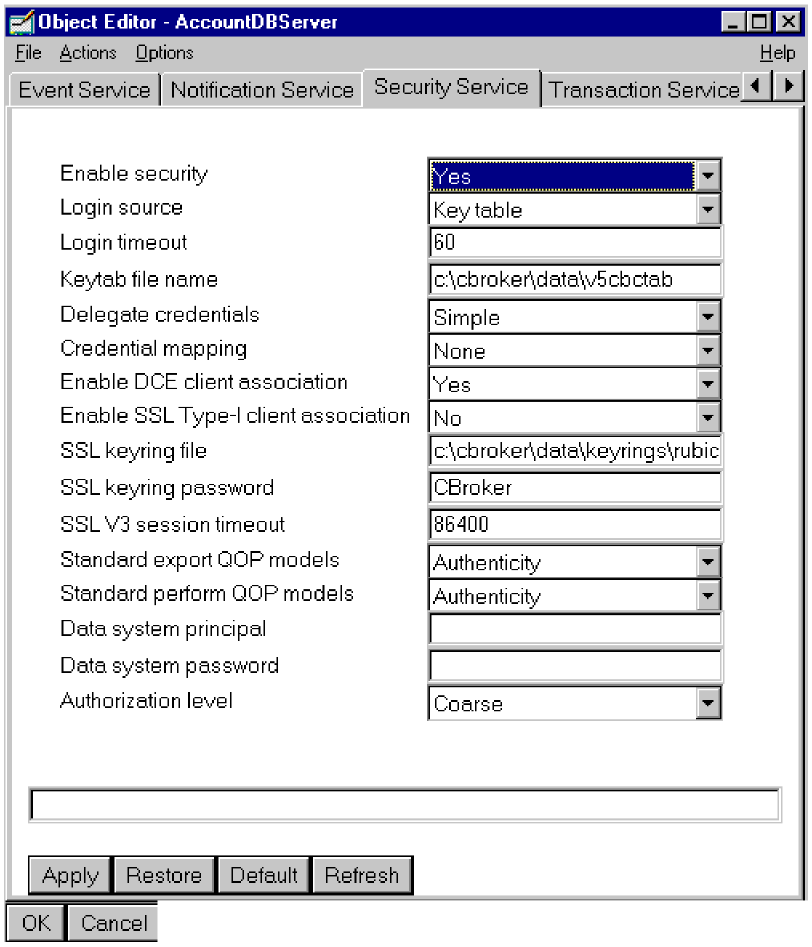
Your Password: *******

Figure 92. Administrator's DCE window

That's all we need to do. To check out if we have created it correctly, try to login to DCE using the new account. If the login is successful, we can use it for our sample.

24.4.1.2 Set the security properties of the server

First we have to set the security properties for the server. Go to the System Management User Interface (SMUI) and expand the management zone and the configuration where we have the server with all the configured applications. Select the server, click right and edit on the popup menu. Go to the Security Service page and set Yes to the Enable security list box as shown in Figure 93.


Object Editor - AccountDBServer
File Actions Options Help
Event Service Notification Service Security Service Transaction Service
Apply Restore Default Refresh
OK Cancel

Enable security	Yes
Login source	Key table
Login timeout	60
Keytab file name	c:\cbroker\data\v5cbctab
Delegate credentials	Simple
Credential mapping	None
Enable DCE client association	Yes
Enable SSL Type-I client association	No
SSL keyring file	c:\cbroker\data\keyrings\rubic
SSL keyring password	CBroker
SSL V3 session timeout	86400
Standard export QOP models	Authenticity
Standard perform QOP models	Authenticity
Data system principal	
Data system password	
Authorization level	Coarse

Figure 93. Setting the security peoperties for the server

The login source list box should be set to Key table. This means that the server authenticates itself automatically when it starts up using the keytab file generated automatically for each server host. The keytab file stores the login information in order to authenticate itself. We can also create a new keytab file for our application server instead of using the default one which is created automatically with the name **v5srvtab** for the server host. In that case, we have to supply the location of the new keytab file in the keytab file name field.

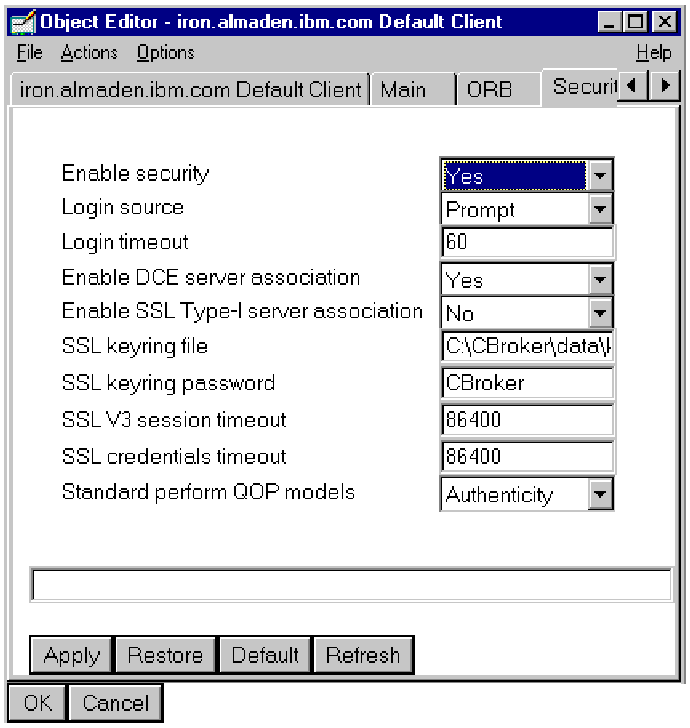
We may want to extend the login timeout to 300 as well. That's all we need to do at the moment for the server side.

24.4.1.3 Set the security properties of the clients

When we use a security server, we have to set up the security properties of the client as well. This step is different if we have a Java client or a C++ client. The DCE client needs to be installed in order to let the C++ client use the security service. In the current release of Component Broker, C++ does not support SSL authentification mechanism. The Java client supports both.

C++ client

Go to the System Management User Interface, Host Images, and expand the registered client. Expand the Client Style Images and click on the name of the client. Then right click,and on the pop-up menu, select edit. In the Security Service page, set Yes in the Enable security list box and set prompt in the login source list box. This means that a login panel appears when the user is asked to provide userid and password in the authentification of the client process. The security properties should be like those shown in Figure 94.


Object Editor - iron.almaden.ibm.com Default Client
File Actions Options Help
iron.almaden.ibm.com Default Client
Main
ORB
Securit
Apply
Restore
Default
Refresh
OK
Cancel

Enable security	Yes
Login source	Prompt
Login timeout	60
Enable DCE server association	Yes
Enable SSL Type-I server association	No
SSL keyring file	C:\CBroker\data\
SSL keyring password	CBroker
SSL V3 session timeout	86400
SSL credentials timeout	86400
Standard perform QOP models	Authenticity

Figure 94. Security properties of the client

When we run the C++ client, we are asked to login to a DCE. We can do that before launching the application or, if we do not, the DCE client pops up a login window asking for our credentials.

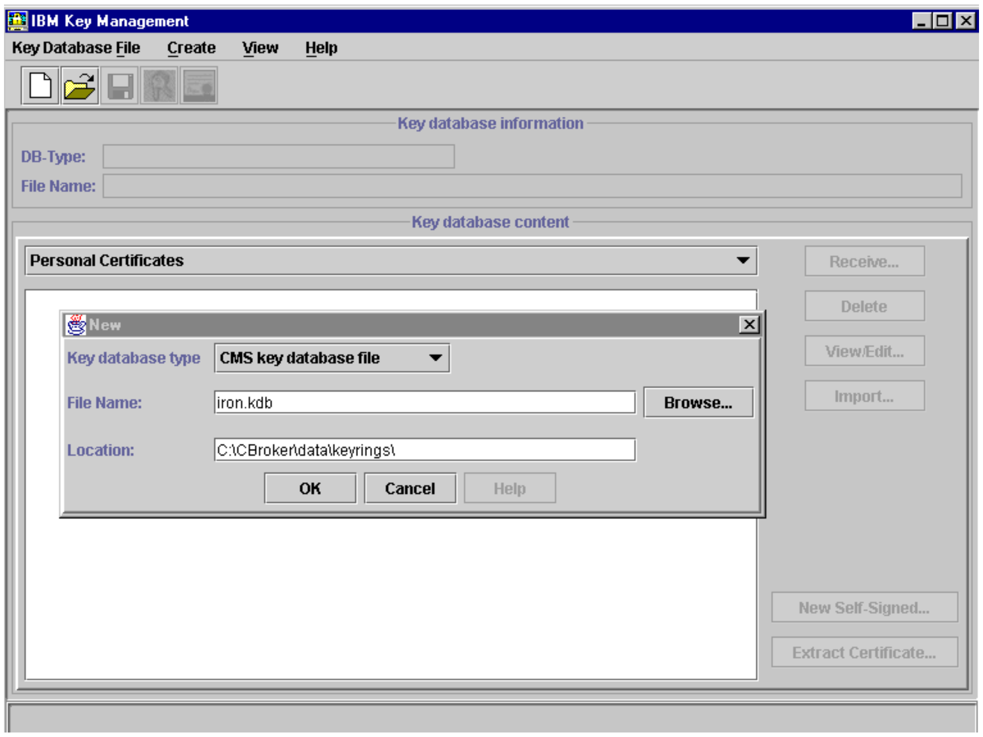
Java client
In the System Manager User Interface, we do not have the registered clients as we do in the C++ clients. Due to this fact, when we start up the Java application (Applets cannot use the DCE authentification mechanism), we have to specify that we are going to use DCE as the security mechanism. The system property **com.ibm.CORBA.EnableDCESecurity** with the value **true** defines the DCE as the security mechanism. We can even define such system property during the launch of the Java application with the Java Virtual Machine (JVM) flag -D, or put the system property with its value in the property instance that we pass to the orb during its initialization in our code.

24.4.2 Using SSL

Secure Sockets Layer (SSL) protocol was developed by Netscape. It runs above the TCP/IP protocol and provides fundamental security features such as server authentification, client authentification, and connection encryption. SSL uses public-key cryptography and digital certificates for the client and server authentification. Component Broker uses a SKit SSL implementation library for the server and SSLight for the client. It uses public-key Rivest, Shamir, Adleman (RSA) algorithm for authentification, MD5 or Secure Hash Algoritm (SHA) for hash functions, and Rivest Cipher version 2 or 4 (RC2-RC4) for encryption.

24.4.2.1 Create the digital certificate

First, we have to set the security properties of the server correctly in order to use SSL as authentification mechanism. Before doing that, we need to have a digital certificate for the server. We can get one from a Certificate Authority (CA), use the test certificate provided by Component Broker (which is named CBDevTest.kdb), or create our own self-signed certificate for testing purposes only. We can use the IBM Key Management tool to create a digital certificate for the server. Create a new key database file by selecting **new** in the **Key database file** menu. Once we have done this, we should see a window like the one shown in Figure 95. Insert the filename of the file with the kdb extension. We have to make sure that the CMS key database file is selected from the list box.



File Name:

Personal Certificates

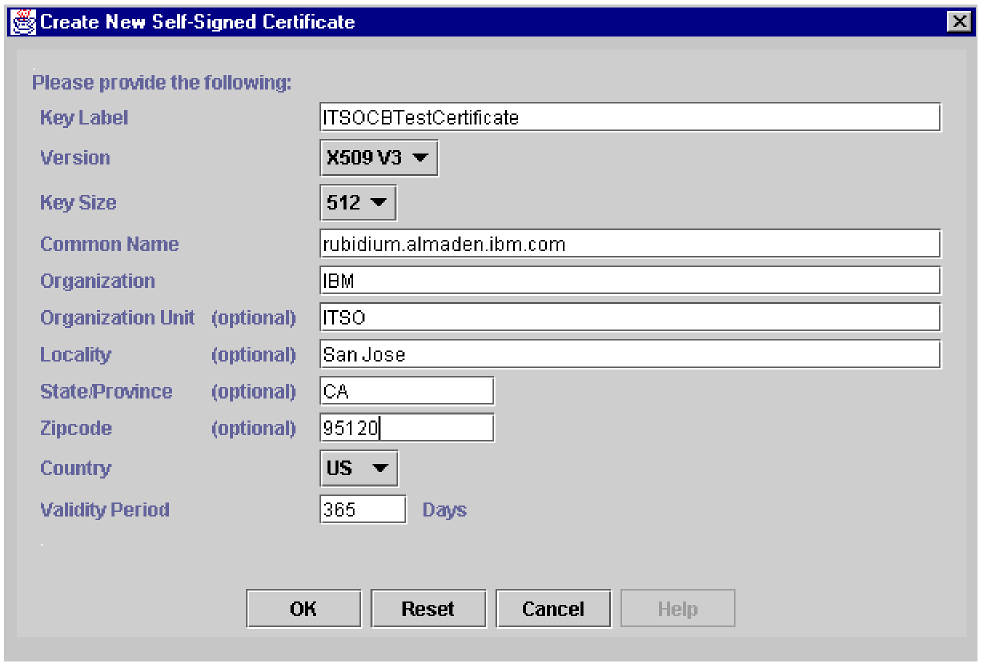
Key database type: CMS key database file

File Name: iron.kdb

Location: C:\CBroker\data\keyrings\

Figure 95. Key management too - how to create a digital certificate for the server

Click OK and another window should appear asking the password for the keyring file. Digit **Cbroker** as **default password**. After having created the key database file, we can add, modify, or delete certificates. To create a new certificate, select **Personal Certificates** in the list box and click on the **New Self Signed** button. Now we are asked to provide all the information for the x509 certificate. We should see a window like that shown in Figure 96. After this step, we should see our new certificate in the Personal Certificates list.



Please provide the following:

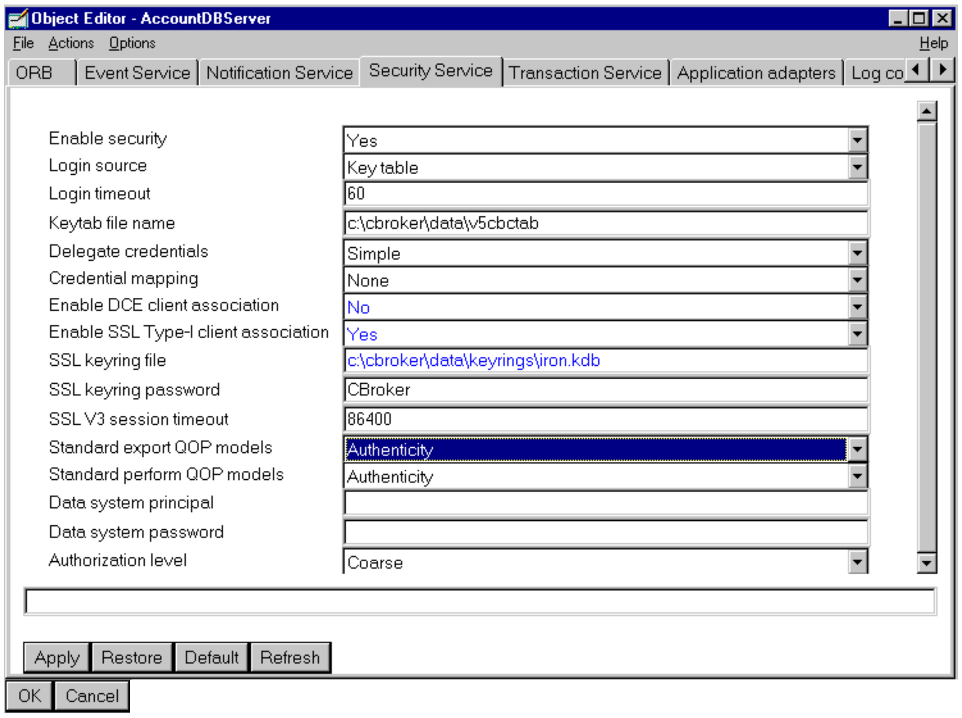
Key Label	ITSOCBTestCertificate
Version	X509 V3
Key Size	512
Common Name	rubidium.almaden.ibm.com
Organization	IBM
Organization Unit (optional)	ITSO
Locality (optional)	San Jose
State/Province (optional)	CA
Zipcode (optional)	95120
Country	US
Validity Period	365 Days

Figure 96. X509 certificate creation window

Now we have to export the certificate in a Java class format. Click **Extract Certificate** button and on the showed window, select **SSLight key database** class in the data type field, digit the class name in the Certificate file name field , and choose the location of it in your file system. Click OK and a Java class will be created.

24.4.2.2 Set the security properties of the server

Now that we have a digital certificate, we can set the security properties of the server. Go to the System Management User Interface (SMUI), and expand the management zone and the configuration where we have the server with all the configured applications. Select the server, click right and edit on the popup menu. Go to the Security Service page. We have to set enable security field to yes, Enable SSL Type-I Client association to yes, SSL keyring file to the location of your kdb file created by IBM Key Management tool (In case we use the default certificate, insert CBDevTest.kdb) , and the SSL keyring password to Cbroker (default). The security properties of the server should look like the ones shown in the Figure 97. Then click OK.

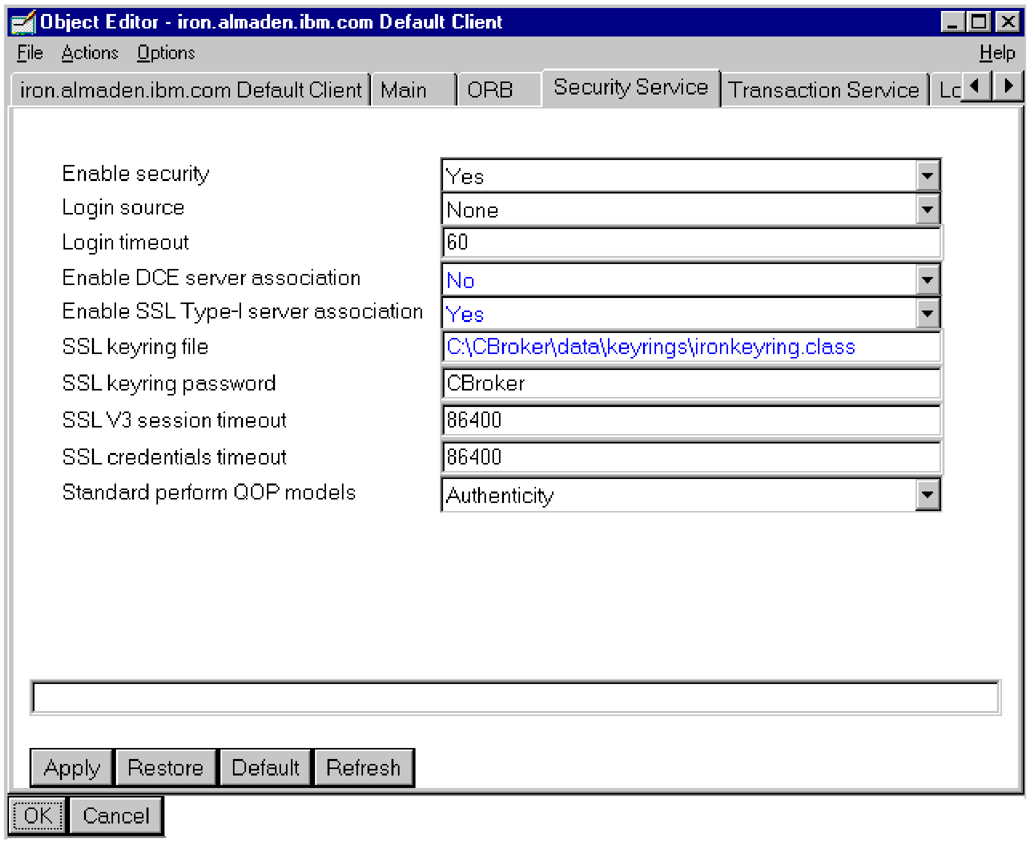

Object Editor - AccountDBServer
File Actions Options Help
ORB | Event Service | Notification Service | Security Service | Transaction Service | Application adapters | Log co
Apply Restore Default Refresh
OK Cancel

Enable security	Yes
Login source	Key table
Login timeout	60
Keytab file name	c:\cbroker\data\v5cbctab
Delegate credentials	Simple
Credential mapping	None
Enable DCE client association	No
Enable SSL Type-I client association	Yes
SSL keyring file	c:\cbroker\data\keyrings\iron.kdb
SSL keyring password	CBroker
SSL V3 session timeout	86400
Standard export QOP models	Authenticity
Standard perform QOP models	Authenticity
Data system principal	
Data system password	
Authorization level	Coarse

Figure 97. Setting server security properties

24.4.2.3 Set the security properties of the client

Go to the System Management User Interface, Host Images and expand the registered client. Expand the Client Style Images and click on the name of the client. Then right-click and on the pop-up menu, select edit. In the Security Service page set Yes. We have to set the Enable Security field to Yes, the Enable SSL Type-I Client association to Yes, the SSL keyring file to the location of your Java class file created by the IBM Key Management tool (for the CB-provided class insert CBDevTestClKeyRing.class), and the SSL keyring password to Cbroker. Figure 98 shows the correct settings of the client. Then click OK.


Object Editor - iron.almaden.ibm.com Default Client
File Actions Options Help
iron.almaden.ibm.com Default Client | Main | ORB | Security Service | Transaction Service | Lo
Apply Restore Default Refresh
OK Cancel

Enable security	Yes
Login source	None
Login timeout	60
Enable DCE server association	No
Enable SSL Type-I server association	Yes
SSL keyring file	C:\CBroker\data\keyrings\ironkeyring.class
SSL keyring password	CBroker
SSL V3 session timeout	86400
SSL credentials timeout	86400
Standard perform QOP models	Authenticity

Figure 98. Setting client security properties

C++ client

This version of CBConnector does not support SSL with C++ clients.

Java client

When we start the Java application or Java applet, we have to provide the location of the client security property file. The value of the system property **com.ibm.CORBA.ClientStyleImageURL** should point to the property file of the client. We may want to get it available on the Web. In case we use the default certificate, the default property file name is CBDevTestClDefault.properties.

Chapter 25. Putting it all together

In each of the books written on CBConnector, you can find a recapitulation of the purpose and summary of all the experiences you are supposed to have during the reading of the book. This book continues that pattern.

As you have discovered during the reading of this book, CBConnector is composed of an industry-leading set of technologies that facilitate distributed-object applications. As the only solution of its kind on the market today, it combines three critical dimensions:

- Runtime
- Systems Management
- Development

This document provides you an opportunity to become familiar with the CBConnector Toolkit and the development process itself. As a secondary benefit, you learn to use System Management. We have tried to guide you through the necessary mechanical steps of the development process by using CBConnector Object Builder and by using the System Management User Interface.

We showed you the differences in server and client applications development. You created C++ server applications. You could create and execute three different client applications and access the servers with them. They were CORBA C++, Java, and ActiveX clients.

You became familiar with the Top-Down and Bottom-Up development paradigms.

25.1 The incremental Account scenario

From Chapter 11, “Transient Object sample” on page 161 through Chapter 24, “Security service” on page 387, we were consistent in the way we let you study the incremental *Account* scenario. In the first step, we built a transient scenario, where the object's essential state of data is kept in the memory. You used a Primary Key as an identifier to find or instantiate objects on the server side. We also introduced the Copy Helper object for the first time.

In the next step, we let you work with a specialized home. Specialized homes are a very efficient way to create a new object with the essential state of data with a single method request.

We continued our incremental Account scenario, introducing a Persistent Object and the database scheme mapper. We let you store the *Account* Object and its essential state of data in a relational database. We also demonstrated the implementation and usage of a Persistent Object with an implicitly started transaction. In case there is no active transaction, the Object Transaction Service will implicitly start a transaction. This transaction will commit all changes when the method returns.

Transaction Service provides robustness in a distributed system so that the participants at different locations of the system can work together in a coordinated fashion. The Transaction Service provides a simple interface to determine the scope of a transaction and the resources to take part in a transaction.

In the next step of our journey through the incremental scenario, we enhanced the persistent scenario using the Transaction Service and explicitly starting a transaction on the client side.

To make the **Account** Business Object a transactional one, it was only a matter of correctly configuring a container with the right transactional policy.

The Object Service-oriented samples tried to explain and show you the Event and LifeCycle services. In the Event sample, you learned to send and pull events from a channel. In the LifeCycle sample, we demonstrated how to create your own Location Objects.

In the next samples of our scenarios, we used the "meet-in-the-middle" paradigm. You learned how to re-use an existing relational database table when defining your Business Object in Object Builder.

You also learned that a client program can use object's essential state of data stored in different databases, without having to know where the data is stored. Then you learned about the Query Service. Queries are evaluated against collections. Homes and reference collections are queryable and iterable collections.

We demonstrated to you another feature of CBConnector architecture, the Interlanguage Object Model allowing communication between C++ and Java Business Objects.

25.2 Thank you !

CBConnector changes the way the object-oriented, distributed applications are traditionally developed. We provided you a chance to study this development paradigm. The most important aspect for you and for us was the opportunity to help you grow with the samples we provided. We believe you should now be ready to go beyond this publication and start to look at the real implementation of your needs.

We will enhance the CBConnector series of books with new scenarios when future architectural implementations are ready in new releases of CBConnector.

We thank you for your attention and wish you the best of luck when using CBConnector in the future.

Appendix A. Rational Rose

Rational Rose is an object-oriented analysis and design modeling tool. You can use it to design your application and then export the design to Object Builder, where you can finish its implementation. The Rose Bridge is a tool for exporting from Rational Rose to Object Builder. This tool can create a Business Object, Data Object, Key and Copy Helper. You don't have to design these objects. Component Broker Toolkit offers the packaged framework classes in three Rational Rose packages:

- Managed Object Frameworks
- BOIM Application Adaptors Frameworks
- Object Services

You can recognize them as files with the *.cat* extension. To inherit from these classes, you have first import them to Rational Rose.

This chapter briefly explains how to use the Rose Bridge in your object's design. We used the transient object scenario with a specialized home for this purpose.

A.1 Set up the Rose Bridge

In order to set up Rational Rose, you have to go through several steps:

Copy the *Rose_cpp.pty* and *Rose_cpp.mnu* files from the CD-ROM's *drive:\CBConnector\Rose\* directory to the directory where you installed your Rational Rose product, for example, *C:\Rose\*.

Update the *rose.ini* file. You can find *Rose.ini* in the *drive:\winnt* directory.

Find the entry **[Rational Rose 4.0]** and update its content as follows:

```
ROSE_MENU_PATH="C:\rose\rose_cpp.mnu"
ROSE_PTY=C:\rose\Rose_cpp.pty
```

Find entry **[Virtual Path Map]** and add the following statement immediately after the entry:

```
BOSS_PATH=<fully-qualified path to Object Builder directory>\rose
```

A.2 Design a model Using Rational Rose

We deliver two models in the CD-ROM directory *drive:\CBConnector\Rose*:

- Account.mdl
- AccountSH.mdl

You can load them to Rational Rose and study them. The transient scenario with a specialized home model is displayed in Figure 99. Please refer to Chapter 12, “Specialized home sample” on page 187 for more information about the attribute and method definitions.

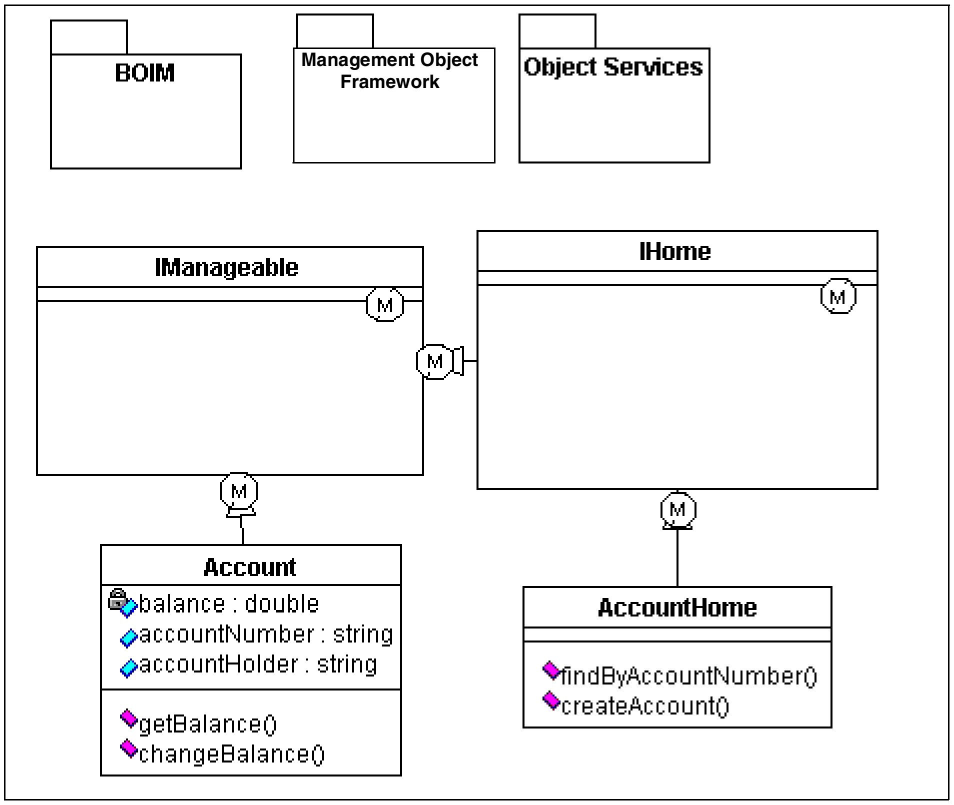


Figure 99. Specialized home object model

If you are creating a new model, follow these steps:

Create a Component Broker Object in Rational Rose. Import the packages from the *CBroker\rose\* directory. As a minimum, import the **Managed Object Framework** package.

Create the **Account** class. This class will inherit from the **IManagedClient::imanageable** class.

Create the **AcountHome** class. The *AccountHome* object will inherit from the **IManagedClient::IHome** class.

Note

Notice that in the first release of CBConnector, the Rose Bridge supports only **Adding Object/Attribute/Method**. If you delete or change Object/Attribute/Method, you should delete all files in the target directory or change the target directory.

A.3 Export your model to Object Builder

When you have finished the modelling process, you can export your model through Rose Bridge to Object Builder.

Select **File -> Export to Object Builder**.

When Rose Bridge dialog box is displayed, make the following selections:

Specify a target directory to export the Rose model to.

The **Account** class will be imported with the following:

- Business Object Implementation
- Data Object Interface
- Copy Helper
- Key Helper

The **AccountHome** class will be imported with:

- Business Object Implementation
- Data Object Interface

A.4 Modify object properties

The Rose Bridge created the appropriate objects in Object Builder. You have to manually change the following attribute definitions in Object Builder:

- The attributes' values in the **Account** Interface.
- Add the `NotEnough` exception in the Constructs page.
- Select **accountNumber** as the **Primary Key** in **AccountKey** properties.
- Select **accountNumber** and **accountHolder** as **AccountCopy** attributes.
- In the Business Object, **AccountBO**, select **AccountKey** as a Key selection.

Appendix B. Tips and hints

On the following pages, you can find useful tips for your development environment.

B.1 Two-compiler environment

If you are working in an environment that requires both ActiveX and CORBA clients' runtime and SDKs installed on the same system, you will experience conflicts, which we can help you to avoid. Here are the general steps:

- Create separate directory for ActiveX header files.
- Define a separate environment for each compiler.

B.1.1 Creating a header directory for an ActiveX client

You need to create a separate directory to contain CBConnector header files supplied with the ActiveX client SDK. By default, the installation stores the ActiveX client SDK files in the *\Cbroker\include* directory.

Below are the header files using the *MS_INC_DIR* variable in their implementation. These are intended only for use with a Microsoft Visual C++ compiler. If these header files are used together with the VisualAge C++ compiler, errors occur:

- ctype.h
- limits.h
- stdio.h
- stdlib.h
- string.h
- time.h
- .hp1.sy\types.h
- .hp1.wchar.h
- .hp1.windows.h

To create the directories, perform the following steps:

- Create the *VCPP* subdirectory under *\CBroker\include*.
- Move the above-specified ActiveX header files to this directory.
- Modify the *INCLUDE* variable.

B.1.2 Preparing separate environments

Create separate batch files with `SET` statements to accomplish that. In this manner, you can create a window for each compiler environment.

VisualAge C++ environment:

The following are suggestions for the `INCLUDE` and `PATH` variables:

- Modify the *INCLUDE* variable by removing any reference to Microsoft Visual C++ directories.
- Ensure that the *INCLUDE* variable contains the *\CBroker\include;* reference before the VisualAge C++ directory references.
- Modify the *PATH* variable by removing any references to Microsoft Visual C++.

Microsoft Visual C++ environment:

The following are suggestions for the *INCLUDE* and *PATH* variables:

- Modify the *INCLUDE* variable by removing any reference to the VisualAge C++ directories.
- Modify the *INCLUDE* variable by adding the new directory you created for ActiveX header files.
- Ensure that the *\CBroker\include* and the *\Cbroker\include\VCPP* are before any reference to Microsoft Visual C++ directories.
- Modify the *PATH* variable by removing any references to VisualAge C++.

B.2 Backup and restore

If you are experimenting with Component Broker Connector with different server configurations, it is a good idea to have the possibility to reset the environment to a specific baseline. This may be the configuration after the activation of the name server or any other stable configuration.

We developed a simple way of backing up and restoring the server environment, which is described in the next to sections. You can deinstall all components also with the System Management User Interface as described in the *Quick Beginnings Guide*.

Note

This procedure is not a supported function, but it might help in many cases.

B.2.1 Backing up the server environment

To be able to restore the server environment at any time, follow these steps:

Stop all server images using the System Management User Interface.

Stop all services:

```
net stop cbconnector
net stop "DCE Auto-Start Service"
net stop "DCE CDS Advertiser Service"
net stop "DCE CDS Server Service"
net stop "DCE Distributed Time Service"
net stop "DCE Name Service Gateway"
net stop "DCE Security Client Service"
net stop "DCE Security Server"
```

If you started the ORB daemon manually, stop it with CTRL-C or kill the process using the task manager.

Back up the following directories:

<drive>:\CBroker\Data

<drive>:\CBroker\Etc\IRWork

<drive>:\Opt\Digital\dcelocal including all subdirectories

Restart the services:

```
net start CBConnector
net start "DCE Auto Start Service"
```

Log in as DCE cell administrator:

```
dce_login <cell_admin userid> <password>
```

B.2.2 Restoring the server environment

Stop all services:

```
net start CBConnector
net stop "DCE Auto-Start Service"
```

If you started the ORB daemon manually, stop it with CTRL-C or kill the process using the task manager.

Restore the following directories from your backup:

```
<drive>:\CBroker\Data

<drive>:\CBroker\Etc\IRWork

<drive>:\Opt\Digital\dcelocal, including all subdirectories
```

Uninstall all Component Broker applications that are not in the backup configuration: **Control -> Add/Remove Programs**

Restart the services:

```
net start CBConnector
net stop "DCE Auto-Start Service"
```

Login as DCE cell administrator:

```
dce_login <cell_admin userid> <password>
```

B.3 Management zones and configurations

In the default configuration of the CBConnector System Management application, only one Management Zone with the sample configuration is defined. This sample configuration includes the name server.

The name server, like every other server, is inspected during activation to see if any changes are required. Unless the administrator has altered any of naming objects in the configuration (cell or workgroup), no changes should be required in the name server. It remains up and running. You can read more about this subject in the CBConnector online documentation.

Placing new applications in a separate Management Zone will reduce the activation time, since all of the things in the sample zone and configuration are unaffected.

You should create another Management Zone (for example, Account Management Zone) and insert a new configuration (for example, Account Configuration). On this new configuration, you can configure your applications in the usual way. An activation of this configuration does not alter the running status of the name server and is therefore a lot faster.

B.4 Consistency check for a DDL

Sometimes, the Systems Management EUI provides the user with no explanation as to why a DDL file did not load. In these cases, all the user will see is something like `Failed to load DDL file during the installation process`. One way to get more information in these cases is the following:

CD x:\CBroker\data (where *x:\CBroker\* is where Component Broker Connector was installed)

Somsmaiic <DDL file name> <agent alias>

```
    Usage : somsmaiic <ddl file name> ç<agent alias> ç<host image>ŸŸ where
      <ddl file name> is the name of the DDL file to process
      <agent alias>   is the name of the CBConnector System Manager
                      or Agent to
                      load the DDL file into (if this is omitted, then
                      only a syntax check is performed)
      <host image>    is the name of the CBConnector Host Image
                      (if this is different
                      from <agent alias>, which typically it won't be)
```

B.5 Java related considerations

To run a Java client in Netscape, you have to initialize the ORB in the following way:

```
com.ibm.CORBA.iiop.ORB orb = null;
      orb = (com.ibm.CORBA.iiop.ORB) ORB.init (args, props);
```

This ensures that the ORB that you are finding is the IBM ORB, rather than the Visigenic ORB that is built into the Netscape browser.

Java clients and server applications with Java BOs cannot exist in the same environment because the class files for the Java ORB and Java BOs conflict with each other.

In CBConnector Release 1.2, an object model which uses Java BOs must be compiled on a CBConnector server machine with the CBConnector Toolkit installed. If you separate the development environment from the server runtime environment, some of the class files needed to compile Java BOs will not be found.

Appendix C. Client development with VisualAge for C++

When implementing the GUI part of the client in VisualAge for C++, we encountered certain problems. If you are as new to VisualAge C++ as we were, you might want to read these tips and hints.

C.1 The development environment

We used VisualAge for C++ as our development environment, and used Workframe to compile and link our application.

You need to have Object Builder installed in order to access all the .lib files needed to link the clients.

Create a new default project in Workframe. Make sure that the environment settings like the *Working* directory and target files are set. Generate visual parts from within the Workframe environment, so that the current directory in the VisualBuilder is set correctly.

C.2 Writing the user code

When creating the nonvisual parts, such as the **AccountWrapper** and the **AcctCBCProxyWrapper**, you need to specify the *.hpv* and *.cpv* files for the feature code. The feature code is the skeleton VisualAge generates for you based on the methods and attributes specified for this part. You fill in this code outside the Visual Builder. Be aware that if you regenerate feature code, it will be appended to the *.hpv* and *.cpv* files. This is so that the code you already wrote will not be overwritten, but it means that you need to "clean up" these files after each generation, matching the new skeleton with the user-defined code.

Remember to generate main part code for your main window or for the window you want to be opened when you start the application. Also, remember to define the necessary header files for the VisualAge part in the class editor pane of each class. This mainly applies to the nonvisual wrapper classes.

When working with lists displaying objects, you must specify how these objects are to displayed as strings in the list. This is defined in the settings of the listbox itself, and you need to create a *strgen.hpp* file for the convert object to string method. An example of this is shown in Chapter 19, “Query Service — reuse of an existing table” on page 291.

C.3 Compiling and linking

Within Workframe, you need to set the compile options to multithread under **Options -> Compile -> Object -> Library** selection. You also need to define that templates are being used under **Options -> Link -> Templates**. Specify not to search the extended dictionary by unchecking this option under **Options -> Link -> Processing**. Finally, in order to be able to link to the CORBA-specific library files, specify the following files under **Options -> Link -> File Names -> Add libraries**:

C:\cbroker\lib\somorori.lib C:\cbroker\lib\sompmcii.lib C:\cbroker\lib\somosa1i.lib C:\CBroker\lib\sompmg1i.LIB C:\CBroker\lib\somibl1i.LIB.

To get an exact list of which files are needed for a specific client, look at the project's *.lib* files defined in the **VisualAgeCpp\Generated** folder of the VisualAge client, or look at the *mkacc.bat* file in the **SimpleCpp** folder for the client.

To create the makefile, choose the **Make** function from the **Project** menu. Select **Compile -> Link -> Resource Compile -> Resource Precompile**

Make sure to select the *.lib* file for this client, as generated by Object Builder (for example *AccountDBC.lib* for the Persistent Object sample).

Then choose the **Make** function from the **Project** menu to compile and link your application.

C.4 From transient to persistent in five minutes

With our modularized client development approach, we were able to create the Persistent Object sample from the transient one in only five minutes for the client part. The GUI is unchanged, and for the **acctproxy**, it is only the scope name that needs to be altered. When the client DLL and header files generated by Object Builder are replaced, just make and run it!

These are the steps that need to be performed:

- Copy the directory for your C++ client to a new directory.
- Replace the *Account.hh* and *AccountKey.hh* with the header files generated from Object Builder when you created the persistent model.
- Replace the *AccountTRC.dll* and *AccountTRC.lib* with *AccountDBC.dll* and *AccountDBC.lib* generated from Object Builder when you created the persistent model.
- Replace the *acctproxy.obj* with the new one, which has been edited to include the new scope name.
- Open the Workframe project in this directory and set the source and *Working* directory to your persistent directory. Rename the executable file to *AccoutnDBC.exe*.
- Generate a new makefile, and make sure you select the new files.
- Compile, link, and run!

Appendix D. Using the additional material

This redbook also contains additional material in CD-ROM format, and Web material. See the appropriate section below for instructions on using or downloading each type of material.

D.1 Using the CD-ROM or diskette

The CD-ROM that accompanies this redbook contains the code for the samples used throughout this book.

There are three directories on the CD-ROM for each sample described in this redbook: Client, Install, and Server. The Client directory is divided into the three subdirectories holding the ActiveX, VisualAge C++, and Java clients. In the Install directory, you can find the installation executable. The server subdirectory holds the Object model and the working subdirectory with Data and Disk1 subdirectories.

There are a total of 6800 files in 633 folders on the CD-ROM.

D.1.1 System requirements for using the CD-ROM or diskette

The following system configuration is recommended for optimal use of the CD-ROM.

Hard disk space:	500MB additional to CB installation
Operating System:	Windows NT 4.0 SP 4
Processor:	Pentium II 200 or higher
Memory:	128MB minimum
Other:	CD-ROM drive, 800x600 SVGA display

D.1.2 How to use the CD-ROM

Please refer to section 1.2, “The best way to use this book” on page 5 for information about how to use the CD-ROM.

D.2 Locating the additional material on the Internet

The CD-ROM associated with this redbook is also available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG242033/

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number.

D.3 Using the Web material

The additional Web material that accompanies this redbook includes a zip file containing all the files of the CD-ROM.

See section 1.1, “How this book is organized” on page 3 for a description of the files and directories contained in the zip file.

D.3.1 System requirements for downloading the Web material

The following system configuration is recommended for downloading and using the additional Web material.

Hard disk space:	500MB additional to CB installation
Operating System:	Windows NT 4.0 SP 4
Processor:	Pentium II 200 or higher
Memory:	128MB minimum

D.3.2 How to use the Web material

Download the zip file to your workstation and un-pack the file using WinZip to the root of a drive mapped as **O**.

Use the sample code as described in section 1.2, “The best way to use this book” on page 5

Appendix E. Special notices

This publication is intended to help developers and architects understand the IBM Component Broker solution and product offering for distributed object computing. The information in this publication is not intended as the specification of any programming interfaces that are provided by IBM WebSphere Application Server Enterprise Edition Component Broker. See the PUBLICATIONS section of the IBM Programming Announcement for Component Broker for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

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

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer

AIX	AS/400
AT	CICS
CICS/ESA	CT
DB2	IBM ®
MQSeries	Netfinity
OS/2	RACF
RS/6000	SecureWay
SecureWay	SP
System/390	VisualAge
WebSphere	400

Appendix F. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

F.1 IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 425.

- *IBM Component Broker Connector Overview*, SG24-2022
- *IBM CBConnector Cookbook Collection: CBConnector Bank User Guide*, SG24-5121
- *IBM CBConnector Cookbook Collection: CBConnector Bank Implementation*, SG24-5119

F.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at ibm.com/redbooks for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	**Collection Kit Number**
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

F.3 Other publications

These publications are further information sources:

- *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - Getting Started with Component Broker*, SC09-4433
- *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - Programming Guide*, SC09-4442
- *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - CICS and IMS Application Adaptor Quick Beginnings*, SC09-4439
- *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - System Administration Guide*, SC09-4445
- *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - Application Development Tools*, SC09-4448
- *WebSphere Application Server Enterprise Edition Component Broker for Windows NT - Advanced Programming Guide*, SC09-4443
- *Writing Enterprise Beans in WebSphere*, SC09-4431

F.4 Other resources

These publications are relevant as further information sources:

- *Orbix 2, Programming Guide*
- *Orbix 2, Reference Guide*
- *The Essential Distributed Objects Survival Guide*, ISBN 0471129933
- *Client/Server Programming with JAVA and CORBA*, ISBN 047124578X
- *Understanding CORBA*, ISBN 0134598849

How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site ibm.com/redbooks**

  Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

  Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the IBM Redbooks fax order form to:

	e-mail address
In United States or Canada	pubscan@us.ibm.com
Outside North America	Contact information is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and Redbooks by accessing the IBM Intranet Web site at http://w3.itso.ibm.com/ and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at http://w3.ibm.com/ for redbook, residency, and workshop announcements.

IBM Redbooks fax order form

Please send me the following:

Title	**Order Number**	**Quantity**

First name	Last name	
Company		
Address		
City	Postal code	Country
Telephone number	Telefax number	VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date	Card issued to	Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Glossary

Abstract Interface. An abstract interface is one that is introduced in order to specify required behaviors without providing an actual implementation for them. The implementation must be provided by a derived interface. Often, the derived interface achieves this by delegating responsibilities to another object.

Access Identity. The identity of a principal used to specify access policies pertaining to that principal.

Access Policies. The rules that define whether a principal should be allowed to perform a particular operation on a particular object.

Administrative Interface. The interface of an object that defines its administrative and systems management behavior. Typically, the administrative interfaces of an object are only used by systems management and administration programs.

Application Access Policy. The mechanisms built into an application to control access to resources that it contains. Application access policies are enforced within the application implementation, although possibly with the assistance of security services for acquiring principal credentials. (See also object invocation access policy.)

Application Adaptor (AA). Provides a home for, and a certain quality of service to, its Managed Objects. It is responsible for providing systems capabilities such as identity, caching, persistence, recoverability, concurrency, and security to its Managed Objects.

Application Adapter Mixin. A special object provided to a business object by an instance manager. The mixin object provides an implementation of various interfaces that are inherited in the CBConnector server environment.

Application Object. An application object is an object that implements the transient and persistent state of actively executing applications.

Attribute. An identifiable association between an object and a value. An attribute, A, is made visible to clients as a pair of operations: getA() and setA(). Read-only attributes only generate a get operation.

Audit Identity. The identity of a principal used to audit that principal's actions in the security system. Typically, a principal's audit identity is anonymous to the principal.

Authentication. The process of assuring that a principal is who they say they are; they are authentic. There are numerous ways for performing authentication which generally depend on one or more of something the principal knows (such as a secret or password), something the principal has (a badge or door key), or something the principal is (biometrics, a signature, finger-print, voice-print, retina-pattern, and so forth).

Basic Business Object. Single entity business object containing business methods. The logic and state is intended for use within business applications.

Basic Object Adaptor (BOA). BOA is a component of the ORB and exists on each CBConnector server. Its main function is to analyze each request received by the ORB and dispatch it to the object implementation that is the target of the request.

Behavior. The observable effects of an object performing the requested operation, including its results binding. See language binding, dynamic invocation, static invocation, or method resolution for alternatives.

Bind Policy. Bind policy determines which server of a CBConnector server group should be selected to receive the next request for a specific object. Object Transaction Monitor determines which bind policies apply to a particular object according to definitions done by the CBConnector system administrator.

Business Object. An object containing business methods (logic) and state that is intended for use within business applications. Business objects are Managed Objects. In some contexts, the term business object in this book is used to refer to a business object class. It may also be used to refer to a composition of business object classes.

Computer-Aided Software Engineering (CASE). CASE refers to the use of computerized tools for the collection and transformation of application domain information necessary during software construction. Some CASE tools emphasize the front-end of the development lifecycle. These are referred to as Upper CASE tools, as distinguished from Lower CASE tools which emphasize activities near the end of the development lifecycle.

Common Data Representation (CDR). Low-level data representation in the General Inter-ORB Protocol (GIOP). The language representation is marshaled and demarshaled to and from the CDR format for transmission over a wire by the ORB.

Cell Directory Service (CDS). A component of DCE that provides the ability to assign a set of attributes to a name structured into a directory hierarchy. The CDS is used primarily within DCE to store RPC bindings, but its use is not limited to this.

Client. The code or process that invokes an operation on an object.

Collection. A logical grouping of elements. In the context of this book, each element is an object.

Common Data Model (CDM). The Common Data Model is a template of the CBConnector configuration data structure.

Common Data Store (CDS). The Common Data Store is a mechanism for storing structured data. The data in it is arranged as an arbitrarily complex tree of named objects each of which can have any number of named values.

Composed Business Object. Consists of multiple basic business objects.

Compound Name. A sequence of simple names that represents a traversal path through a Name Tree relative to some starting point. (See also simple name.)

Container. A component of an instance manager that provides physical and administrative boundaries for Managed Objects. For example, a container holds Managed Objects and defines policies for them.

Common Object Request Broker Architecture (CORBA). CORBA is an architectural standard proposed by Object Management Group (OMG), an industry standards organization, for creating object descriptions that are portable among programming languages and execution platforms.

CORBAservices. The CORBAservices specify the standard interfaces of the OMG object services.

Credentials. Information stored in a Security Context about a principal. The information is related to a session established between a principal and a CBConnector server.

Data Object. An object that provides an object-oriented rendering of application and data. The data typically comes from data stores such as relational databases or CICS.

Data Type. A categorization of values operation arguments, typically covering both behavior and representation (such as the traditional non-object-oriented programming language notion of type).

DII. Dynamic Invocation Interface provides a dynamic interface to the ORB. With the help of the Interface Repository, a client can determine the interface at runtime and dynamically invoke a method on it.

Domain. As a concept important to interoperability, a domain is a distinct scope, within which common characteristics are exhibited, common rules observed, and over which a distribution transparency is preserved.

DSI. Dynamic Skeleton Interface allows a client to invoke a method on an object it had no knowledge of at compile time.

Dynamic Invocation. Constructing and issuing a request whose signature is possibly not known until runtime.

Dynamic Skeleton. An interface-independent type of skeleton. It is used by CBConnector servers to handle requests whose signatures are not known until runtime.

Embedded Aggregation. A form of aggregation where the sub-objects remain visible from outside of the aggregate.

ENC. Extended Naming Context - A specialization of a naming context with extensions for properties, query, identity, administration, and name strings.

Externalized Object Reference. An object reference expressed as an ORB-specific string. Suitable for storage in files or other external media.

Framework. Ted Lewis, et al., define a framework as "an object-oriented class hierarchy plus a built-in model which defines how the objects derived from the hierarchy interact with one another."

Home. The logical owner of a Managed Object. A Managed Object has only one home. A home has factory and collection interfaces.

Interface Definition Language (IDL). IDL is a contractual, neutral, and declarative language that specifies an object's boundaries and its interfaces. IDL provides operating system and programming language independent interfaces to all services and components that resides on a CORBA bus.

IIOP. Internet Inter-ORB Protocol is an industry standard protocol. It defines how General Inter-ORB Protocol (GIOP) messages are exchanged over a TCP/IP network. The IIOP makes it possible to use the Internet itself as a backbone ORB through which other ORBs can bridge.

Implementation Interface. An interface introduced as a derivative of the most specialized interface of the object. The implementation interface is intended to designate the type of a specific implementation - the Type Id (see CORBA specification) of the implementation interface can be used within CORBA to differentiate implementations of the same interface. The implementation interface is also used to bring together the operational interface with the specific administrative interface relevant to that particular implementation.

Implementation. A definition that provides the information needed to create an object and allow the object to participate in providing an appropriate set of services. An implementation typically includes a description of the data structure used to represent the core state associated with an object, as well as definitions of the methods that access that data structure. It will also typically include information about the intended interface of the object.

Implementation Definition Language. A notation for describing implementations. The implementation definition language is currently beyond the scope of the ORB standard. It may contain vendor-specific and adaptor-specific notations.

Implementation Inheritance. The construction of an implementation by incremental modification of other implementations. The ORB does not provide implementation inheritance. Implementation inheritance may be provided by higher-level tools.

Implementation Object. An object that serves as an implementation definition. Implementation objects reside in an implementation repository.

Implementation Repository. A storage place for object implementation information.

Inheritance. The construction of a definition by incremental modification of other definitions. See interface and implementation inheritance.

Instance. An object is an instance of an interface if it provides the operations, signatures and semantics specified by that interface. An object is an instance of an implementation if its behavior is provided by that implementation.

Instance Manager. See Application Adaptor

Instance Manager Mixin. See Application Adaptor Mixin

Interface. A listing of the operations and attributes that an object provides. This includes the signatures of the operations and the types of the attributes. An interface definition ideally includes the semantics as well. An object satisfies an interface if it can be specified as the target object in each potential request described by the interface.

Interface Inheritance. The construction of an interface by incremental modification of other interfaces. The IDL language provides interface inheritance.

Interface Object. An object that serves to describe an interface. Interface objects reside in an Interface Repository.

Interface Repository. A storage place for interface information.

Interface Type. A type satisfied by any object that satisfies a particular interface.

Interoperability. The ability for two or more ORBs to cooperate to deliver requests to the proper object. Interoperating ORBs appear to a client to be a single ORB.

Interoperable Object Reference (IOR). An IOR keeps information about the type and key of an object and the communications profiles needed to contact the CBConnector server and locate the object.

Junction. A junction represents a transition in a Name Tree federation between different implementations of a naming context. If a DB-based naming context is bound into a CDS-based naming context, that binding forms a junction.

Language Binding or Mapping. The conventions by which a programmer writing in a specific programming language accesses ORB capabilities.

Lock. A lock is used to coordinate concurrent use of a resource.

Managed Object. An object that is managed by an instance manager. Managed Objects can have a complex composition of inheritance and containment relationships.

Master Container. The container owned by the Root Application Adaptor

Metadata. Metadata is the self-descriptive information that can describe both services and information. With metadata, new services can be added to a system and discovered at run time.

Method. An implementation of an operation. Code that may be executed to perform a requested service. Methods associated with an object may be structured into one or more programs.

Method Resolution. The selection of the method to perform a requested operation.

Mixin Object. See instance manager Mixin.

Multiple Inheritance. The construction of a definition by incremental modification of more than one other definition.

Naming Context. A naming context is a container of name bindings that associates a human readable name to an object reference. Naming contexts support the CosNaming::Naming Context interface. (See also ENC.)

Object. A combination of state and a set of methods that explicitly embodies an abstraction characterized by the behavior of relevant requests. An object is an instance of an implementation and an interface. An object models a real-world entity, and it is implemented as a computational entity that encapsulates state and operations (internally implemented as data and methods) and responds to requestor services.

Object Adaptor. The ORB component which provides object reference, activation, and state related services to an object implementation. There may be different adaptors provided for different kinds of implementations.

Object Builder. This is an application development tool that helps users develop business objects in CBConnector. It provides a set of SmartGuides to help users define their business objects, and it will generate all the necessary definition and implementation files in IDL, C++ and/or Java.

Object Creation. An event that causes the existence of an object that is distinct from every other object.

Object Destruction. Object destruction is a physical deletion of an object from main memory and permanent storage.

Object Invocation Access Policy. The mechanisms within the systems that transparently control access to objects. The object invocation access policy is enforced by the system without application involvement. An event that causes an object to cease to exist.

Object Reference. A value that unambiguously identifies an object. Object references are never reused to identify another object.

Object Services. IBM's CORBAservices implementation and enhancements. These include Naming, Security, Life Cycle, Event, Externalization, Identity, Query, Transaction, and Concurrency services.

Object Management Group (OMG). OMG is a consortium of vendors with the mission to define standards pertaining to object-oriented, distributed systems. The OMG is responsible for defining CORBA, CORBAservices, and CORBAfacilities in accordance with the Object Management Architecture.

Object Transaction Monitor. Object Transaction Monitor enhances the ORB by allowing management of business object instances across CBConnector servers. It minimizes client complexity by having a single system image and single object image for objects across groups of CBConnector servers.

One-way Request. A request where the client does not wait for completion of the request, nor does it intend to accept results. Contrast with deferred synchronous request and synchronous request.

Operation. A service that can be requested. An operation has an associated signature, which may restrict which actual parameters are valid.

Operational Interface. The interface of an object that defines its operational behavior. Typically, the operational interface of an object is used by general business applications - also known as the API (application programming interface). (See also administrative interface.)

Object Request Broker (ORB). ORB provides the means by which clients make and receive requests and responses.

ORB core. The ORB component which moves a request from a client to the appropriate adaptor for the target object.

Open Software Foundation (OSF). OSF is a consortium of vendors who have collaboratively produced a reference implementation of the Distributed Computing Environment (DCE), along with several other de-facto standards such as Motif.

Object Tranasction Service (OTS). OTS is the CORBA services specification for managing atomic units-of-work over a series of method requests on recoverable objects.

Parameter passing Mode. Describes the direction of information flow for a operation parameter. The parameter passing modes are IN, OUT, and INOUT.

Persistent Object. An object that can survive the process or thread that created it. A persistent object exists until it is explicitly deleted.

Principal. A principal is a human user or system entity that is identified (through a security name) to the security system. Principals can be authenticated by the security system.

Privilege Attribute. Security information associated with a principal that can be used to decide what that principal can access.

Propagation. A function of the Transaction Service that transfers the transactional context of a client along with a method request to a served object. The Transaction Service supports both implicit and explicit propagation of a transaction context. Implicit propagation will occur automatically as the result of invoking any method on an object whose class inherits from CosTransactions::Transactional. Explicit propagation will occur only when the client obtains a context object and passes that as an explicit argument on a method request.

Recoverable Object. An object whose data is effected by committing or rolling back a transaction.

Proxy. The proxy object has the same interface as the server object it represents. Instead of having the actual method implementation, its methods communicate with the ORB.

Recoverable Server. A server containing one or more recoverable objects.

Referential Integrity. The property ensuring that an object reference that exists in the state associated with an object reliably identifies a single object.

Repository. See Interface Repository and Implementation Repository.

Request. A client issues a request to cause a service to be performed. A request consists of an operation and zero or more actual parameters.

Results. The information returned to the client, which may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.

Restart Daemon. A restart facility provided by the Transaction Service that may be used to automatically restart transactional processes.

Restart repository. A file holding information about the transactional programs being run on a machine. It is used by the restart daemon during restart of failed transactional processes.

Root Application Adaptor. The Root Application Adaptor provides a container that contains other instance manager containers. The Root Application Adaptor Container is a bootstrap mechanism integrated directly into the CBConnector server environment

Security Name. The identity of a principal used to authenticate that principal to the security system. A set of security attributes are associated with a principal through the principal's security name, including the principal's access identity, audit identity, privileges, and so on. The security name is often referred to as a user ID.

Server. A process implementing one or more operations on one or more objects.

Server Object. An object that responds to a request for a service. A given object may be a client for some requests and a server for other requests.

Service Context. The Service Context is the information that flows from the client to the server (or from the server back to the client). The information is used to inform the server of the context running on the client; for any service, the appropriate behavior on the server is defined by the context of the client.

Service's Context. Services often have their own context. Service's context is information that is used to control the behavior of the service–that is, the "environment" in which the service executes.

Signature. Defines the parameters of a given operation including their number order, data types, and passing mode, the results if any, and the possible outcomes (normal vs. exceptional) that might occur

Simple Name. A name in a naming context that identifies a particular name binding. A simple name is normally composed of an ID field and a kind field. Also referred to as a name component. (See also compound name.)

Single Inheritance. The construction of a definition by incremental modification of one definition. Contrast with multiple inheritance.

Skeleton. The object-interface-specific ORB component that assists an Object Adaptor in passing requests to particular methods.

System Management Application (SMAPPL). SMAPPL is a central point in which definitional configuration data is held. It also contains a copy of the Common Data Model (CDM). Only one host houses the central point in a management network topology.

SMI. Systems Management Interface - An interface supported by an object that supports operations that allow the object to be managed by a systems management service. Typically, the systems management interface is introduced to object services in the administrable interface.

State. The time-varying properties of an object that affect that object's behavior.

Static Invocation. Constructing a request at compile time. Calling an operation through a stub procedure.

Stub. A local procedure corresponding to a single operation that invokes that operation when called.

Synchronous Request. A request where the client pauses to wait for completion of the request. Contrast with deferred synchronous request and one-way request.

Transactional Object. An object whose behavior is affected by being invoked within the scope of a transaction.

Transactional Server. A server containing one or more transactional objects.

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

UUID. Universally Unique Identifier - A value constructed with an algorithm that provides a reasonable assurance that the identity value is unique within the known universe. Typically, UUIDs are 16 bytes long.

Value. Any entity that may be a possible actual parameter in a request. Values that serve to identify objects are called object references.

List of abbreviations

AA	Application Adaptor
AIX	Advanced Interactive Executive
AMS	Application Management Specification
API	Application Programming Interface
APPC	Advanced Program-To-Program Communication
BO	Business Object
BMS	Basic Mapping Support
BOA	Basic Object Adaptor
BOIM	Business Object Instance Manager alias for BOIM Application Adaptor
CASE	Computer-Aided Software Engineering
CBConnector/SM	CBConnector Systems Management
CDM	Common Data Model
CDS	Common Data Store
CDR	Common Data Representation
CICON	CICS/IMS Connection
CICS	Customer Information Control System
CLI	Call Level Interface
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CRUD	Create, Retrieve, Update, and Delete
DAO	Data Access Object
DB2	Database 2
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DII	Dynamic Invocation Interface
DLL	Dynamic Link Library
DO	Data Object
DSI	Dynamic Skeleton Interface
DSOM	Distributed System Object Model
DTD	Document Type Description (part of the XML standard)
DTS	Direct-to-SOM
ECD	Edit, Compile, Debug
ECI	External Call Interface
ESIOP	Environment Specific Inter-ORB Protocols
GIOP	General Inter-ORB Protocol
HOD	IBM Host on Demand
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol

IM	Instance Manager alias for Application Adaptor
IMS	Information Management System
IOM	Interlanguage Object Model
IOR	Interoperable Object Reference
IR	Interface Repository
ISV	Independent Software Vendor
JIT	Just-in-Time
MDL	Model Definition Language
LU 6.2	Logical Unit Type 6.2
MFS	Message Format Services
MO	Managed Object
MQSeries	Message Queuing Series
ODBC	Open Database Connectivity
OLE	Object Linking and Embedding
OLTP	On-line Transaction Processing
OO-SQL	Object Oriented-Structured Query Language
OMG	Object Management Group
ORB	Object Request Broker
PAA	Procedural Application Adaptor
PAO	Procedural Adaptor Object
PDA	Personal Digital Assistant
QOP	Quality of Protection
QoS	Quality of Services
RACF	Resource Access and Control Facility
RAS	Reliability, Availability, Serviceability
RDBMS	Relational Database Management System
RPC	Remote Procedure Call
RRBC	Release-to-Release Binary Compatibility
SAO	Server Administration Object
SLI	Single Logical Image
SMAPPL	Systems Management Application
SOM	System Object Model
SPI	System Programming Interface
TME	Tivoli Management Environment
TO	Transaction Object
TR	Transaction Record
UML	Unified Modeling Language
UUID	Universally Unique Identifier
VSAM	Virtual Sequential Access Method
XML	Extended Markup Language

Index

IBM Redbooks review

Your feedback is valued by the Redbook authors. In particular we are interested in situations where a Redbook "made the difference" in a task or problem you encountered. Using one of the following methods, **please review the Redbook, addressing value, subject matter, structure, depth and quality as appropriate.**

- Use the online **Contact us** review redbook form found at **ibm.com**/redbooks
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Document Number **Redbook Title**	SG24-2033-03 IBM WebSphere Application Server Enterprise Edition Component Broker 3.0: First Steps
Review	
What other subjects would you like to see IBM Redbooks address?	
Please rate your overall satisfaction:	O Very Good O Good O Average O Poor
Please identify yourself as belonging to one of the following groups:	O Customer O Business Partner O Solution Developer O IBM, Lotus or Tivoli Employee O None of the above
Your email address: The data you provide here may be used to provide you with information from IBM or our business partners about our products, services or activities.	 O Please do not use the information collected here for future marketing or promotional contacts or other communications beyond the scope of this transaction.
Questions about IBM's privacy policy?	The following link explains how we protect your personal information. **ibm.com**/privacy/yourprivacy/

IBM

Redbooks

IBM WebSphere Application Server Enterprise Edition

Component Broker 3.0: First Steps

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

IBM WebSphere Application Server Enterprise Edition

Component Broker 3.0: First Steps

Distributed object computing in a multi-tier environment

Runtime, systems management, and development

Enterprise JavaBeans and CORBA objects support

CD-ROM INCLUDED

This IBM Redbook introduces you to Component Broker object-oriented (OO) application development in a distributed, multi-tier environment.

Component Broker is a member of the IBM WebSphere Application Server Enterprise Edition product family. It provides a middle-tier application environment that allows Business Objects to be highly managed and integrated with back-end databases and transactional systems. Different types of first-tier clients can access the middle-tier Business Objects. Component Broker is designed to work with existing resource managers that provide persistence, concurrency control, and other services needed in commercial computing environments.

Component Broker consists of two parts: The Component Broker provides the runtime environment and supports systems management. The Component Broker Toolkit contains the tools that application developers use to define and implement the objects running on the middle-tier.

This IBM Redbook describes the development process and the use of the Component Broker Toolkit. It uses many incremental samples to explain how to develop applications for all kinds of application topologies.

SG24-2033-03 ISBN 0738419184